

Computer Science I: Logic and Set Theory

— Winter Term 2010/2011 —

DHBW Stuttgart

Prof. Dr. Karl Stroetmann

February 21, 2011

Contents

1	Mathematical Foundations	4
1.1	Motivation	4
1.2	Mathematical Formulæ	6
1.2.1	Why Use Mathematical Formulæ?	6
1.2.2	Mathematical Formulæ As Abbreviations	6
1.2.3	Examples	9
1.3	Sets and Relations	10
1.3.1	Enumerations	11
1.3.2	The Set of all Natural Numbers	12
1.3.3	The Axiom of Separation	12
1.3.4	The Axiom of Power Sets	12
1.3.5	Union	13
1.3.6	Intersection	13
1.3.7	Set Difference	13
1.3.8	Images	13
1.3.9	Cartesian Products	14
1.3.10	Equality of Sets	14
1.3.11	Set Algebra	15
1.4	Relations	16
1.4.1	Binary Relations and Functions	16
1.4.2	Binary Relations on a Set	23
2	SETL2	34
2.1	Introduction to SETL2	34
2.2	The Data Type Set	38
2.3	Pairs, Relations, and Functions	41
2.4	Sequences	42
2.5	Set Operators	42
2.6	Special Operators on Set and Lists	44
2.6.1	Case Study: <i>Selection Sort</i>	47
2.7	Boolean Expressions	47
2.8	Flow of Control	49
2.8.1	Case Statements	49
2.8.2	Loops	50

2.8.3	Case Study: Computation of Probabilities in Poker	53
2.9	Case Study: Graph Search	55
2.9.1	Computing the Transitive Closure	55
2.9.2	Computing the Paths	59
2.9.3	Puzzle: Crossing the River	62
2.9.4	Concluding Remarks	65
3	Propositional Logic	66
3.1	Applications of Propositional Logic	67
3.2	Semantics of Propositional Formulæ	68
3.2.1	Extensional Interpretation of Implication	70
3.2.2	Implementation in SETL2	70
3.2.3	An Application	72
3.3	Tautologies	75
3.3.1	Checking Tautologies with SETL	76
3.3.2	Conjunctive Normal Form	77
3.3.3	Implementing the Algorithm	81
3.4	Formal Proofs	87
3.4.1	Properties of the Notion of Provability	89
3.5	The Algorithm of Davis and Putnam	90
3.5.1	Simplifications via the Cut Rule	91
3.5.2	Simplification via Subsumption	92
3.5.3	Simplification via Case Distinction	92
3.5.4	An Example	93
3.5.5	Implementing the Algorithm of Davis and Putnam	95
3.6	The Eight Queens Puzzle	98
4	First-order Logic	105
4.1	Syntax	105
4.2	Semantics	109
4.2.1	Implementing First-order Structures in SETL2	114
4.3	Normal Forms for First-order Formulæ	118
4.4	Unification	123
4.5	A Calculus for First-order Logic	128
5	<i>Prolog</i>	134
5.1	Wie arbeitet <i>Prolog</i> ?	137
5.1.1	Die Tiefensuche	143
5.2	Ein komplexeres Beispiel	144
5.3	Listen	148
5.3.1	Sortieren durch Einfügen	149
5.3.2	Sortieren durch Mischen	151
5.3.3	Symbolisches Differenzieren	153
5.4	Negation in <i>Prolog</i>	158

5.4.1	Berechnung der Differenz zweier Listen	158
5.4.2	Semantik des Negations-Operators in PROLOG	159
5.5	Die Tiefen-Suche in <i>Prolog</i>	160
5.5.1	Missionare und Kannibalen	164
5.6	Das 8-Damen-Problem in <i>Prolog</i>	167
5.7	Der Cut-Operator	170
5.7.1	Verbesserung der Effizienz von <i>Prolog</i> -Programmen durch den Cut-Operator	173
5.8	Literaturhinweise	175

Chapter 1

Mathematical Foundations

Formal logic and *set theory* are at the foundation of computer science. Therefore, these two topics are the central themes of this introductory computer science lecture. Both set theory and formal logic are rather abstract and quite difficult. According to my experience, a lot of students have a hard time understanding these topics. Therefore it is necessary to motivate the use of set theory and formal logic. To this end, the following section will convince you that computer science needs a solid scientific foundation. As this lecture progresses you will see that set theory and logic can indeed provide this scientific foundation.

1.1 Motivation

Modern software and hardware system are the most complex systems ever designed by humanity. There are IT projects that take several thousand developers and several years to complete. It is obvious that the failure of such a project is connected with enormous financial losses. Let me give a few examples of software projects that illustrate this point.

1. On June 9th of 1996 the Ariane 5 rocket disintegrated on its maiden voyage. This was due to a chain of software errors:
 - A sensor of the navigation system measured the horizontal inclination and stored this number as a 64 bit floating point number.
 - This number was converted into a 16 bit fixed point number. During this conversion there was an overflow because the 64 bit number could not be represented with 16 bits.
 - As a consequence, the navigation system produced an error message, which was sent to the controlling unit.
 - The controlling unit wrongly interpreted the error message as flight data and tried to correct the flight path accordingly.
 - The resulting accelerations caused the rocket to disintegrate. Four satellites were lost, the financial loss was estimated to be a few hundred million dollars.
 - You can find a complete report on the Ariane 5 disaster at
<http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html>
2. The Therac 25 is a medical device for X-ray treatment of cancer patients. Due to a software error in 1985 at least 6 patients got a severe overdose of radiation. Three of these did not survive the overdose. A detailed description of these accidents is available at
http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac.1.html

3. During the first Gulf War, an Iraqi *Scud* missile could not be intercepted by the *Patriot* anti-aircraft system. This was due to a programming error in the *Patriot* control software. As a result, 28 soldiers lost their lives, and a hundred more were severely injured.
4. You can find a listing of more software related accidents at
<http://www.cs.tau.ac.il/~nachumd/horror.html>.

The examples given above show that the construction of IT systems requires both diligence and precision. Therefore, the development of IT systems needs a solid scientific foundation. This foundation consists of both formal logic and set theory. Besides being used as foundation, both set theory and formal logic have immediate applications in computer science.

1. Set theory and the theory of relations (which is a part of set theory) is the foundation of the theory of relational databases.
2. The programming languages *Prolog*, which is used primarily in AI¹ applications, is based on predicate logic.

Besides the immediate applications of formal logic and set theory, our engagement with these topics has another very important reason: Complex systems can only be controlled by suitable abstractions. Nobody is able to understand every detail of a program consisting of several hundred thousand lines of codes. The only way to manage a software system of that size is to introduce suitable abstractions. Therefore, a higher-than-average ability to abstract is the most important ability of a computer scientist. Our treatment of logic and set theory aims at improving this ability.

Of course, there is another very important reason that should motivate you to study both formal logic and set theory sincerely and I don't want to conceal this reason: It is the written examination at the end of this term.

Overview This lecture will cover the following topics:

1. mathematical formulæ
 Mathematical formulæ can express complex statements of affairs much more concise than descriptions given in natural language.
2. set theory
 Set theory is the foundation of both modern mathematics and computer science.
3. SETL
 The programming language SETL is directly based on set theory.
4. propositional logic
 Propositional logic analyzes the meaning of connectors like *and*, *or*, and *if ... then*.
5. predicate logic
 Predicate logic adds the *quantifiers* “*forall*” and “*exists*” to propositional logic.
6. *Prolog*
Prolog is a programming language based on predicate logic.
7. limits of computability
 Certain problems cannot be solved by algorithms. We will see that the question, whether a given program halts on a given input, is not decidable.

¹AI is short for *artificial intelligence*.

8. Hoare calculus

The *Hoare calculus* is a system that enables us to formally prove the correctness of algorithms.

1.2 Mathematical Formulæ

Mathematical formulæ are used to represent facts unambiguously. We will define mathematical formulæ as abbreviations. Let us first motivate their use.

1.2.1 Why Use Mathematical Formulæ?

Take a look at the following statement:

If we add two numbers and then square this sum, we will get the same result that we get when we square the numbers separately, add these squares and finally add the product of both numbers twice.

This text given above is nothing else than the well known formula

$$(a + b)^2 = a^2 + b^2 + 2 \cdot a \cdot b.$$

Obviously, this formula is much easier to understand than the text given above. But besides being easier to read, mathematical formulæ offer two more benefits.

1. Mathematical formulæ can be processed and analyzed by a computer program. At the present time, it is not possible for a computer to “*understand*” natural language.
2. The meaning of a mathematical formula can be defined unambiguously. In contrast, natural language is often ambiguous: Different people may interpret the same sentence differently.

1.2.2 Mathematical Formulæ As Abbreviations

Let us first introduce the ingredients that are used to build mathematical formulæ.

1. *variables*

Variables are used as names for arbitrary objects. In the example given above we used the variables *a* and *b* to denote arbitrary numbers.

2. *constants*

A constant is used to denote a fixed object. In mathematics, numbers like 1 or π are used as constants. If we were to devise a theory describing biblical genealogy, we would use constants like *Adam* and *Eve*.

Both variables and constants are known as *atomic terms*. The attribute *atomic* relates to the fact that neither variables nor constants can be subdivided into further components.

3. *function symbols*

Function symbols are used to construct complex expressions. In the example above, we have used “+”, “*”, and “²” as function symbols. Expressions that are made up from function symbols, variables and constants are called “*terms*”.

Every function symbol has an *arity*. The arity specifies the number of arguments that the function symbol needs. To give an example, the function symbol “+” expects two arguments, so its arity is two. The square root function $\sqrt{}$ is an example of a function symbol with arity 1.

Instead of saying that the function symbol *f* has arity *n* we will call *f* an *n*-ary function symbol. If a function symbol is 2-ary, it is also called a *binary* function symbol, and a 1-ary function symbol is also known as a *unary* function symbol.

4. terms

Terms denote complex objects. The notion of a term is defined inductively:

- (a) Every variable is a term.
- (b) Every constant is a term.
- (c) If f is an n -ary function symbol and we already know that t_1, t_2, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is also a term.

The notation $f(t_1, \dots, t_n)$ is called the *prefix notation*. In mathematical practice we often use an infix notation for binary function symbols. If o is a binary function symbol and t_1 and t_2 are terms, then we often write

$$t_1 \ o \ t_2 \quad \text{instead of} \quad o(t_1, t_2).$$

A well known example is the binary function symbol “+”: Of course we write $x + y$ instead of $+(x, y)$. However, when several different binary function symbols are used, we have to define a *precedence* of the different function symbols, or otherwise the interpretation of a term would be ambiguous. For example, in school mathematics the term

$$x + y * z \quad \text{is interpreted as} \quad +(x, *(y, z))$$

as the binary function symbol “*” has a higher precedence than the binary function symbol “+”.

5. predicate symbols

Predicate symbols are used to make a statement about objects. Like a function symbol, a predicate symbol has an arity which is a natural number. If p is an n -ary predicate symbol and if t_1, t_2, \dots, t_n are terms, then

$$p(t_1, t_2, \dots, t_n)$$

is an *atomic formula*. It states that the terms t_1, \dots, t_n are in the relation denoted by the predicate symbol p . To give an example, consider the two terms x and $+(x, 1)$ and the predicate symbol $<$. Then

$$<(x, +(x, 1))$$

is an *atomic formula*. Of course, this is easier to read if presented in infix notation as

$$x < x + 1.$$

While terms denote objects, atomic formulæ have a *truth value*: They are either true or false.

6. sentential connectives

Sentential connectives are also known as *logical operators*. They are used to build more complex formulæ from atomic formulæ. The simplest sentential connective is “and”. We will write the logical operator denoting “and” as “ \wedge ”. Given two formulæ F_1 and F_2 , we can connect these formulæ with the operator “ \wedge ” to build the new formula

$$F_1 \wedge F_2.$$

We read this formula as “ F_1 and F_2 ”. It states that both F_1 and F_2 hold true.

The following table lists all the logical operators that we are going to use:

operator	read as
$\neg F$	not F
$F_1 \wedge F_2$	F_1 and F_2
$F_1 \vee F_2$	F_1 or F_2
$F_1 \rightarrow F_2$	if F_1 holds, then F_2 holds
$F_1 \leftrightarrow F_2$	F_1 holds if and only if F_2 holds

In a complex formula we will have to use parenthesis to eliminate ambiguities. However, as excessive use of parenthesis renders a complex formula unreadable, we agree to assign precedences to these logical operators. The operator “ \neg ” has the highest precedence. Both “ \wedge ” and “ \vee ” have the same precedence and this precedence is higher than the precedence of “ \rightarrow ”. Finally, the precedence of “ \leftrightarrow ” is lower than any other precedence. Therefore, the formula

$$P \wedge Q \rightarrow R \leftrightarrow \neg R \rightarrow \neg P \vee \neg Q$$

is interpreted as

$$((P \wedge Q) \rightarrow R) \leftrightarrow ((\neg R) \rightarrow ((\neg P) \vee (\neg Q))).$$

In formal logic, the logical operators given above are known under the following name:

- (a) \neg : negation
- (b) \wedge : conjunction
- (c) \vee : disjunction
- (d) \rightarrow : conditional
- (e) \leftrightarrow : biconditional

In natural language, the usage of the operator “*or*” is ambiguous, because it can be used *exclusively* as in “*either A or B but not both*” or *inclusively* as in “*A or B or both*”. In mathematics, the agreement is to use the operator “*or*” inclusively, so the formula $A \vee B$ is also true if both A and B are true. We will defer a more formal treatment of the interpretation of the logical operators to a later chapter.

7. quantifiers

A quantifier specifies how a variable is used in a formula. We will use two quantifiers:

- (a) \forall (read: *for all*) is known as the *universal quantifier*. If x is a variable and F is a formula presumably containing x then

$$\forall x : F$$

is a formula that is read as

“*for all x , F is true*”

To give an example, if F is the formula $x \leq x \cdot x$, then

$$\forall x : x \leq x \cdot x$$

is a formula stating that for all x the square of x is at least as big as x . Whether the formula is actually true depends on the *universe of interpretation*: If this universe is made up of all the natural numbers, the formula is true. However, if the universe also contains the rational numbers, the formula is false as $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} < \frac{1}{2}$.

- (b) \exists (read: *exists*) is known as the *existential quantifier*. If x is a variable and F is a formula presumably containing x then

$$\exists x : F$$

is a formula that is read as

“*there exists a value for x such that F is true*”

To give an example, if F is the formula $x \cdot x = -1$, then

$$\exists x : x \cdot x = -1$$

is a formula stating that there is a number x such that the square of x is -1 . Again, the truth of this depends on the universe of interpretation: Using the real numbers,

this formula is obviously false. However, in the theory of complex numbers the formula is true.

Sometimes you will find so called “*qualified quantifiers*”. A universal qualified quantifier has the form

$$\forall x \in M : F.$$

This is read as

“*for all x from M we have F*”

Here M denotes some set. In fact, the qualified quantifier can be understood as an abbreviation:

$$\forall x \in M : F \quad \text{is the same as} \quad \forall x: (x \in M \rightarrow F).$$

In a similar way we have

$$\exists x \in M : F \quad \text{is an abbreviation for} \quad \exists x: (x \in M \wedge F).$$

It is time for an example. The formula

$$\forall x \in \mathbb{R} : \exists n \in \mathbb{N} : n > x$$

is read as follows:

For all x from \mathbb{R} there is an n from \mathbb{N} such that n is bigger than x.

If we have a formula containing both quantifiers and logical operators we have to assign a precedence to the quantifiers. Our agreement will be that quantifiers have a higher precedence than all logical operators. Therefore

$$\forall x: p(x) \wedge q(y)$$

is the same as:

$$(\forall x: p(x)) \wedge q(y).$$

1.2.3 Examples

To clarify the notions of *terms* and *formulae* we proceed to present some examples. Rather than choosing examples from mathematics we choose formulae that describe kindred-ship. We start by specifying the constants, variables, function and predicate symbols that we are going to use.

1. The following words are used as *constants*:

“**adam**”, “**eve**”, “**cain**” and “**abel**”.

2. The *variables* are the letters

“**x**”, “**y**” and “**z**”.

3. The following words are *function symbols*:

“**father**” and “**mother**”.

Both of these function symbols are unary.

4. The following words are *predicate symbols*:

“**brother**”, “**sister**”, “**uncle**”, “**male**”, and “**female**”.

In addition, we use the symbol “**=**” as a predicate symbol. All of these predicate symbols are binary.

A collection of constants, variables, function symbols, and predicate symbols together with their arity is known as a *signature*. First, we give some terms that can be build with the given signature.

1. “**cain**” is a term, as “**cain**” is a constant.
2. “**father(cain)**” is also a term, since “**cain**” is a term and “**father**” is a unary function symbol.
3. “**mother(father(cain))**” is a term as “**father(cain)**” is a term and “**mother**” is a unary function symbol,
4. “**male(cain)**” is a formula, as “**cain**” is a term and “**male**” is a unary predicate symbol.
5. “**male(lisa)**” is a formula, as “**lisa**” is a term.
6. “ $\forall x : \forall y : (\mathbf{father}(x) = \mathbf{father}(y) \wedge \mathbf{mother}(x) = \mathbf{mother}(y) \rightarrow \mathbf{brother}(x, y) \vee \mathbf{sister}(x, y))$ ” is a formula.
7. “ $\forall x : \forall y : (\mathbf{brother}(x, y) \vee \mathbf{sister}(x, y))$ ”
is a formula. Of course, this formula is wrong if we choose the obvious universe of interpretation.

For now, we treat formulæ as abbreviations. In order to be able to define the *truth* of a formula we need to make the notion of the *universe of interpretation* precise. In order to do this, we have to introduce set theory first. This is done in the remaining part of this chapter.

1.3 Sets and Relations

Set theory has been developed at the end of the 19th century in order to put mathematics on a solid foundation. This was deemed necessary as the notion of infinity gave rise to a number of paradoxes that troubled the mathematicians of that time,

Set theory was devised by Georg Cantor (1845 – 1918). The first definition of a set that was given by Cantor reads as follows [Can95]:

A *set* is a well-defined collection of *elements*.

The attribute “*well-defined*” expresses that, given a set M and an object x we must be able to decide whether x is an element of the set M . In this case we will write

$$x \in M$$

and read this formula as “ x is an element of M ”. As you can see, we use the symbol “ \in ” as a binary predicate symbol that is used with infix notation.

In order to make the notion of a *well-defined collection* precise, Cantor used the so called *comprehension axiom*. This works as follows: If $p(x)$ is a property that an object x either has or does not have, then we can build the set of all those objects, that have this property. This set is written as

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

We read this formula as

“ M is the set of all x for which $p(x)$ is true”.

Here, the property $p(x)$ is a formula that contains the variable x . Let us give an example: Let \mathbb{N} denote the set of all natural numbers. Starting from \mathbb{N} we can then define the set of all *even* natural numbers. In order to do this we have to define the property of *evenness* via a mathematical formula. Now a natural number x is even if and only if there is a natural number y , such that x

is two times y . Therefore, the property $p(x)$ can be defined as follows:

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

Now we can define the set of all even natural numbers as

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

Unfortunately, the unrestricted use of the comprehension axiom readily leads into difficulties. To see the problem, let us define the following set using the comprehension axiom:

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

This set contains all those sets that do not contain themselves. Intuitively, you might think that no set can possibly contain itself. But let us check formally whether R is a member of itself or not. Basically, there can only be two cases:

1. $\neg R \in R$.

So R does not contain itself. Now R is defined as the set of all those sets that do not contain itself. Therefore, R would be a member of itself. This contradicts our starting assumption $\neg R \in R$.

Thus we are forced to investigate the next case.

2. case: $R \in R$.

Substituting the definition of R for the second R in the formula we conclude

$$R \in \{x \mid \neg x \in x\}.$$

But this would entail that the formula $\neg x \in x$, which is the defining formula for R , holds true for R itself. Thus we conclude

$$\neg R \in R.$$

Unfortunately, this contradicts the starting assumption of the second case.

As we get a contradiction in either case, something must be wrong. The only way out is to realize that the object

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

is not a set, the comprehension axiom is way too general and has to be restricted. It was the British logician Bertrand Russell (1872 – 1970) who discovered the paradox demonstrated above. In order to avoid this and other paradoxes it is necessary to be more careful when constructing sets. Therefore we will now provide methods to construct sets that are weaker than the comprehension axiom, but that have not yet led to inconsistencies.

1.3.1 Enumerations

The simplest method to construct a set is to list all its elements. These elements will be bordered by the curly braces “{” and “}”. They are separated by commas “,”. An example is

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

which contains the elements 1, 2 and 3. As a second example, the set of all lower case letters is given as:

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

1.3.2 The Set of all Natural Numbers

If we list the elements of a set explicitly, we can never get an infinite set. However, we know that there are constructs of mathematical imagination that consist of an unbounded number of elements. The simplest such construct is the set of natural numbers. We postulate that the collection \mathbb{N} of all natural numbers is a well defined set:

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

Besides the set \mathbb{N} of all natural numbers we will use the following sets of numbers:

1. \mathbb{Z} is the set of all integers.
2. \mathbb{Q} is the set of all rational numbers.
3. \mathbb{R} is the set of all real numbers.

In mathematics, it is shown that these sets can all be derived from the set of natural numbers.

1.3.3 The Axiom of Separation

The axiom of *separation* is a weak form of the axiom of comprehension. The idea is to select those elements from an already given set that possess some property $p(x)$. We write this as:

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

Then N is the set of all those elements of M such that $p(x)$ is true.

Example: The set of even numbers can be defined as:

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

1.3.4 The Axiom of Power Sets

In order to introduce the axiom of power sets we have to define subsets first. Given two sets M and N , M is a *subset* of N if and only if every element from M is also an element of N . This is written as $M \subseteq N$. Formally this can be defined as:

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

Now the *power set* of a given set M is the set of all subsets of M . It is written as 2^M . Formally we have

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

Example: We compute the power set of the set $\{1, 2, 3\}$ and find:

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

This set has $8 = 2^3$ elements. It can be shown that the power set of a set containing n elements will itself contain 2^n elements. If we denote the number of elements of a finite set with $\text{card}(M)$, we therefore have

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

This is the motivation for the denoting the power set as 2^M .

1.3.5 Union

Given two sets M and N , the union of M and N contains all elements that are either in M or in N . We will write $M \cup N$ to denote the union of M and N . Then we have:

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

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

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

The notion of the union of two sets can be extended to the notion of a union of a set of sets. So imagine a set X all of whose elements are itself sets. We can then construct the union of all those sets contained in X . This union is written as $\bigcup X$. Formally, we have:

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

Example: Assume that the set X is defined as follows:

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

Then we have

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

1.3.6 Intersection

The *intersection* of two sets M and N is defined as the set that contains all those objects that are elements of both M and N . The intersection of M and N is denoted as $M \cap N$. More formally, $M \cap N$ is defined as follows:

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

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

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

1.3.7 Set Difference

Given two sets M and N , the *set difference* of M without N is the set of all those elements of M that are not elements of N . The set difference is written as $M \setminus N$. This is read as M *without* N . The formal definition is:

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

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

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

1.3.8 Images

If M is a set and f is a function that is defined for all x from M , then for all $x \in M$ we call $f(x)$ the *image* of x under f . The set of all images from elements from M is called the *image* of M under f and is denoted as $f(M)$. Formally, the image $f(M)$ is defined as follows:

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

Equivalently, we can write

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

Example: The set of all integer squares is defined as

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

1.3.9 Cartesian Products

In order to introduce the notion of the Cartesian product of two sets we first need the notion of an *ordered pair* of two objects x and y . The ordered pair of x and y is written as

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

The object x is the *first component* of the pair $\langle x, y \rangle$, 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 the respective components are equal. Therefore we have

$$\langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle \quad \text{iff}^2 \quad x_1 = x_2 \wedge y_1 = y_2.$$

The *Cartesian product* of two sets M and N is defined as the set of all ordered pairs p such that the first component of p is an element of M and the second component is an element of N . The Cartesian product is written as $M \times N$. Formally, we have

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

This can also be written as

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

The Cartesian product is also known as the *product set*.

Example: Let $M = \{1, 2, 3\}$ and $N = \{5, 7\}$. Then

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

The notion of an ordered pair is easily extended to the notion of a finite sequence: A finite sequence of length n has the form

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

Finite sequences are also known as *lists*. Finite sequences enable us to extend the notion of product sets to more than two sets. We define the product set of the sets M_1, \dots, M_n as

$$M_1 \times \dots \times M_n = \{\langle x_1, x_2, \dots, x_n \rangle \mid x_1 \in M_1 \wedge \dots \wedge x_n \in M_n\}.$$

If f is a function defined on $M_1 \times \dots \times M_n$, then we agree to simplify our notation as follows: We write

$$f(x_1, \dots, x_n) \quad \text{instead of} \quad f(\langle x_1, \dots, x_n \rangle).$$

1.3.10 Equality of Sets

Two sets are considered equal if and only if they have the same elements. Therefore we have:

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

Therefore the order in which the elements of a set are listed, is unimportant. For example, we have

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

since both sets contain the same elements.

If sets are defined by explicitly listing their elements, then the question, whether two sets are equal, is straightforward to answer. However, in general it may be arbitrarily difficult to decide whether two sets are equal. For example, it can be shown that

²In computer science and mathematics, it is customary to use “*iff*” as an abbreviation for “*if and only if*”.

$$\{n \in \mathbb{N} \mid \exists x, y, z \in \mathbb{N} : x > 0 \wedge y > 0 \wedge x^n + y^n = z^n\} = \{1, 2\}.$$

However the proof of this claim is equivalent to *Fermat's Last Theorem*, which was postulated in 1637 by *Pierre de Fermat*. This proof could only be given in 1995 by Andrew Wiles. In general, it can be shown that the question, whether two sets are equal, is algorithmically undecidable.

1.3.11 Set Algebra

The set operations “ \cup ”, “ \cap ”, and “ \setminus ” satisfy a number of equations. These are given below. As is customary, the empty set $\{\}$ is denoted as \emptyset .

- | | |
|--|---|
| 1. $M \cup \emptyset = M$ | $M \cap \emptyset = \emptyset$ |
| 2. $M \cup M = M$ | $M \cap M = M$ |
| 3. $M \cup N = N \cup M$ | $M \cap N = N \cap M$ |
| 4. $(K \cup M) \cup N = K \cup (M \cup N)$ | $(K \cap M) \cap N = K \cap (M \cap N)$ |
| 5. $(K \cup M) \cap N = (K \cap N) \cup (M \cap N)$ | $(K \cap M) \cup N = (K \cup N) \cap (M \cup N)$ |
| 6. $M \setminus \emptyset = M$ | $M \setminus M = \emptyset$ |
| 7. $K \setminus (M \cup N) = (K \setminus M) \cap (K \setminus N)$ | $K \setminus (M \cap N) = (K \setminus M) \cup (K \setminus N)$ |
| 8. $(K \cup M) \setminus N = (K \setminus N) \cup (M \setminus N)$ | $(K \cap M) \setminus N = (K \setminus N) \cap (M \setminus N)$ |
| 9. $K \setminus (M \setminus N) = (K \setminus M) \cup (K \cap N)$ | $(K \setminus M) \setminus N = K \setminus (M \cup N)$ |
| 10. $M \cup (N \setminus M) = M \cup N$ | $M \cap (N \setminus M) = \emptyset$ |
| 11. $M \cup (M \cap N) = M$ | $M \cap (M \cup N) = M$ |

In order to show how these equations can be proved, we demonstrate the validity of the equation

$$K \setminus (M \cup N) = (K \setminus M) \cap (K \setminus N).$$

In order to prove the equality of two sets M and N we have to show that M and N contain the same elements. We have the following chain of equivalences:

$$\begin{aligned}
 & x \in K \setminus (M \cup N) \\
 \leftrightarrow & x \in K \wedge \neg x \in M \cup N \\
 \leftrightarrow & x \in K \wedge \neg (x \in M \vee x \in N) \\
 \leftrightarrow & x \in K \wedge (\neg x \in M) \wedge (\neg x \in N) \\
 \leftrightarrow & (x \in K \wedge \neg x \in M) \wedge (x \in K \wedge \neg x \in N) \\
 \leftrightarrow & (x \in K \setminus M) \wedge (x \in K \setminus N) \\
 \leftrightarrow & x \in (K \setminus M) \cap (K \setminus N).
 \end{aligned}$$

The remaining equations can be verified in the same way.

In order to simplify our proofs we introduce the following notation: If M is a set and x is an object, we write $x \notin M$ instead of $\neg x \in M$:

$$x \notin M \stackrel{\text{def}}{\iff} \neg x \in M.$$

We use an analogous notation for equality:

$$x \neq y \stackrel{\text{def}}{\iff} \neg (x = y).$$

Exercise: Prove the equation $(K \cup M) \cap N = (K \cap N) \cup (M \cap N)$.

1.4 Relations

Relations abound in computer science. A very important application of relations is found in the theory of relational databases. We start our investigation of relations with the special case of *binary relations*. We will investigate the connection between binary relations and functions. We will see, that functions are a special case of binary relations. Turned around, this means that the notion of a binary relation is a generalization of the notion of a function.

Furthermore, we will discuss order relations and equivalence relations.

1.4.1 Binary Relations and Functions

If a set R is a subset of the Cartesian product $M \times N$ of two sets M and N , that is if we have

$$R \subseteq M \times N,$$

then R is called a *binary relation*. In this case, the *domain* of R is defined as

$$\text{dom}(R) := \{x \mid \exists y \in N: \langle x, y \rangle \in R\},$$

while the *range* of R is defined as

$$\text{rng}(R) := \{y \mid \exists x \in M: \langle x, y \rangle \in R\}.$$

Example: Define $R = \{\langle 1, 1 \rangle, \langle 2, 4 \rangle, \langle 3, 9 \rangle\}$. Then we have

$$\text{dom}(R) = \{1, 2, 3\} \quad \text{and} \quad \text{rng}(R) = \{1, 4, 9\}.$$

□

Example: The database of a car dealer stores facts concerning customers and the cars they bought. Let us assume that the sets *Car* and *Customer* are defined as follows:

$$\text{Customer} = \{\text{Bauer, Maier, Schmidt}\} \quad \text{and} \quad \text{Car} = \{\text{Polo, Fox, Golf}\}.$$

Then the binary relation

$$\text{Sales} \subseteq \text{Customer} \times \text{Car}$$

could be given as follows:

$$\{\langle \text{Bauer, Golf} \rangle, \langle \text{Bauer, Fox} \rangle, \langle \text{Schmidt, Polo} \rangle\}.$$

This relation is interpreted as follows:

- Mr. Bauer bought both a Golf and a Fox.
- Mr. Schmidt has bought a Polo.
- Mr. Maier hasn't bought a car so far.

In the theory of databases, this information is expressed by a table as follows:

<i>Customer</i>	<i>Car</i>
Bauer	Golf
Bauer	Fox
Schmidt	Polo

The header row containing the column captions *Customer* and *Car* is not part of the relation but is the *relation schema*. The relation together with its schema is called a *table*.

Left-Unique and Right-Unique Relations A relation $R \subseteq M \times N$ is called *right-unique* iff the following holds:

$$\forall x \in M: \forall y_1, y_2 \in N: (\langle x, y_1 \rangle \in R \wedge \langle x, y_2 \rangle \in R \rightarrow y_1 = y_2).$$

If the relation $R \subseteq M \times N$ is right-unique, then for every $x \in M$ there is at most one $y \in N$ such that $\langle x, y \rangle \in R$ holds. A right-unique relation is also known as a *functional relation*.

Analogously a relation $R \subseteq M \times N$ *left-unique* iff we have:

$$\forall y \in N: \forall x_1, x_2 \in M: (\langle x_1, y \rangle \in R \wedge \langle x_2, y \rangle \in R \rightarrow x_1 = x_2).$$

Therefore, if $R \subseteq M \times N$ is left-unique, then for every $y \in N$ there is at most one $x \in M$ such that $\langle x, y \rangle \in R$ gilt. A left-unique relation is also known as an *injective relation*.

Example: Let $M = \{1, 2, 3\}$ and $N = \{4, 5, 6\}$.

1. Assume that the relation R_1 is defined as

$$R_1 = \{\langle 1, 4 \rangle, \langle 1, 6 \rangle\}.$$

This relation is not right-unique, since $4 \neq 6$. However, this relation is left-unique as the right hand sides of all tuples in R_1 are different.

2. Assume that the relation R_2 is defined as

$$R_2 = \{\langle 1, 4 \rangle, \langle 2, 6 \rangle\}.$$

This relation is both left-unique and right-unique.

3. Assume the relation R_3 is defined as

$$R_3 = \{\langle 1, 4 \rangle, \langle 2, 6 \rangle, \langle 3, 6 \rangle\}.$$

This relation is right-unique, but it is not left-unique, because we have $\langle 2, 6 \rangle \in R$ and $\langle 3, 6 \rangle \in R$, but $2 \neq 3$.

Left-Total and Right-Total Relations A binary relation $R \subseteq M \times N$ is *left-total on M* iff

$$\forall x \in M: \exists y \in N: \langle x, y \rangle \in R.$$

That means for every x from M there is a y from N such that $\langle x, y \rangle$ is a member of the relation R . In the example above, the relation R_3 is left-total.

A binary relation $R \subseteq M \times N$ is *right-total on N* iff

$$\forall y \in N: \exists x \in M: \langle x, y \rangle \in R.$$

That means for every y from N there is an x from M such that the pair $\langle x, y \rangle$ is a member of the relation R .

Functions If a relation $R \subseteq M \times N$ is both left-total on M and right-unique, then R is a *function* on M . The reason for this definition is that in this case we can define a function $f_R: M \rightarrow N$ as follows:

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

This definition works, because the fact that R is left-total ensures that for every $x \in M$ there is an $y \in N$ such that $\langle x, y \rangle \in R$. Therefore, f_R is defined for all x from the set M . The fact that R is right-unique ensures that we find at most one y such that $\langle x, y \rangle \in R$, so $f_R(x)$ is indeed well-defined.

On the other hand, if a function $f: M \rightarrow N$ is given, we can define a relation $\text{graph}(f) \subseteq M \times N$ as follows.

$$\text{graph}(f) := \{\langle x, f(x) \rangle \mid x \in M\}.$$

The relation $\text{graph}(f)$ is left-total because the function f computes a result $f(x)$ for every $x \in M$ and the relation is right-unique because for a given argument $x \in M$ there is at most one value $f(x)$.

Therefore functions are just a special case of relations. The set of all relations on $M \times N$ that are functions on M is written as N^M , we have

$$N^M := \{R \subseteq M \times N \mid R \text{ is a function on } M\}.$$

This notation is explained as follows: If M and N are finite sets with m and n elements, then there are n^m different functions from M to N , we have

$$\text{card}(N^M) = \text{card}(N)^{\text{card}(M)}.$$

In the following, we make no distinction between a relation that is both left-total on M and right-unique and the corresponding function. If $R \subseteq M \times N$ is left-total on M and right-unique and $x \in M$, then we will write $R(x)$ to denote the uniquely determined value $y \in N$ that satisfies $\langle x, y \rangle \in R$.

Example:

1. Let $M = \{1, 2, 3\}$, $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and define

$$R := \{\langle 1, 1 \rangle, \langle 2, 4 \rangle, \langle 3, 9 \rangle\}.$$

Then R is a function on M . This function computes the squares on the set M .

2. This time we have $M = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $N = \{1, 2, 3\}$ and define

$$R := \{\langle 1, 1 \rangle, \langle 4, 2 \rangle, \langle 9, 3 \rangle\}.$$

R is not a function on M , as R is not left-total on M . For example, the number 2 is not mapped by R to an element of N .

3. Let $M = \{1, 2, 3\}$, $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and define

$$R := \{\langle 1, 1 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 9 \rangle\}$$

R is not a function on M , as R is not right-unique. For example, the element 2 is mapped both to 3 and also to 4.

If $R \subseteq M \times N$ is a binary relation and $X \subseteq M$, then we define the *image of X under R* as

$$R(X) := \{y \mid \exists x \in X: \langle x, y \rangle \in R\}.$$

Inverse Relation Given a relation $R \subseteq M \times N$, we define the *inverse relation* $R^{-1} \subseteq N \times M$ as follows:

$$R^{-1} := \{\langle y, x \rangle \mid \langle x, y \rangle \in R\}.$$

Therefore R^{-1} is right-unique iff R is left-unique. Furthermore, R^{-1} is left-total on N iff R is right-total on N . If a relation R is both left-unique and right-unique and furthermore both left-total on M and right-total on N , then R is called *bijective*. In this case, we can define two functions:

$$f_R : M \rightarrow N \quad \text{and} \quad f_{R^{-1}} : N \rightarrow M$$

The definition of $f_{R^{-1}}$ reads:

$$f_{R^{-1}}(y) := x \stackrel{\text{def}}{\iff} \langle y, x \rangle \in R^{-1} \iff \langle x, y \rangle \in R.$$

The function $f_{R^{-1}}$ is the *inverse function* of f_R , as we have

$$\forall y \in N: f_R(f_{R^{-1}}(y)) = y \quad \text{and} \quad \forall x \in M: f_{R^{-1}}(f_R(x)) = x.$$

This justifies the notation R^{-1} for the inverse relation.

Composition of Relations As functions can be composed, so can relations be composed. Assume L , M and N are sets. If we have two relations $R \subseteq L \times M$ and $Q \subseteq M \times N$, then the *relational product* $R \circ Q$ is defined as follows:

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

The relational product $R \circ Q$ is also known as the *composition* of Q and R . In the theory of relational databases, you will meet the composition of relations again in the more general form of a *join*.

Example: Let $L = \{1, 2, 3\}$, $M = \{4, 5, 6\}$ and $N = \{7, 8, 9\}$. Furthermore, the relations Q and R are given as follows:

$$R = \{ \langle 1, 4 \rangle, \langle 1, 6 \rangle, \langle 3, 5 \rangle \} \quad \text{and} \quad Q = \{ \langle 4, 7 \rangle, \langle 6, 8 \rangle, \langle 6, 9 \rangle \}.$$

Then we have

$$R \circ Q = \{ \langle 1, 7 \rangle, \langle 1, 8 \rangle, \langle 1, 9 \rangle \}.$$

If $R \subseteq L \times M$ is a function on L and $Q \subseteq M \times N$ is a function on M , then $R \circ Q$ is a function on L and the function $f_{R \circ Q}$ can be computed as follows from f_R and f_Q :

$$f_{R \circ Q}(x) = f_Q(f_R(x)).$$

Observe that the order, in which R and Q appear on the left hand side of this equation, is different than the order of R and Q on the right hand side.

Remark: You should be aware that some text books define the relational product $R \circ Q$ as $Q \circ R$. In that case, the definition of $R \circ Q$ reads as follows: If $R \subseteq M \times N$ and $Q \subseteq L \times M$, then

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

The definition used in this lecture is more intuitive when we compute the transitive closure of a binary relation. □

Example: The next example demonstrates the use of the composition of two relations in the context of a database. Let us assume that the database of a car dealer contains the following two tables.

Purchase:

<i>Customer</i>	<i>Car</i>
Bauer	Golf
Bauer	Fox
Schmidt	Polo

Price:

<i>Car</i>	<i>Amount</i>
Golf	20 000
Fox	10 000
Polo	13 000

Then the relational product of $Purchase \circ Price$ is given as follows:

<i>Car</i>	<i>Amount</i>
Bauer	20 000
Bauer	10 000
Schmidt	13 000

This relation could be used for invoicing practice. □

Properties of the Relational Product The composition of relations is *associative*: If

$$R \subseteq K \times L, \quad Q \subseteq L \times M \quad \text{and} \quad P \subseteq M \times N$$

are binary relations, then we have

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

Proof: We show

$$\langle x, u \rangle \in (R \circ Q) \circ P \leftrightarrow \langle x, u \rangle \in R \circ (Q \circ P) \quad (1.1)$$

First, we manipulate the left hand side $\langle x, u \rangle \in (R \circ Q) \circ P$ of (1.1). We have

$$\begin{aligned} \langle x, u \rangle &\in (R \circ Q) \circ P \\ \leftrightarrow \exists z : (\langle x, z \rangle \in R \wedge \langle z, u \rangle \in P) &\quad \text{definition of } (R \circ Q) \circ P \\ \leftrightarrow \exists z : ((\exists y : \langle x, y \rangle \in R \wedge \langle y, z \rangle \in Q) \wedge \langle z, u \rangle \in P) &\quad \text{definition of } R \circ Q \end{aligned}$$

As the variable y does not occur in the formula $\langle z, u \rangle \in P$, the existential quantifier $\exists y$ can be lifted as follows:

$$\exists z : \exists y : (\langle x, y \rangle \in R \wedge \langle y, z \rangle \in Q \wedge \langle z, u \rangle \in P) \quad (1.2)$$

Now we manipulate the right hand side of (1.1):

$$\begin{aligned} \langle x, u \rangle &\in R \circ (Q \circ P) \\ \leftrightarrow \exists y : (\langle x, y \rangle \in R \wedge \langle y, u \rangle \in Q \circ P) &\quad \text{definition of } R \circ (Q \circ P) \\ \leftrightarrow \exists y : (\langle x, y \rangle \in R \wedge \exists z : (\langle y, z \rangle \in Q \wedge \langle z, u \rangle \in P)) &\quad \text{definition of } Q \circ P \end{aligned}$$

As the variable z does not occur in the formula $\langle x, y \rangle \in R$, the existential quantifier with respect to z can be lifted and we get

$$\exists z : \exists y : (\langle x, y \rangle \in R \wedge \langle y, z \rangle \in Q \wedge \langle z, u \rangle \in P) \quad (1.3)$$

Interchanging the order of the existential quantifiers we arrive at

$$\exists z : \exists y : (\langle x, y \rangle \in R \wedge \langle y, z \rangle \in Q \wedge \langle z, u \rangle \in P) \quad (1.4)$$

Now the formulæ (1.2) and (1.4) are identical. Therefore we have shown the equivalence (1.1) and the law of associativity is proved. \square

Another important property of the relational product is the following: If $R \subseteq L \times M$ and $Q \subseteq M \times N$ are two relations, then we have

$$(R \circ Q)^{-1} = Q^{-1} \circ R^{-1}.$$

You should note that the sequence of Q and R is different on the left side of this equation from the order in which Q and R appear on the right side. To prove this equation we have to show that for all pairs $\langle z, x \rangle \in N \times L$ the equivalence

$$\langle z, x \rangle \in (R \circ Q)^{-1} \leftrightarrow \langle z, x \rangle \in R^{-1} \circ Q^{-1}$$

is valid. The proof is given as follows:

$$\begin{aligned} \langle z, x \rangle &\in (R \circ Q)^{-1} \\ \leftrightarrow \langle x, z \rangle &\in R \circ Q \\ \leftrightarrow \exists y \in M : (\langle x, y \rangle \in R \wedge \langle y, z \rangle \in Q) \\ \leftrightarrow \exists y \in M : (\langle y, z \rangle \in Q \wedge \langle x, y \rangle \in R) \\ \leftrightarrow \exists y \in M : (\langle z, y \rangle \in Q^{-1} \wedge \langle y, x \rangle \in R^{-1}) \\ \leftrightarrow \langle z, x \rangle &\in Q^{-1} \circ R^{-1} \quad \square \end{aligned}$$

We note the following law of distributivity: If R_1 and R_2 are relations on $L \times M$ and Q is a relation on $M \times N$, then we have:

$$(R_1 \cup R_2) \circ Q = (R_1 \circ Q) \cup (R_2 \circ Q).$$

In a similar way we have

$$R \circ (Q_1 \cup Q_2) = (R \circ Q_1) \cup (R \circ Q_2),$$

provided R is a relation on $L \times M$ and Q_1 and Q_2 are relations on $M \times N$. In order to be able to give a concise proof of the first equation we agree that the composition operator \circ has a higher precedence than both \cup and \cap . We prove the first law of distributivity by showing

$$\langle x, z \rangle \in (R_1 \cup R_2) \circ Q \leftrightarrow \langle x, z \rangle \in R_1 \circ Q \cup R_2 \circ Q. \quad (1.5)$$

To this end, the formula $\langle x, z \rangle \in (R_1 \cup R_2) \circ Q$ is manipulated as follows:

$$\begin{aligned} & \langle x, z \rangle \in (R_1 \cup R_2) \circ Q \\ \leftrightarrow & \exists y : (\langle x, y \rangle \in R_1 \cup R_2 \wedge \langle y, z \rangle \in Q) && \text{definition of } (R_1 \cup R_2) \circ Q \\ \leftrightarrow & \exists y : ((\langle x, y \rangle \in R_1 \vee \langle x, y \rangle \in R_2) \wedge \langle y, z \rangle \in Q) && \text{definition of } R_1 \cup R_2 \end{aligned}$$

This formula is manipulated using the law of distributivity from propositional logic. Later, in propositional logic we will show that for arbitrary formulæ F_1 , F_2 and G the equivalence

$$(F_1 \vee F_2) \wedge G \leftrightarrow (F_1 \wedge G) \vee (F_2 \wedge G)$$

holds. Using this law we get:

$$\begin{aligned} & \exists y : \left(\underbrace{(\langle x, y \rangle \in R_1)}_{F_1} \vee \underbrace{(\langle x, y \rangle \in R_2)}_{F_2} \right) \wedge \underbrace{\langle y, z \rangle \in Q}_G \\ \leftrightarrow & \exists y : \left(\underbrace{(\langle x, y \rangle \in R_1)}_{F_1} \wedge \underbrace{\langle y, z \rangle \in Q}_G \right) \vee \left(\underbrace{(\langle x, y \rangle \in R_2)}_{F_2} \wedge \underbrace{\langle y, z \rangle \in Q}_G \right) \end{aligned} \quad (1.6)$$

Next, we manipulate $\langle x, z \rangle \in R_1 \circ Q \cup R_2 \circ Q$:

$$\begin{aligned} & \langle x, z \rangle \in R_1 \circ Q \cup R_2 \circ Q \\ \leftrightarrow & \langle x, z \rangle \in R_1 \circ Q \vee \langle x, z \rangle \in R_2 \circ Q && \text{definition of } \cup \\ \leftrightarrow & (\exists y : (\langle x, y \rangle \in R_1 \wedge \langle y, z \rangle \in Q)) \vee (\exists y : (\langle x, y \rangle \in R_2 \wedge \langle y, z \rangle \in Q)) \\ & \text{definition of } R_1 \circ Q \text{ and } R_2 \circ Q \end{aligned}$$

This formula is now manipulated using a law of distributivity for predicate logic. In the chapter on predicate logic we will see that for arbitrary formulæ F_1 and F_2 the equivalence

$$\exists y : (F_1 \vee F_2) \leftrightarrow (\exists y : F_1) \vee (\exists y : F_2)$$

holds. Then we conclude

$$\begin{aligned} & \exists y : \left(\underbrace{(\langle x, y \rangle \in R_1 \wedge \langle y, z \rangle \in Q)}_{F_1} \right) \vee \exists y : \left(\underbrace{(\langle x, y \rangle \in R_2 \wedge \langle y, z \rangle \in Q)}_{F_2} \right) \\ \leftrightarrow & \exists y : \left(\underbrace{(\langle x, y \rangle \in R_1 \wedge \langle y, z \rangle \in Q)}_{F_1} \right) \vee \left(\underbrace{(\langle x, y \rangle \in R_2 \wedge \langle y, z \rangle \in Q)}_{F_2} \right) \end{aligned} \quad (1.7)$$

As (1.6) and (1.7) are identical, the proof is complete. \square

Remark: It is interesting to note that for the intersection and the composition operator there is no law of distributivity, the equation

$$(R_1 \cap R_2) \circ Q = R_1 \circ Q \cap R_2 \circ Q$$

does not hold in general. In order to validate this claim we provide a counter example. Let us define the relations R_1 , R_2 and Q as follows:

$$R_1 := \{\langle 1, 2 \rangle\}, \quad R_2 := \{\langle 1, 3 \rangle\} \quad \text{and} \quad Q = \{\langle 2, 4 \rangle, \langle 3, 4 \rangle\}.$$

Then we have

$$R_1 \circ Q = \{\langle 1, 4 \rangle\}, \quad R_2 \circ Q = \{\langle 1, 4 \rangle\}, \quad \text{therefore} \\ R_1 \circ Q \cap R_2 \circ Q = \{\langle 1, 4 \rangle\}.$$

But on the other hand, we have

$$(R_1 \cap R_2) \circ Q = \emptyset \circ Q = \emptyset \neq \{\langle 1, 4 \rangle\} = R_1 \circ Q \cap R_2 \circ Q. \quad \square$$

Identical Relation If M is a set, we define the *identical relation* $\text{id}_M \subseteq M \times M$ as follows:

$$\text{id}_M := \{\langle x, x \rangle \mid x \in M\}.$$

Example: Let $M = \{1, 2, 3\}$. Then

$$\text{id}_M := \{\langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 3 \rangle\}.$$

An immediate consequence of this definition is

$$\text{id}_M^{-1} = \text{id}_M.$$

If $R \subseteq M \times N$ is a binary relation we have

$$R \circ \text{id}_N = R \quad \text{and} \quad \text{id}_M \circ R = R.$$

We prove the second equation. According to the definition of the composition we have

$$\text{id}_M \circ R = \{\langle x, z \rangle \mid \exists y : \langle x, y \rangle \in \text{id}_M \wedge \langle y, z \rangle \in R\}.$$

Now we have $\langle x, y \rangle \in \text{id}_M$ if and only if $x = y$, therefore

$$\text{id}_M \circ R = \{\langle x, z \rangle \mid \exists y : x = y \wedge \langle y, z \rangle \in R\}.$$

Furthermore, we have

$$\exists y : (x = y \wedge \langle y, z \rangle \in R) \leftrightarrow \langle x, z \rangle \in R.$$

This is easy to see: If we have $\exists y : x = y \wedge \langle y, z \rangle \in R$, the y must be equal to x and then we have $\langle x, z \rangle \in R$. If on the other hand $\langle x, z \rangle \in R$, define $y := x$. For this y we obviously have $x = y \wedge \langle y, z \rangle \in R$. Using the equivalence given above we conclude

$$\text{id}_M \circ R = \{\langle x, z \rangle \mid \langle x, z \rangle \in R\}.$$

As $R \subseteq M \times N$ we know that R is a set of ordered pairs and therefore

$$R = \{\langle x, z \rangle \mid \langle x, z \rangle \in R\}.$$

This proves $\text{id}_M \circ R = R$. \square

Exercise 1: Assume $R \subseteq M \times N$. State conditions for the relation R such that

$$R \circ R^{-1} = \text{id}_M$$

holds. State conditions such that

$$R^{-1} \circ R = \text{id}_M$$

holds. \square

1.4.2 Binary Relations on a Set

In the following, we discuss the special case of those relations $R \subseteq M \times N$ that satisfy $M = N$. We define: A relation $R \subseteq M \times M$ is a relation *on* the set M . In the rest of this section we restrict ourself to discuss relations of this kind. Instead of $R \subseteq M \times M$ we sometimes write $R \subseteq M^2$. Furthermore, if R is a relations on M and if $x, y \in M$, we sometimes write xRy instead of $\langle x, y \rangle \in R$ and thereby treat R as an infix operator. For example, the relation \leq on \mathbb{N} is defined as follows:

$$\leq := \{\langle x, y \rangle \in \mathbb{N} \times \mathbb{N} \mid \exists z \in \mathbb{N}: x + z = y\}.$$

Instead of $\langle x, y \rangle \in \leq$ it is customary to write $x \leq y$.

Definition 1 The relation $R \subseteq M \times M$ is called *reflexive* on M iff

$$\forall x \in M: \langle x, x \rangle \in R.$$

Proposition 2 The relation $R \subseteq M \times M$ is reflexive on M iff $\text{id}_M \subseteq R$.

Proof: We have

$$\begin{aligned} & \text{id}_M \subseteq R \\ \text{iff } & \{\langle x, x \rangle \mid x \in M\} \subseteq R \\ \text{iff } & \forall x \in M: \langle x, x \rangle \in R \\ \text{iff } & R \text{ is reflexive.} \end{aligned}$$

□

Definition 3 The relation $R \subseteq M \times M$ is *symmetric* iff

$$\forall x, y \in M: (\langle x, y \rangle \in R \rightarrow \langle y, x \rangle \in R).$$

Proposition 4 The relation $R \subseteq M \times M$ is symmetric iff $R^{-1} \subseteq R$.

Proof: Let us rewrite the formula $R^{-1} \subseteq R$ by substituting

$$R^{-1} = \{\langle y, x \rangle \mid \langle x, y \rangle \in R\}.$$

Then $R^{-1} \subseteq R$ is equivalent to

$$\{\langle y, x \rangle \mid \langle x, y \rangle \in R\} \subseteq R.$$

According to the definition of \subseteq this is the same as

$$\forall x, y \in M: (\langle x, y \rangle \in R \rightarrow \langle y, x \rangle \in R)$$

and this is the condition for R being symmetric. □

Definition 5 The relation $R \subseteq M \times M$ is *anti-symmetric* iff

$$\forall x, y \in M: (\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R \rightarrow x = y).$$

Proposition 6 The relation $R \subseteq M \times M$ is anti-symmetric iff $R \cap R^{-1} \subseteq \text{id}_M$.

Proof: We split the proof into two parts. Let us first assume that R is anti-symmetric and therefore

$$\forall x, y \in M: (\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R \rightarrow x = y)$$

holds. We show that $R \cap R^{-1} \subseteq \text{id}_M$ is a consequence of this assumption. Assume $\langle x, y \rangle \in R \cap R^{-1}$. Then $\langle x, y \rangle \in R$ and from $\langle x, y \rangle \in R^{-1}$ we know $\langle y, x \rangle \in R$. From our assumption we can then conclude $x = y$. This implies $\langle x, y \rangle \in \text{id}_M$, showing $R \cap R^{-1} \subseteq \text{id}_M$.

To prove the remaining part, assume $R \cap R^{-1} \subseteq \text{id}_M$. We have to show that this implies

$$\forall x, y \in M: (\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R \rightarrow x = y).$$

Therefore, take $x, y \in M$ such that $\langle x, y \rangle \in R$ and $\langle y, x \rangle \in R$. We have to show $x = y$. From $\langle y, x \rangle \in R$ we have $\langle x, y \rangle \in R^{-1}$. Therefore $\langle x, y \rangle \in R \cap R^{-1}$. As we have assumed $R \cap R^{-1} \subseteq \text{id}_M$, we can conclude $\langle x, y \rangle \in \text{id}_M$ and that implies $x = y$. \square

Definition 7 The relation $R \subseteq M \times M$ is *asymmetric* iff

$$\forall x, y \in M: \neg(\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R).$$

Although the names are similar, the concept of an asymmetric relation is quite different from the concept of an anti-symmetric relation. Therefore, it is important not to confound these notions.

Definition 8 The relation $R \subseteq M \times M$ is *transitive* iff

$$\forall x, y, z \in M: (\langle x, y \rangle \in R \wedge \langle y, z \rangle \in R \rightarrow \langle x, z \rangle \in R).$$

Proposition 9 The relation $R \subseteq M \times M$ is transitive iff $R \circ R \subseteq R$.

Proof: Let us first assume that R is transitive. We have to show that this implies $R \circ R \subseteq R$. In order to show $R \circ R \subseteq R$ assume $\langle x, z \rangle \in R \circ R$. We have to show $\langle x, z \rangle \in R$. According to the definition of the composition $R \circ R$ the assumption $\langle x, z \rangle \in R \circ R$ entails that there exists a y such that we have

$$\langle x, y \rangle \in R \quad \text{and} \quad \langle y, z \rangle \in R.$$

As we have assumed R transitive this implies $\langle x, z \rangle \in R$.

To prove the second part, let us assume $R \circ R \subseteq R$. We have to show that this implies

$$\forall x, y, z \in M: (\langle x, y \rangle \in R \wedge \langle y, z \rangle \in R \rightarrow \langle x, z \rangle \in R).$$

In order to show this, assume $\langle x, y \rangle \in R$ and $\langle y, z \rangle \in R$. According to the definition of the relational product, this implies $\langle x, z \rangle \in R \circ R$ and using the assumption $R \circ R \subseteq R$ we conclude $\langle x, z \rangle \in R$. \square

Examples: For the first two examples, define $M = \{1, 2, 3\}$.

1. $R_1 = \{\langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 3 \rangle\}$.

R_1 is reflexive on M , symmetric, anti-symmetric, and transitive. Note that R_1 is not asymmetric as we can set $x := 1$ and $y := 1$ and then have both $\langle x, y \rangle \in R$ and $\langle y, x \rangle \in R$.

2. $R_2 = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 3, 3 \rangle\}$.

R_2 is not reflexive on M , as $\langle 1, 1 \rangle \notin R_2$. R_2 is symmetric. R_2 isn't anti-symmetric, because we have both $\langle 1, 2 \rangle \in R_2$ and $\langle 2, 1 \rangle \in R_2$, but $2 \neq 1$. As $\langle 3, 3 \rangle \in R$, R isn't asymmetric either. Finally, R_2 isn't transitive, because if it were, then the facts $\langle 1, 2 \rangle \in R_2$ and $\langle 2, 1 \rangle \in R_2$ would imply $\langle 1, 1 \rangle \in R_2$, which is wrong.

In the following examples we have $M = \mathbb{N}$.

3. $R_3 := \{\langle n, m \rangle \in \mathbb{N}^2 \mid n \leq m\}$.

As we have $n \leq n$ for all $n \in \mathbb{N}$, R_3 is reflexive on \mathbb{N} . R_3 is not symmetric: For example, we have $1 \leq 2$ but we don't have $2 \leq 1$. R_3 is anti-symmetric: If we have both $n \leq m$ and $m \leq n$, then we can conclude $m = n$. R_3 is not asymmetric, as we have $1 \leq 1$. Finally, R_3 is transitive: If we have $k \leq m$ and $m \leq n$, then we can conclude $k \leq n$.

$$4. R_4 := \{ \langle m, n \rangle \in \mathbb{N}^2 \mid \exists k \in \mathbb{N} : m \cdot k = n \}$$

For two positive integers m and n we have $\langle m, n \rangle \in R_4$ if m divides n . Therefore R_4 is reflexive on \mathbb{N} , because every positive integer divides itself. R_4 is not symmetric: For example, 1 divides 2 but 2 does not divide 1. However, R_4 is anti-symmetric: If m divides n and n divides m , then we must have $m = n$. As R_4 is reflexive, it cannot be asymmetric. Finally, R_4 is transitive: If m divides n and n divides o , we must have natural numbers a and b such that

$$n = a \cdot m \quad \text{and} \quad o = b \cdot n.$$

This implies $o = b \cdot a \cdot m$ and therefore m divides o .

Sometimes, it is necessary to convert a relation R that is not transitive into a transitive relation. In order to do so, we define the powers R^n for all $n \in \mathbb{N}$. This definition is given by induction on n .

1. Base Case: $n = 0$. Define

$$R^0 := \text{id}_M$$

2. Induction Step: $n \rightarrow n + 1$.

According to the inductive hypothesis, R^n is already defined. We define R^{n+1} as

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

Later, we need the following law of powers: For any $k, l \in \mathbb{N}$ we have

$$R^k \circ R^l = R^{k+l}.$$

Proof: We proof this law by induction on k .

B.C.: $k = 0$. We have

$$R^0 \circ R^l = \text{id}_M \circ R^l = R^l = R^{0+l}.$$

I.S.: $k \mapsto k + 1$. We have

$$\begin{aligned} R^{k+1} \circ R^l &= (R \circ R^k) \circ R^l && \text{definition of } R^{k+1} \\ &= R \circ (R^k \circ R^l) && \text{associativity of } \circ \\ &= R \circ R^{k+l} && \text{induction hypothesis} \\ &= R^{(k+l)+1} && \text{definition } R^{(k+l)+1} \\ &= R^{(k+1)+l}. && \square \end{aligned}$$

We define the *transitive closure* of a binary relation R on a set M as

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

Here, the expression $\bigcup_{i=1}^{\infty} R^n$ is to be interpreted as

$$\bigcup_{i=1}^{\infty} R^n = R^1 \cup R^2 \cup R^3 \cup \dots$$

Proposition 10 Assume M is a set and $R \subseteq M \times M$ is a binary relation on M . Then we have the following:

1. R^+ is transitive.

2. With respect to the inclusion ordering \subseteq , the relation R^+ is the smallest relation T on M that is both transitive and extends R . To put it differently: If T is a transitive relation on M such that $R \subseteq T$, then we must have $R^+ \subseteq T$, as R^+ is the smallest relation with these properties.

Proof:

1. Let us first prove that R^+ is transitive. We have to prove

$$\forall x, y, z : (\langle x, y \rangle \in R^+ \wedge \langle y, z \rangle \in R^+ \rightarrow \langle x, z \rangle \in R^+).$$

Therefore, assume $\langle x, y \rangle \in R^+$ and $\langle y, z \rangle \in R^+$. We have to show that this implies $\langle x, z \rangle \in R^+$. According to the definition of R^+ we have

$$\langle x, y \rangle \in \bigcup_{n=1}^{\infty} R^n \quad \text{and} \quad \langle y, z \rangle \in \bigcup_{n=1}^{\infty} R^n.$$

According to the definition of $\bigcup_{n=1}^{\infty} R^n$ there are natural numbers $k, l \in \mathbb{N}$ such that we have

$$\langle x, y \rangle \in R^k \quad \text{and} \quad \langle y, z \rangle \in R^l.$$

Using the definition of the composition $R^k \circ R^l$ we conclude

$$\langle x, z \rangle \in R^k \circ R^l.$$

Using the law of powers we have

$$R^k \circ R^l = R^{k+l}.$$

Combining these formulæ, we conclude $\langle x, z \rangle \in R^{k+l}$ and this implies

$$\langle x, z \rangle \in \bigcup_{n=1}^{\infty} R^n.$$

Therefore, we have $\langle x, z \rangle \in R^+$ and thus have shown R^+ to be transitive.

2. Next, we prove that R^+ is indeed the smallest relation that is both transitive and contains R . Therefore, assume T is a transitive relation such that $R \subseteq T$. We have to show $R^+ \subseteq T$.

In order to do this, we first prove the following claim by induction on n

$$\forall n \in \mathbb{N} : (n \geq 1 \rightarrow R^n \subseteq T).$$

B.C.: $n = 1$. We have to show $R^1 \subseteq T$. We have

$$R^1 = R \circ \text{id}_M = R.$$

Therefore, $R^1 \subseteq T$ is an immediate consequence of our assumption $R \subseteq T$.

I.S.: $n \mapsto n + 1$. According to the induction hypothesis, we have

$$R^n \subseteq T.$$

We multiply this inclusion with R and arrive at

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

If we multiply both sides of the assumption $R \subseteq T$ with T we get

$$R \circ T \subseteq T \circ T.$$

As T is transitive, we have

$$T \circ T \subseteq T.$$

Taken together, we have the following chain of inclusions

$$R^{n+1} \subseteq R \circ T \subseteq T \circ T \subseteq T.$$

Therefore, we have shown $R^{n+1} \subseteq T$ and the claim is proven.

From this, $R^+ \subseteq T$ is an immediate consequence of the definition of R^+ . \square

Example: Define *Human* as the set of all human beings that have ever lived. We define the *parent* on *Human* as follows:

$$\text{parent} := \{\langle x, y \rangle \in \text{Human}^2 \mid x \text{ is father of } y \text{ or } x \text{ is mother of } y\}.$$

The transitive closure of *parent* is the set of all those pairs $\langle x, y \rangle$ such that x is an ancestor of y :

$$\text{parent}^+ = \{\langle x, y \rangle \in \text{Human}^2 \mid x \text{ is an ancestor of } y\}.$$

Example: Define F as the set of all airports. Let us define a relation D on F as follows:

$$D := \{\langle x, y \rangle \in F \times F \mid \text{there is a direct flight from } x \text{ to } y\}.$$

So D is the set of all direct connections. The relation D^2 is defined as

$$D^2 = \{\langle x, z \rangle \in F \times F \mid \exists y \in F : (\langle x, y \rangle \in D \wedge \langle y, z \rangle \in D)\}.$$

These are all pairs $\langle x, y \rangle$ such that there is a connection from x to y with exactly one stop. Similarly, D^3 is the set of all pairs $\langle x, y \rangle$ such that you can get from x to y with two stops. In general, D^k is the set of those pairs $\langle x, z \rangle$, such that you need $k - 1$ stops to go from x to y . The transitive closure D^+ contains all those pairs of airports $\langle x, y \rangle$ such that there is a connection from x to y that can contain any number of stops in between.

Exercise 2: Let us define the relation R on the set \mathbb{N} as follows:

$$R = \{\langle k, k + 1 \rangle \mid k \in \mathbb{N}\}.$$

Compute the following relations:

1. R^2 ,
2. R^3 ,
3. R^n for any $n \in \mathbb{N}$ such that $n \geq 1$,
4. R^+ .

Definition 11 The relation $R \subseteq M \times M$ is an *equivalence relation* on M iff

1. R is reflexive on M ,
2. R is symmetric and
3. R is transitive.

The notion of an equivalence relation is a generalization of the notion of equality, as the identical relation id_M is a trivial example of an equivalence relation on M . To present a non-trivial example, given a positive natural number n , we define the relation

$$\approx_n := \{ \langle x, y \rangle \in \mathbb{Z}^2 \mid \exists k \in \mathbb{Z} : k \cdot n = x - y \}$$

We have $x \approx_n y$ iff x and y yield the same remainder when divided by n . We show that for $n \neq 0$ the relation \approx_n is an equivalence relation on \mathbb{Z} .

1. In order to prove that \approx_n is reflexive we have to show that

$$\langle x, x \rangle \in \approx_n \quad \text{holds for all } x \in \mathbb{Z}.$$

According to the definition of \approx_n we therefore have to show

$$\langle x, x \rangle \in \{ \langle x, y \rangle \in \mathbb{Z}^2 \mid \exists k \in \mathbb{Z} : k \cdot n = x - y \} \quad \text{for all } x \in \mathbb{Z}.$$

This is equivalent to

$$\exists k \in \mathbb{Z} : k \cdot n = x - x.$$

Obviously, choosing $k = 0$ satisfies this equation.

2. Next, we prove that \approx_n is symmetric. Assume $\langle x, y \rangle \in \approx_n$. Then there is a $k \in \mathbb{Z}$ such that

$$k \cdot n = x - y.$$

Therefore, we also have

$$(-k) \cdot n = y - x.$$

Hence $\langle y, x \rangle \in \approx_n$.

3. In order to prove \approx transitive we assume that both $\langle x, y \rangle \in \approx_n$ and $\langle y, z \rangle \in \approx_n$ holds. Then there are $k_1, k_2 \in \mathbb{Z}$ such that

$$k_1 \cdot n = x - y \quad \text{and} \quad k_2 \cdot n = y - z.$$

Adding these equation we see

$$(k_1 + k_2) \cdot n = x - z.$$

Defining $k_3 := k_1 + k_2$ we have $k_3 \cdot n = x - z$ and that shows $\langle x, z \rangle \in \approx_n$. Therefore, the relation \approx_n is transitive. \square

Proposition 12 Assume M and N are sets and

$$f : M \rightarrow N$$

is a function mapping M to N . If we define the relation $R_f \subseteq M \times M$ as

$$R_f := \{ \langle x, y \rangle \in M \times M \mid f(x) = f(y) \},$$

then R_f is an equivalence relation. We call R_f the equivalence relation *induced* by f .

Proof: We have to prove that R_f is reflexive on M , symmetric, and transitive.

1. We have

$$\forall x \in M : f(x) = f(x).$$

Therefore

$$\forall x \in M : \langle x, x \rangle \in R_f,$$

showing that R_f is reflexive.

2. In order to establish R_f as symmetric, we have to prove

$$\forall x, y \in M : (\langle x, y \rangle \in R_f \rightarrow \langle y, x \rangle \in R_f).$$

Let us therefore assume $\langle x, y \rangle \in R_f$. According to the definition of R_f we conclude

$$f(x) = f(y).$$

From this

$$f(y) = f(x)$$

is immediate and according to the definition of R_f this implies

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

3. To establish R_f as transitive, we have to prove

$$\forall x, y, z \in M : (\langle x, y \rangle \in R_f \wedge \langle y, z \rangle \in R_f \rightarrow \langle x, z \rangle \in R_f).$$

Therefore, assume

$$\langle x, y \rangle \in R_f \wedge \langle y, z \rangle \in R_f.$$

According to the definition of R_f we have

$$f(x) = f(y) \wedge f(y) = f(z).$$

But then

$$f(x) = f(z).$$

This is the same as

$$\langle x, z \rangle \in R_f. \quad \square$$

Example: Take H as the set of all human beings and take N as the set of all nations. To simplify the discussion, let us assume that every human being has exactly one citizenship. Then we can define a function

$$cs : H \rightarrow N$$

such that for every human x , $cs(x)$ is the citizenship of x . In the equivalence relation R_{CS} induced by the function cs , all those human beings are equivalent that have the same citizenship.

Definition 13 (Equivalence Class) If R is an equivalence relation on M , we define the set $[x]_R$ for all $x \in M$ as follows:

$$[x]_R := \{y \in M \mid x R y\} \quad (\text{Here, } x R y \text{ is used as abbreviation for } \langle x, y \rangle \in R.)$$

The set $[x]_R$ is called the *equivalence class generated by x* .

Proposition 14 If $R \subseteq M \times M$ is an equivalence relation, then we have:

1. $\forall x \in M : x \in [x]_R$
2. $\forall x, y \in M : (x R y \rightarrow [x]_R = [y]_R)$
3. $\forall x, y \in M : (\neg x R y \rightarrow [x]_R \cap [y]_R = \emptyset)$

Remark: For any $x, y \in M$ we either have $x R y$ or $\neg(x R y)$. Therefore, the equivalence classes generated by two elements x and y are either identical or they share no elements:

$$\forall x, y \in M : ([x]_R = [y]_R \vee [x]_R \cap [y]_R = \emptyset).$$

Proof of proposition 14:

1. We have

$$x \in [x]_R \leftrightarrow x \in \{y \in M \mid x R y\} \leftrightarrow x R x.$$

Now $x R x$ is always true since R is an equivalence relation and therefore has to be reflexive. This shows

$$\forall x \in M: x \in [x]_R.$$

2. Assume $x R y$. In order to prove $[x]_R = [y]_R$ we have to show both $[x]_R \subseteq [y]_R$ and $[y]_R \subseteq [x]_R$.

We start with the proof of $[x]_R \subseteq [y]_R$. Assume $u \in [x]_R$. According to the definition of $[x]_R$ this implies $x R u$. From $x R y$, since R is symmetric, we have $y R x$. Now since R is transitive, from $y R x$ and $x R u$ we conclude $y R u$ gilt. According to the definition of $[y]_R$ we therefore have $u \in [y]_R$. This shows $[x]_R \subseteq [y]_R$.

We show $[y]_R \subseteq [x]_R$ next and assume $u \in [y]_R$. Then we have $y R u$. From $x R y$ and $y R u$ we conclude $x R u$, as R is transitive. This implies $u \in [x]_R$ and therefore we have proven $[y]_R \subseteq [x]_R$.

3. Assume $\neg(x R y)$. In order to prove $[x]_R \cap [y]_R = \emptyset$ we assume that there exists a z such that $z \in [x]_R \cap [y]_R$. We will show that this assumption must lead to a contradiction.

So we start with $z \in [x]_R$ and $z \in [y]_R$. According to the definition of $[x]_R$ and $[y]_R$ we must have

$$x R z \quad \text{and} \quad y R z.$$

Since R is symmetric we can turn $y R z$ around and have

$$x R z \quad \text{and} \quad z R y.$$

As R is transitive we conclude $x R y$, contradiction the assumption $\neg(x R y)$. Therefore, there can be no z such that $z \in [x]_R \cap [y]_R$ holds. As a consequence, the set $[x]_R \cap [y]_R$ must be empty. \square

Definition 15 (Partition) Let $\mathcal{P} \subseteq 2^M$ be a set of subsets of M . We call \mathcal{P} a *partition* of M iff the following is true:

1. $\forall x \in M : \exists K \in \mathcal{P} : x \in K$, (completeness property)
for every element from M there is a set in \mathcal{P} containing this element.
2. $\forall K, L \in \mathcal{P} : (K \cap L = \emptyset \vee K = L)$, (separation property)
two sets from \mathcal{P} are either equal or they do not share any elements.

Remark: The last proposition shows that for every equivalence relation R on a set M , the set

$$\{[x]_R \mid x \in M\}$$

of equivalence classes generated by R is a partition of M . Next, we will show that this can be turned around: If we have a partition \mathcal{P} on M , then this partition gives rise to an equivalence relation on M .

Proposition 16 Assume M is a set and $\mathcal{P} \subseteq 2^M$ is a partition of M . Define the relation R as follows:

$$R := \{ \langle x, y \rangle \in M \times M \mid \exists K \in \mathcal{P} : (x \in K \wedge y \in K) \}.$$

Then R is an equivalence relation on M .

Proof: We have to show that R is reflexive on M , symmetric and transitive.

1. In order to show that R is reflexive on M we have to show

$$\forall x \in M : x R x.$$

According to the definition of R , this is the same as

$$\forall x \in M : \exists K \in \mathcal{P} : (x \in K \wedge x \in K)$$

However, the latter is an immediate consequence of the completeness property of \mathcal{P}

$$\forall x \in M : \exists K \in \mathcal{P} : x \in K.$$

2. In order to show that R is symmetric we have to show

$$\forall x, y \in M : (x R y \rightarrow y R x).$$

Let us assume that

$$x R y$$

holds. From the definition of R this is the same as

$$\exists K \in \mathcal{P} : (x \in K \wedge y \in K).$$

Now this formula can be rewritten to

$$\exists K \in \mathcal{P} : (y \in K \wedge x \in K)$$

and then the definition of R shows

$$y R x.$$

3. In order to show that R is transitive we have to prove

$$\forall x, y, z \in M : (x R y \wedge y R z \rightarrow x R z).$$

Let us assume

$$x R y \wedge y R z.$$

According to the definition of R this is the same as

$$\exists K \in \mathcal{P} : (x \in K \wedge y \in K) \wedge \exists L \in \mathcal{P} : (y \in L \wedge z \in L).$$

But then there are two sets $K, L \in \mathcal{P}$ such that

$$x \in K \wedge y \in K \cap L \wedge z \in L.$$

Therefore $K \cap L \neq \emptyset$ and the separation property of \mathcal{P} shows that

$$K = L.$$

Then we have

$$\exists K \in \mathcal{P} : (x \in K \wedge z \in K)$$

and by the definition of R this means

$$x R z.$$

□

Partial Order, Linear Order A relation $R \subseteq M \times M$ is a *partial order* (in the sense of \leq) on M iff R is

1. reflexive on M ,
2. anti-symmetric, and
3. transitive.

The relation R is a *linear order* on M if it is a partial order and, furthermore, we have

$$\forall x \in M : \forall y \in M : (x R y \vee y R x).$$

The obvious example of a linear order is the relation \leq on integers. However, as we will soon see, there are many more interesting examples.

Example: The divisibility relation **div** on natural numbers is defined as follows:

$$\mathbf{div} := \{ \langle x, y \rangle \in \mathbb{N} \times \mathbb{N} \mid \exists k \in \mathbb{N} : k \cdot x = y \}.$$

We prove that **div** is a partial order on \mathbb{N} in the sense of \leq . To this end, we prove that **div** is reflexive on \mathbb{N} , symmetric, and

1. **div** is reflexive on \mathbb{N} : We have to prove

$$\forall x \in \mathbb{N} : x \mathbf{div} x.$$

According to the definition of **div** this is the same as

$$\forall x \in \mathbb{N} : \exists k \in \mathbb{N} : k \cdot x = x.$$

Choosing $k = 1$ we have $k \cdot x = x$ for all $x \in \mathbb{N}$ and therefore **div** is reflexive on \mathbb{N} .

2. **div** is symmetric: We have to prove

$$\forall x, y \in \mathbb{N} : (x \mathbf{div} y \wedge y \mathbf{div} x \rightarrow x = y)$$

Therefore, assume

$$x \mathbf{div} y \wedge y \mathbf{div} x.$$

We have to show $x = y$. According to the definition of **div**, our assumption is equivalent to

$$(\exists k_1 \in \mathbb{N} : k_1 \cdot x = y) \wedge (\exists k_2 \in \mathbb{N} : k_2 \cdot y = x)$$

Therefore we have natural numbers k_1 and k_2 such that

$$k_1 \cdot x = y \wedge k_2 \cdot y = x.$$

Substituting these equations into each other we arrive at

$$k_1 \cdot k_2 \cdot y = y \quad \text{and} \quad k_2 \cdot k_1 \cdot x = x.$$

But then we must have

$$k_1 \cdot k_2 = 1 \vee (x = 0 \wedge y = 0).$$

From $k_1 \cdot k_2 = 1$ we immediately conclude $k_1 = 1$ and $k_2 = 1$, as both k_1 and k_2 are natural numbers. Then $x = y$ is immediate. If we have both $x = 0$ and $y = 0$, we again have $x = y$. So in any case, $x = y$ and this had to be shown.

3. **div** is transitive: We have to show

$$\forall x, y, z \in \mathbb{N} : (x \text{ div } y \wedge y \text{ div } z \rightarrow x \text{ div } z)$$

Let us assume

$$x \text{ div } y \wedge y \text{ div } z.$$

We have to show $x \text{ div } z$. According to the definition of **div** this means

$$(\exists k_1 \in \mathbb{N} : k_1 \cdot x = y) \wedge (\exists k_2 \in \mathbb{N} : k_2 \cdot y = z).$$

Then there are natural numbers k_1 and k_2 such that

$$k_1 \cdot x = y \wedge k_2 \cdot y = z.$$

Substituting the first equation into the second we get

$$k_2 \cdot k_1 \cdot x = z.$$

Setting $k_3 := k_2 \cdot k_1$ we have $k_3 \cdot x = z$ and therefore we conclude

$$x \text{ div } z.$$

The relation **div** is not a linear order on \mathbb{N} : For example, we neither have $2 \text{ div } 3$ nor $3 \text{ div } 2$. \diamond

Exercise 3: Define the relation \leq as follows:

$$\leq := \{ \langle x, y \rangle \in \mathbb{N} \times \mathbb{N} \mid \exists k \in \mathbb{N} : x + k = y \}.$$

Prove that \leq is a linear order on \mathbb{N} . \diamond

Exercise 4: The inclusions relation \subseteq is defined on the power set $2^{\mathbb{N}}$ as follows:

$$\subseteq := \{ \langle A, B \rangle \in 2^{\mathbb{N}} \times 2^{\mathbb{N}} \mid \exists C \in 2^{\mathbb{N}} : A \cup C = B \}$$

Prove that \subseteq is a partial order on $2^{\mathbb{N}}$ and show that it is not a linear order. \diamond

Our excursion into set theory ends here. More details can be found in the literature. A good starting point is the book “Set Theory and Related Topics” by Seymour Lipshutz [Lip98]. This book is also notable for its large number of exercises, most of them with solutions.

Chapter 2

SETL2

The concepts from set theory introduced in the last chapter are quite abstract and therefore are difficult to grasp at the beginning. In order to facilitate the assimilation of these concepts we introduce the programming language SETL. The name is short for set language. As SETL is based on set theory, working with SETL increases the understanding of set theoretical concepts. Furthermore, programming in SETL is quite rewarding, as many problems, whose solution would take long and complicated programs in programming languages like C or Java, have short and elegant solutions when programmed in SETL.

Due to time constraints, I won't be able to discuss every feature of SETL. The reader interested in the features not covered in this lesson is referred to the literature for further information. The article [DH90] provides an introduction, while [Sny90] covers the complete language. You can find these articles on the following web page:

<http://wwwlehre.dhbw-stuttgart.de/~stroetma/setl2.html>

This web page also provides a guide to install SETL2, which is the variant of SETL2 that we will use in this lecture.

2.1 Introduction to Setl2

This section demonstrates the basic features of SETL using simple examples. We start with the program `hello.stl` shown in figure 2.1 on page 34. The line numbers in this figure are not part of the program. I have included them in order to be able to refer to individual lines. The program `hello.stl` in figure 2.1 prints a greeting message onto the screen.

```
1  -- My first SETL program.
2  program main;
3      print("Hello world!");
4  end main;
```

Figure 2.1: A simple SETL program.

If the program from figure 2.1 is stored in a file with the name `hello.stl`, then in order to compile and execute this program, we need to execute the following commands.

1. `stll -c setl2.lib`

This commands creates a library in the current directory. In the beginning, this library is empty. The programs that we are going to develop will later be added to this library. As

long as all our program files are put in the same directory, we get by with just one library. Therefore, this command has to be executed only once.

2. `stlc hello.stl`

This command compiles the SETL program from the file “`hello.stl`” and adds the compiled version of it to the library created in the first step.

3. `stlx main`

This command executes the program with the name “`main`” from the library. As a result, the program should print the following message:

Hello world!

Let us discuss the details of the program shown in figure 2.1 on page 34.

1. The first line is a comment. In *Setl*, comments are started with the the double dash “--”. Everything between “--” and the end of the line is considered a comment and ignored by the compiler.

2. The second line is the *program declaration*. This declaration defines the *name* of the program. In SETL all programs are stored in one library. Therefore, we have to assign a name to every program in order to be able to look up the program in the library later.

The program declaration starts with the keyword “`program`”, followed by the name of the program. In general, a program name can be any string consisting of letters, digits and the underscore character “_” that begins with a letter. However, we will be lazy and just call all our programs “`main`”. Note that SETL is not case sensitive: Therefore, the names “`main`”, “`Main`”, or even “`MAIN`” all refer to the same name. This is also true for variable names: Both “`x`” and “`X`” denote the same variable! However, the programs presented in this lecture will never make use of case insensitivity.

The declaration in the second line is terminated with a semicolon “;”. In SETL, all declarations and statements are terminated with a semicolon.

3. The third line has the only statement in this example. It calls the predefined procedure “`print`”. This procedure is called with an arbitrary number of arguments, which have to be separated by commas “,”. A procedure call of the form

`print(a_1, \dots, a_n)`

prints the arguments a_1, \dots, a_n one by one without separation. After that, a newline is printed.

The second line also demonstrates the first data type of SETL. This is the data type *string*. In SETL, strings literals are enclosed in double quotes. *Setl* supports five more data types. In this lecture, we will discuss four of them, namely numbers, sets, lists, and boolean values.

4. The last line terminates the program. There are two things important to note here:
 - (a) After the keyword “`end`” we specify the name of the program ending here. This name has to be the same as the name that is used in the declaration following the keyword “`program`”. If we call all our programs “`main`”, then we will always close the program with “`end main`”. Note however, that specifying the program name is optional, so instead of “`end main`” we could have just used the shorter “`end`” to terminate the program.
 - (b) The line has to end with a semicolon “;”. This might be surprising to a student only used to programming in C, as in C it is not required to have a semicolon after the closing brace of a function. However, in SETL this semicolon is required.

To give a slightly more interesting example, consider the program `sum.stl` shown in figure 2.2. This program reads a number n from the keyboard, calls the procedure $sum(n)$ and finally prints the result computed by this procedure call. We discuss this program line by line.

```

1  program main;
2      read(n);          -- Doesn't work on Windows!
3      total := sum(n);
4      print("Sum of 0 + 1 + 2 + ... + ", n, " = ", total);
5
6      -- Compute the sum  $\sum_{i=0}^n i$ .
7      procedure sum(n);
8          if n = 0 then
9              return 0;
10         else
11             return sum(n-1) + n;
12         end if;
13     end sum;
14 end main;
```

Figure 2.2: Computing the sum $\sum_{i=0}^n i$.

1. In line 2 we read the number n from the keyboard. Note that the variable n has not been declared. Contrary to the programming language C, SETL is dynamically typed. Therefore, variables need not be declared. Also, during the execution of a program, a variable can take values of different types. We could initialize a variable x with a string and later proceed to store a number into x . Although that would not be an advisable programming style, it is possible.

Unfortunately, the interactive SETL programming environment for the operating system *Windows* does not support interactive reading from a terminal. Therefore `read()` only works in *Linux* and *Mac OS* environments.

2. Line 3 calls the procedure `sum()` with n as argument. The result is assigned to the variable `total`. The procedure `sum` is defined below starting in line 7. Observe that it is not necessary to declare this procedure. This is different to programming in C: There, a function can only be used if it has either been defined or at least has been declared.

Another difference to the programming language C is the form of the assignment operator. In SETL, the assignment operator is “:=”, while in C the assignment operator is “=”.

3. We find the definition of the procedure `sum()` in lines 7 to 13. This definition is started with the keyword “**procedure**” followed by the name of the procedure. In general, after the name of a procedure the arguments follow. They are enclosed in parenthesis and separated by a comma. The definition of the procedure is ended in line 13 by the keyword “**end**” followed by the name of the procedure. Again, this name is optional.
4. Lines 8 to 12 make up the body of the procedure. In this case, the body contains only a single statement. This statement is a conditional statement. In SETL, the conditional statement is given as follows:

```

1      if  $test_0$  then
2           $body_0$ 
3      elseif  $test_1$  then
4           $body_1$ 
5      elseif  $test_2$  then
6           $body_2$ 
7           $\vdots$ 
8      elseif  $test_n$  then
9           $body_n$ 
10     else
11          $body_{n+1}$ 
12     end if;

```

All keywords have been underlined. The conditional statement is executed as follows.

- (a) First, the formula $test_0$ is evaluated to either “true” or “false”.
Note that, in contrast to the programming language C, testing the equality of two expressions is done using the operator “=”.
- (b) If the evaluation of $test_0$ yields “true”, the statements in $body_0$ are executed. After that, the execution of the conditional statement terminates.
- (c) Otherwise, the formula $test_1$ is evaluated. If this yields “true”, the statements in $body_1$ are executed. After that, the execution of the conditional statement terminates.
- (d) \vdots
- (e) If all formulae $test_0, test_1, \dots, test_n$ are evaluated as “false”, then the statements in $body_{n+1}$ are executed.

The procedure $sum(n)$ is *recursive*, that is the procedure calls itself. The logic governing this recursion is best described using the following conditional equations:

1. $sum(0) = 0$,
2. $n > 0 \rightarrow sum(n) = sum(n - 1) + n$.

In order to establish the validity of these equations, we substitute the definition

$$sum(n) = \sum_{i=0}^n i$$

for $sum(n)$ in these equations. Then, the equations are transformed into the following:

1. $\sum_{i=0}^0 i = 0$,
2. $\sum_{i=0}^n i = \left(\sum_{i=0}^{n-1} i \right) + n$.

These equations are obviously true.

The general form of a SETL-program is given in figure 2.3 on page 38. Here $body_0$ is the sequence of the statements that is executed when the program is run. These statements can call the procedures that are defined below. The procedure calls have a “*call by value*” evaluation strategy: If a procedure f is called with a variable x and the value of x is changed in the procedure, then the caller of x doesn’t notice this change. In SETL, it is possible to declare a variable as “**rw**” (read write). In that case, changes made to the variable inside a procedure are visible outside also. However, I will not make any use of this feature in this lecture.

```

1  program prog-name;
2      body0;
3
4      procedure function-name1(args1);
5          body1
6      end function-name1;
7          ⋮
8      procedure function-namem(argsm);
9          bodym
10     end function-namem
11 end prog-name;

```

Figure 2.3: General form of a SETL program.

2.2 The Data Type Set

In this section, we will show how to work with sets. We start with a simple program that exemplifies how to compute the union, the intersection, and the difference of two sets. Furthermore, we show how to compute the power set. The program `simple.stl` is shown in figure 2.4.

```

1  program main;
2      a := { 1, 2, 3 };
3      b := { 2, 3, 4 };
4      -- computing the union
5      c := a + b;
6      print(a, " + ", b, " = ", c);
7      -- computing the intersection
8      c := a * b;
9      print(a, " * ", b, " = ", c);
10     -- computing the set difference
11     c := a - b;
12     print(a, " - ", b, " = ", c);
13     -- computing the power set
14     c := pow a;
15     print("pow ", a, " = ", c);
16     -- testing the inclusion relation
17     print("(", a, " <= ", b, ") = ", (a <= b));
18 end main;

```

Figure 2.4: Computing union, intersection, set difference and power set.

Let us discuss this program line by line.

1. Line 2 and 3 show how to define a set as an explicit enumeration.
2. Line 5 computes the union using the operator “+”.
3. Line 8 computes the intersection using the operator “*”.
4. Line 11 computes the set difference using the operator “-”.

5. Line 14 computes the power set using the prefix operator “**pow**”.

Observe that “**pow**” is an operator and not a function. Therefore, we can write “**pow a**” instead of “**pow(a)**”.

6. Line 17 tests whether a is a subset of b .

If we execute this program, we will get the following:

```
{1, 2, 3} + {4, 2, 3} = {4, 1, 2, 3}
{1, 2, 3} * {4, 2, 3} = {2, 3}
{1, 2, 3} - {4, 2, 3} = {1}
pow {1, 2, 3} = {{1, 2, 3}, {}, {2, 3}, {1}, {1, 3}, {2}, {3}, {1, 2}}
({1, 2, 3} <= {4, 2, 3}) = FALSE
```

Inspecting the output we find that the order, in which the elements in a set appear seems to be quite unpredictable. This is due to the fact that this order is irrelevant in a set.

Next, we show some more ways to construct sets.

Arithmetic Enumerations

Defining a set by explicitly enumerating all its elements is only practical if the set is small. Therefore, in SETL, sets can also be defined as *arithmetic enumerations*. For example, the following definition works:

```
a := { 1 .. 100 };
```

This would define **a** as the set of all natural numbers from 1 up to 100. The general form of an arithmetic enumeration is

```
a := { start .. stop };
```

Using this definition, the set **a** would be defined as the set of all integers from *start* to *stop*, i.e. we would have

$$a = \{n \in \mathbb{Z} \mid start \leq n \wedge n \leq stop\}.$$

There is a variation of arithmetic enumerations that we will introduce by an example:

```
a := { 1, 3 .. 100 };
```

This statement defines **a** as the set of all odd natural less than 100. The general form of this definition is

```
a := { start, second .. stop }
```

If we define

$$step = second - start$$

and if $step > 0$, then this set is given as follows:

$$a = \{start + n * step \mid n \in \mathbb{N} \wedge start + n * step \leq stop\}.$$

Iterators

The most powerful way to define a set in SETL is via the use of an *iterator*. Iterators offer an easy way to construct image sets. We begin with an example:

```
p := { n * m : n in {2..10}, m in {2..10} };
```

After this assignment, **p** is the set of all those products that are build from integer factors ≤ 10 . Using the notation from the first chapter of this lecture notes, we therefore have

$$p = \{n * m \mid n \in \mathbb{N} \wedge m \in \mathbb{N} \wedge 2 \leq n \wedge 2 \leq m \wedge n \leq 10 \wedge m \leq 10\}.$$

The power of iterators is best demonstrated with the program `primes-sieve.stl` shown in figure 2.5 on page 40. This program computes the set of all prime numbers up to some given number n . It is as short as it is impressive. The idea is that a number is prime if it can not be written as a non-trivial product. Therefore, in order to compute the prime numbers $\leq n$ we take the set of all natural numbers ≥ 2 and $\leq n$ and remove those numbers from this set that can be written as non-trivial products.

```

1  program main;
2      n := 100;
3      primes := {2..n} - { p * q : p in {2..n}, q in {2..n} };
4      print(primes);
5  end main;
```

Figure 2.5: Computing the primes up to a given number n .

In general, a set can be defined using iterators as follows:

$$\{expr : x_1 \text{ in } S_1, \dots, x_n \text{ in } S_n\} \quad (1)$$

Here, $expr$ is an expression containing the variables x_1, \dots, x_n , while S_1, \dots, S_n are sets. The expressions $x_i \text{ in } S_i$ are called *iterators*, since the variable x_i iterates over all values from the corresponding set S_i . The mathematical interpretation of the set **a** is then given as follows:

$$\{expr \mid x_1 \in S_1 \wedge \dots \wedge x_n \in S_n\} \quad (2)$$

Be careful to distinguish between the SETL syntax used in (1) and the mathematical notation used in (2). In SETL, the operator “|” is written as “:” and instead of “ \wedge ” we use “,”.

It is possible to use the selection principle in SETL. In order to do this, we have to attach a *condition* to an iterator. The general syntax is as follows:

$$\{expr : x_1 \text{ in } S_1, \dots, x_n \text{ in } S_n \mid cond\} \quad (3)$$

Here, $expr$, x_i , and S_i have the same meaning as above, while $cond$ is a boolean expression presumable containing the variables x_1, \dots, x_n that is evaluated either as **true** or **false**. The mathematical interpretation is given as

$$\{expr \mid x_1 \in S_1 \wedge \dots \wedge x_n \in S_n \wedge cond\}. \quad (4)$$

Let us present an example:

$$primes := \left\{ p : p \text{ in } \{2..100\} \mid \{ x : x \text{ in } \{1..p\} \mid p \bmod x = 0 \} = \{1, p\} \right\};$$

After this assignment, **primes** is the set of all prime numbers ≤ 100 . The idea is, that a number p greater than 1 is prime iff it only has the trivial divisors 1 and p . In order to check whether a number p is divisible by a number x we can use the operator **mod**: The expression

$$p \bmod x$$

computes the remainder when p is divided by x . The corresponding operator in the programming language C is written as “%”. The number p is divisible by x iff $p \bmod x = 0$. Therefore,

$$\{ x : x \text{ in } \{1..p\} \mid p \bmod x = 0 \}$$

is the set of all numbers that divide p and p is prime if this set is identical to $\{1, p\}$. The program `primes-slim.stl` shown in figure 2.6 on page 41 uses this idea to compute prime numbers.

In line 3 we have used an abbreviation available in SETL: A set definition of the form

$$\{x : x \text{ in } S \mid cond\}$$

```

1  program main;
2      n := 100;
3      primes := { p in {2..n} | { x in {1..p} | p mod x = 0 } = {1, p} };
4      print( primes );
5  end main;

```

Figure 2.6: Another way to compute the primes.

can be abbreviated as

$\{x \text{ in } S \mid \text{cond}\}$.

2.3 Pairs, Relations, and Functions

In SETL, the pair $\langle x, y \rangle$ is written as $[x, y]$, the angle brackets “ \langle ” and “ \rangle ” are replaced with the square brackets “[” and “]”. In the last chapter, we have seen that a set of pairs is a relation and that the notion of a relation is a generalization of the notion of a function. Furthermore, if the relation $R \subseteq M \times M$ is left-total on M and right-unique, then R represents a function $f_R : M \rightarrow N$. In this case, SETL permits us to use the relation as a function: If R is a function on M and $x \in M$, then $R(x)$ denotes the unique element y such that $\langle x, y \rangle \in R$. The program `function.stl` shown in figure 2.7 on page 41 shows how relations can be used as functions in SETL. Furthermore, we see that $\text{dom}(R)$ is written as `domain()` and $\text{rng}(R)$ is written as `range(R)` in SETL.

```

1  program main;
2      Q := { [n, n*n] : n in {1..10} };
3      print( "Q(3)      = ", Q(3)      );
4      print( "dom(Q) = ", domain(Q) );
5      print( "rng(Q) = ", range(Q)  );
6  end main;

```

Figure 2.7: Binary relations as functions.

The program computes the binary relation that represents the function $x \mapsto x * x$ on the set $\{n \in \mathbb{N} \mid 1 \leq n \wedge n \leq 10\}$. In line 3, the relation is evaluated at $x = 3$. Finally, $\text{dom}(Q)$ and $\text{rng}(Q)$ are computed. We get the following result:

```

Q(3)      = 9
domain(Q) = {4, 8, 1, 5, 9, 2, 6, 10, 3, 7}
range(Q)  = {4, 16, 36, 64, 100, 1, 9, 25, 49, 81}

```

What happens if R is a binary relation and we try to evaluate the expression $R(x)$ but the set $\{y \mid \langle x, y \rangle \in R\}$ is either empty or contains more than one element? The program `buggy-function.stl` shown in figure 2.8 on page 42 computes the answer to this question.

If the set $\{y \mid \langle x, y \rangle \in R\}$ is either empty or has more than one element, then the expression $R(x)$ is undefined. In mathematics, an undefined value is sometimes denoted as Ω . In SETL, the undefined value is printed as “<om>”. If we try to add the undefined value to a set M , then M is not changed. Therefore, line 5 of the program just prints the empty set, as both $R(1)$ and $R(2)$ are undefined.

We can use the notation $R\{x\}$ instead of $R(x)$ to avoid undefined values. For a binary relation R and an object x , the expression $R\{x\}$ is defined as follows:

```

1  program main;
2      R := { [1, 1], [1, 4], [3, 3] };
3      print( "R(1) = ", R(1) );
4      print( "R(2) = ", R(2) );
5      print( "{ R(1), R(2) } = ", { R(1), R(2) } );
6      print( "R{1} = ", R{1} );
7      print( "R{2} = ", R{2} );
8  end main;

```

Figure 2.8: Non-functional binary relations.

$$R\{x\} := \{y \mid \langle x, y \rangle \in R\}.$$

Therefore, the program from figure 2.8 yields the following results:

```

R(1) = <om>
R(2) = <om>
{ R(1), R(2) } = {}
R{1} = {4, 1}
R{2} = {}

```

2.4 Sequences

SETL supports sequences of arbitrary length. These sequences are known as lists in SETL. The methods to define lists are similar to the methods to define sets: We just have to replace the curly braces “{” and “}” with the square brackets “[” and “]”. Then we can define list using the following methods:

1. explicit enumerations, as in

```
x := [ 1, 2, 3 ];
```

2. arithmetic enumerations, as in

```
x := [ 1 .. 100 ];
```

3. iterations, as in

```
x := [ i * i : i in [ 1 .. 100 ] ];
```

4. iterations combined with selections, as in

```
x := [ i * i : i in [ 1 .. 100 ] | i mod 3 = 0 ];
```

The program `primes-tuple.stl` in figure 2.9 computes the set of prime numbers using the same algorithm as the program in figure 2.6 on page 41. However, it has the advantage that the resulting list of primes is sorted.

2.5 Set Operators

The program `sort.stl` given in figure 2.10 on page 44 gives a simple algorithm to turn a set into a list that is sorted. It uses some properties of SETL that have not been discussed so far. SETL provides quite a few binary operators. Besides the arithmetic operators “+”, “-”, “*”, “/”, “mod”, “**” that compute the sum, the difference, the product, the quotient, the remainder, and the power, SETL provides the operators “min” and “max”. These compute the minimum and the

```

1  program main;
2      n := 100;
3      primes := [ p in [2 .. n] | divisorSet(p) = {1, p} ];
4      print(primes);
5
6      procedure divisorSet(p);
7          return { t in {1..p} | p mod t = 0 };
8      end divisorSet;
9  end main;

```

Figure 2.9: Computing the primes using lists.

maximum of two numbers. Furthermore, for every of these operators there is a corresponding set operator that is build by appending the character “/” to the name of the original operator. This new operator can be used either as a unary prefix operator that is applied to a set, or can be used as a binary operator that takes two arguments: The left hand side argument is a number and the right hand side argument is a set of numbers. How this works out is best demonstrated with an example. The expression

$$0 + / \{1, 2, 3\}$$

is evaluated as

$$0 + 1 + 2 + 3.$$

In general, if op is a binary operator then when evaluating the expression

$$x \text{ op} / S$$

the binary operator “ op ” is put between x and all elements from S . If S is empty, the expression is evaluated as x . Therefore, in an expression of the form $x \text{ op} / S$, we will choose x as the identity element of the operator op . For example, if op is “+”, we choose x as 0, but when op is “*”, we take x as 1. If the operator op doesn’t satisfy the laws of commutativity and associativity, it doesn’t make much sense to use the operator $op /$, since the evaluation of the expression

$$x \text{ op} / S$$

will then depend on the order of the elements of S . As this order is not predictable, the result of “ $x \text{ op} / S$ ” would be quite arbitrary.

If we use $op /$ as a unary set operator as in

$$op / S$$

the result is undefined if S is empty. If S is not empty, the result is computed by putting the binary operator op between all elements of S .

Let us explain the program given in figure 2.10 on page 44. The expression

$$0 \text{ max} / \text{sort}(S)$$

computes the biggest element in the set S , provided this element is not negative. Therefore, the variable n in

$$n \text{ in } [1 \dots 0 \text{ max} / S]$$

iterates from 1 up to the biggest number in S . Due to the condition “ $n \text{ in } S$ ”, the number n is inserted into the resulting list iff n is an element of S . Since the iterator

$$n \text{ in } [1 \dots 0 \text{ max} / S]$$

produces the numbers from smallest to biggest, the result of line 6 is the sorted list of all those

```

1  program main;
2      S := { 13, 5, 7, 2, 4 };
3      print( "sort( ", S, " ) = ", sort(S) );
4
5      procedure sort(S);
6          return [ n in [1 .. 0 max/ S] | n in S ];
7      end sort;
8  end main;

```

Figure 2.10: Sorting a set.

natural numbers that are elements of S . This algorithm for sorting a set will only work if the set merely contains natural numbers. Obviously, the algorithm is far from being efficient.

Figure 2.11 shows an elegant evaluation of the sum $\sum_{i=0}^n i$ using the operator “+/>”.

```

1  program main;
2      n      := 36;
3      total := sum(n);
4      print("sum(", n, ") = ", total);
5
6      procedure sum(n);
7          return 0 +/ {1 .. n};
8      end sum;
9  end main;

```

Figure 2.11: Computing the sum $\sum_{i=0}^n i$.

There are some binary operator where the notation

$x \text{ op/ } S$

isn’t very usefull: If the operator op doesn’t have an identity element, then $op/$ should be used as unary operator. For example, in order to compute the smallest element of a set of integers, we would use the operator “min/” as follows

$m := \text{min/ } S;$

The reason is, that the identity element of the operator “min” is ∞ (infinity) and this value can not be represented in SETL.

2.6 Special Operators on Set and Lists

There are some more operators, that can only be used on lists and sets. The first of these operators is the binary operator “from”. This operator is used as follows:

$x \text{ from } S;$

Here, S is a set and x is a variable. After this statement is executed, an arbitrary element from S is removed from S and x is set to the value of this element. If S is empty, then x is undefined (“<om>”) and S remains unchanged. The program `from.stl` in figure 2.12 on page 45 uses the operator `from` to print a set of elements such that every element is printed on a separate line.

```

1  program main;
2      S := { 13, 5, 7, 2, 4 };
3      printSet(S);
4
5      -- print the elements of S on separate lines.
6      procedure printSet(S);
7          if S = {} then
8              return;
9          end if;
10         x from S;
11         print(x);
12         printSet(S);
13     end printSet;
14 end main;

```

Figure 2.12: Printing the elements of a set.

The unary operator “arb” picks an arbitrary element from a given set S , while leaving the set S unchanged. After

```

S := { 1, 2 };
x := arb S;
print("x = ", x);
print("S = ", S);

```

x would either be 1 or 2, while in any case S is $\{1, 2\}$.

Similar to the operator “from” on sets, there are the operators “fromb” and “frome” that work on lists. The operator “fromb” removes the first element from a list, while “frome” removes the last element.

The operator “+” can be used on lists. It appends its second argument to the first argument. After

```

L1 := [ 1, 2 ];
L2 := [ 2, 3 ];
L3 := L1 + L2;

```

the list L3 is given as

[1, 2, 2, 3],

while the lists L1 and L2 are unchanged.

The unary operator “#” can be applied both to sets and to lists. It yields the number of elements. Therefore, after

```

S1 := { 1, 2 };
S2 := { 2, 3 };
S3 := S1 + S2;
x := S3;

```

the variable x would be set to 3 as the set $S3 = \{1, 2, 3\}$.

We can extract the elements of a list l using the notation

$$x := l(n);$$

If the list l contains at least n elements, then x will be assigned the value of the n th element. This can also be turned around. The statement

```
l(n) := x;
```

sets the *n*th element of *l* to *x*. In contrast to the programming language C, the numbering of elements in a list starts with 1. Therefore, after the statements

```
L := [ 1, 2, 3 ];   x := L(1);
```

x will have the value 1.

Finally, lists can appear on the left hand side of an assignment. The statement

```
[ x, y ] := [ y, x ]
```

swaps the values of *x* and *y*.

```

1  program main;
2      a := [ 1, 2, 3 ];
3      b := [ 2, 3, 4, 5, 6 ];
4      c := { 5, 6, 7 };
5      -- appending lists with +
6      print(a, " + ", b, " = ", a + b);
7      -- computing the number of elements
8      print("# ", c, " = ", # c);
9      -- computing the length of a list
10     print("# ", a, " = ", # a);
11     -- removing the first element
12     x fromb a;
13     print("x = ", x); print("a = ", a);
14     -- removing the last element
15     x frome b;
16     print("x = ", x); print("b = ", b);
17     -- selecting the 3rd element from b
18     print(b, "(3) = ", b(3) );
19     -- changing the 3rd element in b
20     b(3) := 42;
21     print("b = ", b);
22     x := 1; y := 2;
23     -- swapping the values of x and y
24     [ x, y ] := [ y, x ];
25     print("x = ", x, ", y = ", y);
26 end main;
```

Figure 2.13: More Operators on lists and sets.

The program `simple-tuple.stl` in figure 2.13 on page 46 shows the operators discussed above in action. The output is as follows:

```

[1, 2, 3] + [2, 3, 4, 5, 6] = [1, 2, 3, 2, 3, 4, 5, 6]
# {5, 6, 7} = 3
# [1, 2, 3] = 3
x = 1
a = [2, 3]
x = 6
b = [2, 3, 4, 5]
[2, 3, 4, 5](3) = 4
b = [2, 3, 42, 5]
x = 2, y = 1
```

2.6.1 Case Study: *Selection Sort*

```
1  procedure minSort(L);
2      if L = [] then
3          return [];
4      end if;
5      m := min/ L;
6      return [ m ] + minSort( [ x in L | x /= m ] );
7  end minSort;
```

Figure 2.14: Selection sort algorithm.

As an application of the operators discussed so far we show how to implement a sorting algorithm. This algorithm is known as *selection sort*. In order to sort a given list L , it works as follows:

1. If L is empty, $\text{sort}(L)$ is the empty list:

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

2. Otherwise, we compute the minimum m of L :

$$m = \min(L).$$

Next, m is removed from L and the resulting list is sorted recursively:

$$\text{sort}(L) = [\min(L)] + \text{sort}([x \in L \mid x \neq \min(L)]).$$

Figure 2.14 on page 47 shows the procedure `min-sort.stl` that implements these ideas.

2.7 Boolean Expressions

SETL provides all means to manipulate the flow of control that are known from other programming languages like **C**. Normally, the program flow is controlled with the help of boolean expressions. These expressions can be build using the following binary operators:

“=”, “/=", “>”, “<”, “>=”, “<=”, and “in”,

Note that the operator “/=" is used to check whether two values are different. In contrast, the programming language **C** uses the operator “!=" for that purpose. For numbers, the operators “>”, “<”, “>=” and “<=” have the same semantics as in the programming language **C**. However, these operators can also be used to compare sets. Then, they compare the corresponding sets for the appropriate inclusion relation, e. g. the expression

$$S1 < S2$$

yields true iff $S1$ is a subset of $S2$ and, furthermore, $S1$ is different from $S2$.

The operator “in” checks whether its first argument is an element of the second argument. Therefore, when S is a set,

$$x \text{ in } S$$

is **true** iff $x \in S$ gilt. However, “in” can also be used when the second argument is a list, so when L is a list

$$x \text{ in } L$$

checks whether the element x occurs in the list L .

Using the operators build described above, we can define simple tests. To implement more complex logical tests we can use the boolean operators “**and**”, “**or**”, and “**not**”. Note that “**and**” and “**or**” both have the same precedence. Therefore, if t_1 , t_2 , and t_3 are tests, then

“ t_1 **and** t_2 **or** t_3 ”

is a syntax error. We have to write either

“(t_1 **and** t_2) **or** t_3 ” or “ t_1 **and** (t_2 **or** t_3)”.

SETL supports the use of quantifiers. The universal quantifier can be used as follows:

forall x in S | $cond$

Here, S is a set (or a list), while $cond$ is a boolean expression. The evaluation of the quantifier yields true iff the evaluation of $cond$ yields true for all elements of S . Program `primes-forall.stl` in figure 2.15 on page 48 shows how the universal quantifier can be used to compute the set of all prime numbers, as the condition

forall x in `factors(p)` | x in $\{1, p\}$

is true for a number p iff the set of all factors of p contains only 1 and p .

```

1  program main;
2      n := 100;
3      -- primes is the set of prime numbers that are less or equal than n.
4      primes := [ p in [2..n] | isPrime(p) ];
5      print( primes );
6
7      procedure isPrime(p);
8          return forall x in factors(p) | x in {1, p};
9      end isPrime;
10
11     procedure factors(p);
12         return { t in {1..p} | p mod t = 0 };
13     end factors;
14 endw@ main;
```

Figure 2.15: Computing the primes using a universal quantifier.

In general, the expression

forall x in S | $cond$

is equivalent to the equality

$\{ x \text{ in } S \mid cond \} = S$.

The existential quantifier is also supported. The syntax is akin to the syntax given for the universal quantifier, it is:

exists x in S | $cond$

Again, S is a set (or a list) and $cond$ is a boolean expression. If there is at least one element x in S such that the evaluation of $cond$ yields **true**, then the existential quantifier yields **true**. Furthermore, in this case the variable x is set to a value such that $cond$ yields true for that value. If the evaluation of the existential quantifier yields **false**, the variable x will be undefined, i. e. it will have the value Ω .

2.8 Flow of Control

In this section we discuss the tools to control the flow of control.

2.8.1 Case Statements

Instead of using **if-then-else** we can alternatively use **case** statements. A case-statement is written as shown in figure 2.16 on page 49. When a case-statement of the form given below is executed, the tests $test_1, \dots, test_n$ are evaluated one by one. If $test_i$ is the first test that yields true, the the statements in $body_i$ are executed next. If all tests $test_1, \dots, test_n$ yields false, the statements in $body_{n+1}$ are executed.

```
1  case
2      when test1 => body1
3      :
4      when testn => bodyn
5      otherwise => bodyn+1
6  end case;
```

Figure 2.16: Structure of a case statement.

Program `case.stl` in figure 2.17 demonstrates a trivial application of a **case**-block. Later, when implementing the algorithm to compute the conjunctive normal form we will see a more realistic application.

```
1  program main;
2      read(n);
3      m := n mod 10;
4      case
5          when m = 0 => print("last digit is 0");
6          when m = 1 => print("last digit is 1");
7          when m = 2 => print("last digit is 2");
8          when m = 3 => print("last digit is 3");
9          when m = 4 => print("last digit is 4");
10         when m = 5 => print("last digit is 5");
11         when m = 6 => print("last digit is 6");
12         when m = 7 => print("last digit is 7");
13         when m = 8 => print("last digit is 8");
14         when m = 9 => print("last digit is 9");
15         otherwise => print("impossible error!");
16     end case;
17 end main;
```

Figure 2.17: A case statement in action.

The programming language **C** has a control structure similar to this **case**-block. However, the keyword corresponding to “**case**” in SETL is “**switch**” and the keyword corresponding to “**when**” is “**case**”. Another difference is the fact, that in SETL at most one of the cases is executed, there is no *fall-through* to the next block as in **C**.

2.8.2 Loops

SETL supports four different kind of loops:

1. **while** loops,
2. **until** loops,
3. **for** loops, and
4. pure loops.

We will discuss three of these loops below. We won't discuss **until** loops, as their syntax is different from the syntax given to these loops in most other languages and is quite counterintuitive. This is demonstrated by the program `until.stl` shown in figure 2.18. Since every **until** loop is executed at least once, execution of the program shown on figure 2.18 results in an infinite loop.

```
1  program main;
2      n := 10;
3      i := 10;
4      until i = n loop
5          print(i);
6          i := i + 1;
7      end loop;
8  end main;
```

Figure 2.18: An infinite loop.

while Loops

The syntax of a **while** loop is show in figure 2.19 on page 50. Here, *test* is a boolean expression. If it is evaluated as “false”, the loop terminates. Otherwise, *body* will be executed. After that, the loop is restarted.

```
while test loop
    body
end loop;
```

Figure 2.19: Syntax of the **while** loop.

The program `primes-while.stl` shown in figure 2.20 on page 51 demonstrates the use of a **while** loop. The program computes the primes. The idea is that a number p is prime iff there is no other prime that is both smaller and a factor of p .

for Loops

The form of a **for** loop is given in figure 2.21 on page 51. Here, S is a set (or a list) and x is a variable. For each element e in S , x is set to e and *body* is executed.

The program `primes-eratosthenes.stl` in figure 2.22 on page 51 shows how to compute the primes using a **for** loop. The algorithm implemented in this program is known as *Sieve of Eratosthenes*. It works as follows: In order to compute all prime number $\leq n$ we build a list of length n . The i th element of this list is set to i . Line 3 in figure 2.22 builds this list. After that, all numbers that are multiples of 2, 3, 4, \dots are removed form the list of prime numbers by setting the corresponding entry to 0. In order to do this, we need two nested loops. The outer

```

1  program main;
2      n := 100;
3      primes := {};
4      p := 2;
5      while p <= n loop
6          if forall t in primes | p mod t /= 0 then
7              print(p);
8              primes := primes + { p };
9          end if;
10         p := p + 1;
11     end loop;
12 end main;

```

Figure 2.20: Computing the primes with a loop.

```

for x in S loop
    body
end loop;

```

Figure 2.21: Structure of a **for** loop

for loop iterates over all number from 2 up to n . The inner **while** loop iterates for a given i over all products $i \cdot j$ which satisfy $i \cdot j \leq n$ and sets the corresponding entry to 0. Finally, the **for** loop in line 14 up to line 18 prints all those numbers i such that $\text{primes}(i)$ has not been set to 0, as these are the prime numbers.

```

1  program main;
2      n := 15;
3      primes := [1 .. n];
4      print(primes);
5      for i in [2 .. n] loop
6          j := 2;
7          while i * j <= n loop
8              primes(i * j) := 0;
9              j := j + 1;
10         end loop;
11     end loop;
12     for i in [2 .. n] loop
13         if primes(i) > 0 then
14             print(i);
15         end if;
16     end loop;
17 end main;

```

Figure 2.22: Computing the primes according to Eratosthenes.

The algorithm from figure 2.22 can be improved by the following observations:

1. In line 6, it is sufficient to initialize j as i as all smaller multiples have been set to 0 before.
2. If i in line 5 is not a prime number, then there is no need for the inner while loop, as all multiples of i are also multiples of the factors of i and therefore have already been set to 0 before.

Program `primes-eratosthenes-fast.stl` in figure 2.23 on page 52 shows the resulting algorithm. In order to skip the iteration of the inner `while` loop when `primes(i) = 0`, we have used the `continue` statement. Calling `continue` aborts the current iteration and sets the looping variable `i` to the next value. The statement `continue` has the same behavior as it has in the programming language `C`.

```

1  program main;
2      n := 10000;
3      primes := [1 .. n];
4      for i in [2 .. n] loop
5          if primes(i) = 0 then
6              continue;
7          end if;
8          j := i;
9          while i * j <= n loop
10             primes(i * j) := 0;
11             j := j + 1;
12         end loop;
13     end loop;
14     for i in [2 .. n] loop
15         if primes(i) > 0 then
16             print(i);
17         end if;
18     end loop;
19 end main;

```

Figure 2.23: More efficient computation of prime numbers.

Unrestricted Loops

The form of an unrestricted loop is shown in figure 2.24 on page 52. This loop will only terminate when an either an `exit` statement or a `return` statement is encountered. A loop of this kind is useful if the termination condition can not be tested at the start of the loop.

```

loop
    body
end loop;

```

Figure 2.24: Pure loops.

Let us inspect an example: Suppose we want to solve the equation

$$x = \cos(x)$$

on the set \mathbb{R} of real numbers. A naive approach that works is based on the observation that if we define the sequence $(x_n)_n$ inductively as

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

then it can be shown that the limit

$$l := \lim_{n \rightarrow \infty} x_n$$

is a solution for the equation given above. Therefore, the program `solve.stl` shown in figure 2.25 on page 53 computes the solution to the equation $x = \cos(x)$.

```

1  program main;
2      x := 1.0;
3      loop
4          old_x := x;
5          x := cos(x);
6          if abs(x - old_x) < 1.0e-13 then
7              print("x = ", x);
8              exit;
9          end if;
10     end loop;
11 end main;

```

Figure 2.25: Solving the equation $x = \cos(x)$ via iteration.

In this program, the loop terminates once the values of `x` and `old_x` are close enough to each other. This test can not be done at the start of the loop, since the variable `old_x` is still undefined. Therefore, we have to postpone the test. The program also demonstrates the `exit` command. This command terminates the loop. In the programming language C there is a command with a similar semantics: It is called `break`.

As a side note, this example demonstrates that SETL supports floating point numbers. Any number containing a decimal point “.” is recognized as a floating point number. SETL provides the following functions on floating point numbers:

1. The trigonometric functions `sin()`, `cos()` and `tan()`.
2. The inverse trigonometric functions `asin()`, `acos()` and `atan()`. Furthermore, `atan2(y, x)` computes the arctangent of $\frac{y}{x}$.
3. The exponential function `exp()` and the natural logarithm `log()`.
4. The function `abs()` computes the absolute value of its argument.
5. The function `fix()` drops the fractional part of a number.
6. The function `sqrt()` computes the square root.

2.8.3 Case Study: Computation of Probabilities in Poker

We want to show how to compute probabilities in *Texas Hold'em Poker*. In this variant of poker, the deck has 52 cards. Every card has a *suit*, these suits are element of the set

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

Furthermore, every card has a value and this value is an element of the set

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

Therefore, we can represent a card as a pair. The complete deck of cards is then given as follows:

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

Every player gets two cards to begin with. This is the so called *preflop*, sometimes also known as *Hole*. After a bidding phase, three more cards are dealt. In contrast to the preflop, these cards are not given to a player. Instead they are put on the table so that everybody can see them. These three cards are known as the *flop*.

Let us assume that a player has been dealt the cards $\{\langle 3, \clubsuit \rangle, \langle 3, \spadesuit \rangle\}$. He wants to know the probability that the flop contains another 3 because then he would held *trips*, which would be quite good for him. In order to compute this probability we have to compute the number of all possible flops and the number of all those flops containing a 3. Dividing the latter number by the former yields the probability sought after. The program `poker-triple.stl` shown in 2.26 performs this computation.

```

1  program main;
2      Suits  := { "c", "h", "d", "s" };
3      Values := { "2", "3", "4", "5", "6", "7", "8", "9",
4                  "T", "J", "Q", "K", "A" };
5      Deck   := { [ v, s ] : v in Values, s in Suits };
6      Hole    := { [ "3", "c" ], [ "3", "s" ] };
7      Rest    := Deck - Hole;
8      Flops   := { { k1, k2, k3 } : k1 in Rest, k2 in Rest, k3 in Rest
9                                | #{ k1, k2, k3 } = 3 };
10     Trips    := { f in Flops | [ "3", "d" ] in f or [ "3", "h" ] in f };
11     print(1.0 * #Trips / #Flops);
12 end main;

```

Figure 2.26: Computing the probability of trips in Texas Hold'em.

1. Line 5 computes the set of all cards.
2. Line 6 defines the set *Hole* as the set containing the cards of the player.
3. The remaining cards are computed in line 7.
4. Line 8 and 9 compute the set of all possible flops. The condition
 $\# \{ k1, k2, k3 \} = 3$
is needed to ensure that the flop consists of 3 different cards.
5. The subset of all those flops containing at least another 3 is computed in line 10.
6. Finally, line 11 computes the probability.

We have to solve one problem here. The variables `#Trips` and `#Flops` are integers. If we would compute `#Trips / #Flops`, then SETL would do an integer division. The result would be 0. Therefore, we have to convert one of the number to a floating point number. This is achieved by multiplying with the floating point number 1.0.

When executing the program we see that the probability to improve a pocket pair to trips is 11,8%.

Remark: The algorithm for computing probabilities shown above is only viable if the sets involved are small enough to be represented explicitly. If this assumption is not true, then we can use the *Monte Carlo Method* to compute the probabilities. This method will be discussed in the second term.

2.9 Case Study: Graph Search

Next we study the problem of finding a path in a *graph*. Here, a graph is considered as a set of locations together with the information about the immediate connections between locations. Therefore, we can represent a *graph* as a binary relation R . We have $\langle x, y \rangle \in R$ iff there is an immediate connection from the location x to the location y . For example, is R is defined as

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

then our locations would be numbered from 1 to 5 and the corresponding connections are shown in figure 2.27. In this graph, the connections are interpreted as one-way roads: For example, there is an immediate connection from location 1 to location 2, but there is no immediate connection from location 2 to location 1.

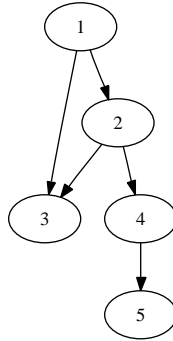


Figure 2.27: A simple graph.

The graph in figure 2.27 does only show the immediate connections between two locations. Of course, there are other connections which are, however, not immediate connections. For example, there is a path from location 1 to location 4 that visits the location 2. We will represent this path as the list

$$[1, 2, 4].$$

In general, a path from x to z in the graph given by the relation R is a list of the form

$$[y_1, \dots, y_n]$$

such that we have the following:

1. $y_1 = x$,
2. $y_n = z$, and
3. $\forall i \in \{1, \dots, n-1\} : \langle y_i, y_{i+1} \rangle \in R$.

Our goal is to develop a program that receives as input a binary relation R representing a graph and two locations x and y . The job of this program is to check whether there is a connection from x to y . If a connection between x and y exists, the program should also compute the corresponding path.

2.9.1 Computing the Transitive Closure

To begin with, we observe that there is a connection from location x to location y iff

$$\langle x, y \rangle \in R^+,$$

where R^+ is the transitive closure of R . In the last chapter, we have shown that the transitive

closure of R can be defined as:

$$R^+ = \bigcup_{i=1}^{\infty} R^i = R \cup R^2 \cup R^3 \cup \dots$$

At first glance this looks as if the computation of R^+ requires an infinite computation. But let us try to dissect the infinite union of sets in the definition of R^+ : First, there is the set R . These are all immediate connections. Then we have R^2 , which is the same as $R \circ R$. Now this is defined as

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

Therefore, R^2 contains all those pairs $\langle x, z \rangle$ such that there is a path from x to z with one stopover. In general, we can show by induction on n that R^n contains all those pairs $\langle x, z \rangle$ such that there are $n - 1$ stopovers. Now the set of locations is finite. So there is some fixed number $k \in \mathbb{N}$ such that there are k different locations. But if there are only k different locations, when traveling from x to z there is no reason to consider more than $k - 2$ stopovers, as otherwise we would visit a location twice. Therefore, the formula for computing R^+ can be shortened to

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

provided there are only k different locations.

We could use the formula given above. However, it is more efficient to use a *fix-point algorithm*. In order to do so, we verify that the transitive closure R^+ satisfies the following *fix-point equation*:

$$R^+ = R \cup R \circ R^+. \quad (2.1)$$

Let me remind you that we have agreed that the relational composition operator “ \circ ” has a higher precedence than the union operator “ \cup ”. Therefore, the expression $R \cup R \circ R^+$ has to be read as $R \cup (R \circ R^+)$.

The fix-point equation 2.1 can be proven algebraically. We have

$$\begin{aligned} & R \cup R \circ R^+ \\ &= R \cup R \circ \bigcup_{i=1}^{\infty} R^i \\ &= R \cup R \circ (R^1 \cup R^2 \cup R^3 \cup \dots) \\ &= R \cup (R \circ R^1 \cup R \circ R^2 \cup R \circ R^3 \cup \dots) \quad \text{law of distributivity} \\ &= R \cup (R^2 \cup R^3 \cup R^4 \cup \dots) \quad \text{definition of } R^n \\ &= \bigcup_{i=1}^{\infty} R^i \\ &= R^+ \end{aligned}$$

We will use the equation 2.1 in order to compute the transitive closure R^+ iteratively. In order to do so we define a sequence $(T_n)_{n \in \mathbb{N}}$ by induction on n :

$$\begin{aligned} \text{I.A. } n = 1: & \quad T_1 := R \\ \text{I.S. } n \mapsto n + 1: & \quad T_{n+1} := R \cup R \circ T_n. \end{aligned}$$

The relations T_n can be written in terms of the relation R :

1. $T_1 = R$.
2. $T_2 = R \cup R \circ T_1 = R \cup R \circ R = R^1 \cup R^2$.
3. $T_3 = R \cup R \circ T_2$
 $= R \cup R \circ (R^1 \cup R^2)$
 $= R^1 \cup R^2 \cup R^3$.

In general, we can prove by induction that

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

holds. The base case is immediate from the definition of T_1 . The induction step works as follows:

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

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

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

The fact that the sequence $(T_n)_{n \in \mathbb{N}}$ is monotonically increasing is an immediate consequence of

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

as we have the following:

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

If the relation R is finite, of course R^+ is finite, too. All sets T_n are subsets of T^+ , because we have

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

Therefore, the relations T_n can not get arbitrarily big. As the sequence $(T_n)_{n \in \mathbb{N}}$ is monotonically increasing, there must be an index $k \in \mathbb{N}$ such that all relations T_n are the same when $n \geq k$:

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

As we have $T_n = \bigcup_{i=1}^n R^i$, we conclude that

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

This implies

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

The algorithm for computing the sequence $(T_n)_{n \in \mathbb{N}}$ is now as follows: We start with $T_1 := R$ and then compute T_{n+1} using

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

This iteration is done until we have $T_{n+1} = T_n$, since then $T_n = R^+$.

The program `transitive.stl` in figure 2.28 on page 58 shows an implementation of the algorithm described above. When executing this program, we get the following result:

```

1  program main;
2
3      R := { [1,2], [2,3], [1,3], [2,4], [4,5] };
4      print( "R = ", R );
5      print( "computing transitive closure of R" );
6      T := closure(R);
7      print( "R+ = ", T );
8
9      -- The procedure call closure(R) computes the transitive closure
10     -- of the binary relation R.
11     procedure closure(R);
12         T := R;
13         loop
14             Old_T := T;
15             T := R + product(R, T);
16             if T = Old_T then
17                 return T;
18             end if;
19         end loop;
20     end closure;
21
22     -- The procedure call product(R1, R2) computes the relational
23     -- product R1 o R2.
24     procedure product(R1, R2);
25         return { [x,z] : [x,y1] in R1, [y2,z] in R2 | y1 = y2 };
26     end product;
27
28 end main;

```

Figure 2.28: Computing the transitive closure.

$R = \{[2, 3], [4, 5], [1, 3], [2, 4], [1, 2]\}$
 $R^+ = \{[1, 5], [2, 3], [4, 5], [1, 4], [1, 3], [2, 4], [1, 2], [2, 5]\}$

The transitive closure R^+ of the relation R can now be interpreted as followed: The pair $\langle x, y \rangle$ is a member of R^+ if there is a *path* starting from x leading to y . The procedure *product()* shows the most general form of a set definition using iterators. In general, we can define a set as follows:

$$\{ \text{expr} : [x_1^{(1)}, \dots, x_{n(1)}^{(1)}] \text{ in } s_1, \dots, [x_1^{(k)}, \dots, x_{n(k)}^{(k)}] \text{ in } s_k \mid \text{cond} \}$$

Here, for all $i = 1, \dots, k$ the expression s_i has to be a set of lists such that each of those lists has the length $n(i)$. When this expression is evaluated, the variables $x_1^{(i)}, \dots, x_{n(i)}^{(i)}$ are bound to the values of the corresponding components in the lists that are members of the sets s_i . For example, the evaluation of

```

s1 := { [ 1, 2, 3 ], [ 5, 6, 7 ] };
s2 := { [ "a", "b" ], [ "c", "d" ] };
M := { [ x1, x2, x3, y1, y2 ] : [ x1, x2, x3 ] in s1, [ y1, y2 ] in s2 | true };

```

yields the following result for the set M

$\{ [1, 2, 3, "a", "b"], [5, 6, 7, "c", "d"],$
 $[1, 2, 3, "c", "d"], [5, 6, 7, "a", "b"] \}.$

If we use this general form to define a list we have to obey an important restriction: All the variables occurring in the various lists $x_1^{(i)}, \dots, x_{n(i)}^{(i)}$ have to be distinct, that is, all variables $x_j^{(i)}$ have to be different from each other. Therefore, the procedure *product* given above can not be implemented as follows:

```

1      procedure product(R1, R2);
2          return { [x,z] : [x,y] in R1, [y,z] in R2 };
3      end product;

```

The reason is that the variable *y* occurs twice here. We have solved this problem by using two distinct variables *y1* and *y2*. In order to enforce that the values of these variables are the same we had to add the condition *y1* = *y2*.

2.9.2 Computing the Paths

The program shown in figure 2.28 computes the transitive closure and therefore answers the question, whether there is a connection between two given locations. However, in practice we do not only want to know whether there is a connection between two locations *x* and *y*, we also want to compute the path leading from *x* to *y*. In order to have a convenient description of the algorithm for computing a path, we define the following functions:

1. The function *first(p)* computes the first element of the list *p*:

$$\text{first}([x_1, \dots, x_m]) = x_1.$$

2. The function *last(p)* computes the last element of the list *p*:

$$\text{last}([x_1, \dots, x_m]) = x_m.$$

3. If $p = [x_1, \dots, x_m]$ and $q = [y_1, \dots, y_n]$ are two paths such that $\text{first}(q) = \text{last}(p)$, we define the *sum* of *p* and *q* as

$$p \oplus q := [x_1, \dots, x_m, y_2, \dots, y_n].$$

If P_1 and P_2 are sets of paths, we define the *path product* of P_1 and P_2 as follows:

$$P_1 \bullet P_2 := \{ p_1 \oplus p_2 \mid p_1 \in P_1 \wedge p_2 \in P_2 \wedge \text{last}(p_1) = \text{first}(p_2) \}.$$

Note that the path product generalizes the notion of relational composition: If P_1 and P_2 are relations, i. e. if they contain only lists of length two, then $P_1 \bullet P_2 = P_1 \circ P_2$.

Now we can change the program from figure 2.28 in a way that all possible paths between two locations are computed. The resulting program *path.st1* is shown in figure 2.29. Unfortunately, this program does not work if the graph admits cycles. Figure 2.30 shows a graph containing a cycle. In this graph, there is an infinite number of paths starting at location 1 and leading to location 2:

$$[1, 2], [1, 2, 4, 1, 2], [1, 2, 4, 1, 2, 4, 1, 2], [1, 2, 4, 1, 2, 4, 1, 2, 4, 1, 2], \dots$$

Obviously, we can forget those paths that are cyclic and therefore contain one location multiple times. In order to keep our sets finite, we have to eliminate these paths.

Figure 2.31 shows how the program has to be changed to make it work even for graphs containing cycles.

1. In line 2, we just collect those paths that are not cyclic.

```

1  program main;
2      R := { [1,2], [2,3], [1,3], [2,4], [4,5] };
3      print( "R = ", R );
4      print( "computing all paths" );
5      P := closure(R);
6      print( "P = ", P );
7
8      procedure closure(R);
9          P := R;
10         loop
11             Old_P := P;
12             P := P + path_product(R, P);
13             if P = Old_P then
14                 return P;
15             end if;
16         end loop;
17     end closure;
18
19     procedure path_product(P, Q);
20         return { add(p, q) : p in P, q in Q | p(#p) = q(1) };
21     end path_product;
22
23     procedure add(p, q);
24         return p + q(2..);
25     end add;
26
27 end main;

```

Figure 2.29: Computing all connections.

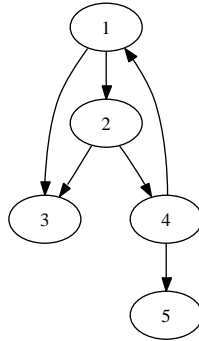


Figure 2.30: A cyclic graph.

2. In line 6 we check, whether a path is cyclic. Now a path is cyclic iff it contains a location multiple times. Therefore, if we transform a cyclic path into a set, the set will contain less elements than the path we started with. The reason is that a set contains every element exactly once.

In general, we are not interested to compute all connections between every possible pair of locations x and y . Instead, we often have a starting location x and a goal location y and we want to compute a path leading from x to y . The program `find-path.stl` shown in figure 2.32 shows the implementation of the procedure `reachable(x, y, R)`. This procedure checks whether there is

```

procedure path_product(P, Q);
  return { add(p,q) : p in P, q in Q | p(#p) = q(1) and not cyclic(add(p,q)) };
end path_product;

procedure cyclic(p);
  return #{ x : x in p } < #p;
end cyclic;

```

Figure 2.31: Computing all connections in a cyclic graph.

a path from x to y in the graph R . Furthermore, a path is computed, provided there is one.

1. Line 3 initializes P to contain the path of length 0 that starts and ends with x .
2. Line 7 selects those paths from P that reach the location y .
3. If the set of those paths is not empty, then we are done. Therefore, in this case, we just pick an arbitrary path leading to y and return it.
4. If we cannot find any more paths, then we leave the procedure in line 12 with a **return** statement. As we do not return any expression, the return value is Ω in this case.

```

1      -- Check whether there is a path from x to y in R and compute it.
2      procedure reachable(x, y, R);
3          P := { [x] };
4          loop
5              Old_P := P;
6              P      := P + path_product(P, R);
7              Found := { p in P | p(#p) = y };
8              if Found /= {} then
9                  return arb Found;
10             end if;
11             if P = Old_P then
12                 return;
13             end if;
14         end loop;
15     end reachable;

```

Figure 2.32: Computing the connections between two points.

2.9.3 Puzzle: Crossing the River

We next show how to solve a puzzle using the theory developed so far.

A farmer needs to bring a wolf, a goat, and a cabbage across the river to reach the market place. The boat is small and the farmer can only transport one passenger at a time. If he leaves the wolf and the goat alone together, the wolf will eat the goat. If he leaves the goat and the cabbage alone together, the goat will eat the cabbage. Of course, neither the wolf nor the goat or the cabbage can steer the boat alone.

Our goal is to develop a schedule that can be used by the farmer to transport all trade goods safely to the market place at the other shore of the river. In order to do so we represent the problem using a binary relation $R \subseteq P \times P$. The elements of the set P will represent the different situations that can occur. We will model these situations as subsets of the set

$$\mathbf{All} := \{\text{farmer, wolf, goat, cabbage}\}.$$

A subset $s \subseteq \mathbf{All}$ is interpreted as the set of those objects that are on the left shore of the river. For example, the set

$$s = \{\text{farmer, goat}\}$$

describes the situation where the farmer and the goat is on the left shore of the river, while the wolf and the cabbage wait on the right shore. Therefore, the set P is just the power set of \mathbf{All} :

$$P := 2^{\mathbf{All}}.$$

In order to define the relation R we define a procedure $\text{problem}(s)$. This procedure gets a set $s \in P$ representing the objects on the left shore of the river after the farmer has crossed over to the right shore. The procedure checks whether there is a problem on the left shore. Now there is a problem if either the goat is together with the cabbage or the wolf is together with the goat. Therefore, this procedure is implemented as follows:

```

procedure problem(S);
  return ("goat" in S and "cabbage" in S) or ("wolf" in S and "goat" in S);
end problem;

```

Using this procedure, we can define a relation R_1 that describes all safe passages from the left shore of the river to the right shore as follows:

$$\begin{aligned}
 R_1 := \{ [S, S - B] : S \text{ in } P, B \text{ in } \text{pow } S \mid \\
 \quad \text{"farmer" in } B \text{ and } \#B \leq 2 \text{ and not problem}(S - B) \\
 \};
 \end{aligned}$$

This relation can be interpreted as follows.

1. S is the set of objects on the left shore. In principle, every subset of \mathbf{All} can be on the left shore of the river. Therefore, we have the iterator " $S \text{ in } P$ ".
2. B is the set of objects crossing over to the right shore using the boat. Of course, for an object to cross from the left shore to the right shore, it has to be on the left shore first. Therefore, B has to be a subset of S . To enforce this, we use the iterator " $B \text{ in } \text{pow } S$ ".
3. $S - B$ is the set of objects that remain on the left shore.
4. Furthermore, we have three conditions concerning the crossing:

- (a) The farmer has to be a passenger of the boat. Therefore, we must have

$$\text{"farmer" in } B.$$

(b) The boat can only contain two passengers. Therefore,

`#B <= 2.`

(c) There must not be a problem on the left shore after the crossing:

`not problem(S - B).`

In a similar way, we can describe the crossings from the right shore to the left shore.

`R2 := { [S, S + B] : S in P, B in pow (All - S) |
"farmer" in B and #B <= 2 and not problem(All - (S + B)) };`

The interpretation of R2 is as follows:

1. Again, S is the set of objects on then left shore. Therefore, we have the iterator “ S in P ”.
2. B is the set of objects crossing over from the right shore to the left shore using the boat. Of course, for an object to cross from the right shore to the left shore, it has to be on the right shore first. Now, if S is the set of objects on the left shore, the set of objects on the right shore is given as $All - S$. Therefore, B has to be a subset of $All - S$. To enforce this, we use the iterator “ B in $pow (All - S)$ ”.
3. $S + B$ is the set of objects that remain on the left shore.
4. Furthermore, we have the following conditions:

(a) The farmer has to be a passenger of the boat. Therefore, we must have

`"farmer" in B.`

(b) The boat can only contain two passengers. Therefore,

`#B <= 2.`

(c) There must not be a problem on the right shore after the crossing. After the crossing, the set of objects on the right shore is given as $All - (S + B)$. Therefore, we need the condition

`not problem(All - (S + B)).`

The set of all possible crossings is the union of both R1 and R2. Finally, we have to model the start state and the goal state we want to reach. In the beginning, everything is on the left shore, so we define

`start := All;`

The goal is to bring everything to the right shore, so that nothing is left on the left shore. Therefore, we define

`goal := {};`

This gives rise to the program shown in figure 2.33. Note that the procedure `reachable` is the same as shown in the previous section. The solution computed by the program is shown in figure 2.34. In order to be more comprehensible, this output has been formatted.

```

1  program main;
2      All := { "farmer", "wolf", "goat", "cabbage" };
3      P   := pow All;
4      R1  := { [ S, S - B ] : S in P, B in pow S |
5              "farmer" in B and #B <= 2 and not problem(S - B) };
6      R2  := { [ S, S + B ] : S in P, B in pow (All - S) |
7              "farmer" in B and #B <= 2 and not problem(All - (S + B)) };
8      R   := R1 + R2;
9
10     start := All;
11     goal  := {};
12
13     path := reachable(start, goal, R);
14     print(path);
15
16     procedure problem(S);
17         return ("goat" in S and "cabbage" in S) or
18                ("wolf" in S and "goat" in S);
19     end problem;
20
21     procedure reachable(x, y, R);
22         P := { [x] };
23         loop
24             Old_P := P;
25             P := P + path_product(P, R);
26             Found := { p in P | p(#p) = y };
27             if Found /= {} then
28                 return arb Found;
29             end if;
30             if P = Old_P then
31                 return;
32             end if;
33         end loop;
34     end reachable;
35
36     procedure path_product(P, Q);
37         return { add(p,q) : p in P, q in Q | p(#p) = q(1) and not cyclic(add(p,q)) };
38     end path_product;
39
40     procedure cyclic(p);
41         return #{ x : x in p } < #p;
42     end cyclic;
43
44     procedure add(p, q);
45         return p + q(2..);
46     end add;
47
48     end main;

```

Figure 2.33: How does the farmer reach the other shore?

```

1  {"goat", "cabbage", "wolf", "farmer"} {}
2      >>>> {"goat", "farmer"} >>>>
3  {"cabbage", "wolf"} {"goat", "farmer"}
4      <<<< {"farmer"} <<<<
5  {"cabbage", "wolf", "farmer"} {"goat"}
6      >>>> {"cabbage", "farmer"} >>>>
7  {"wolf"} {"goat", "cabbage", "farmer"}
8      <<<< {"goat", "farmer"} <<<<
9  {"goat", "wolf", "farmer"} {"cabbage"}
10     >>>> {"wolf", "farmer"} >>>>
11 {"goat"} {"cabbage", "wolf", "farmer"}
12     <<<< {"farmer"} <<<<
13 {"goat", "farmer"} {"cabbage", "wolf"}
14     >>>> {"goat", "farmer"} >>>>
15 {} {"goat", "cabbage", "wolf", "farmer"}

```

Figure 2.34: A schedule for the farmer.

2.9.4 Concluding Remarks

We have discussed only a small set of the features of SETL. In particular we haven't discussed the following topics:

1. string processing,
2. reading and writing of files,
3. interaction with the operation system,
4. object orientation,
5. packages,
6. the interface to the programming language *C*,
7. the interface to *Tk* which enables a graphical user interface.

A detailed discussion of all these aspects is given in the book of Jack Schwartz [Sch03] that can be found at

<http://www.settheory.com>.

Note: Most of the algorithms developed in this chapter are not at all efficient. They have rather been chosen to clarify the notions of set theory. We will discuss more efficient algorithms in the second term.

Chapter 3

Propositional Logic

Propositional logic, which is also known as *propositional calculus*, is the restriction of logic that is concerned with formulæ build up from the following *sentential connectives*:

1. “*and*” (denoted as \wedge),
2. “*or*” (denoted as \vee),
3. “*not*” (denoted as \neg),
4. “*if ... then*” (denoted as \rightarrow), and
5. “*if and only if*” (denoted as \leftrightarrow).

In order to build formulæ using these sentential connectives, we start with a set of so called propositional variables \mathcal{P} . These propositional variables denote atomic propositions that are either true or false. Given a set of propositional variables \mathcal{P} , the set of propositional formulæ \mathcal{F} is defined inductively as follows:

1. \top is a propositional formula. This formula is interpreted as always being true.
2. \perp is a propositional formula. This formula is interpreted as always being false.
3. Every propositional variable $p \in \mathcal{P}$ is a propositional formula.
4. If f is a propositional formula, then

$$\neg f$$

is a propositional formula, too.

5. If f and g are propositional formulæ, then

- (a) $(f \wedge g)$,
- (b) $(f \vee g)$,
- (c) $(f \rightarrow g)$, and
- (d) $(f \leftrightarrow g)$

are propositional formulæ.

The set of propositional formulæ will be denoted as \mathcal{F} .

Example: Assume the set of propositional variables is given as $\mathcal{P} = \{p, q, r\}$. Then the following strings are propositional formulæ:

1. p ,
2. $(p \wedge q)$,
3. $((\neg p \rightarrow q) \vee (q \rightarrow \neg p))$.

□

In order to simplify the notation, we omit parenthesis in the following cases:

1. The outermost parenthesis are always omitted. Therefore, we write

$$p \wedge q \quad \text{instead of} \quad (p \wedge q).$$

2. The connectives “ \vee ” and “ \wedge ” are left associative, therefore we write

$$p \wedge q \wedge r \quad \text{instead of} \quad (p \wedge q) \wedge r.$$

3. The connective “ \rightarrow ” is right-associative, therefore we write

$$p \rightarrow q \rightarrow r \quad \text{instead of} \quad p \rightarrow (q \rightarrow r).$$

4. The connectives “ \vee ” and “ \wedge ” have a higher precedence than “ \rightarrow ”, therefore we write

$$p \wedge q \rightarrow r \quad \text{instead of} \quad (p \wedge q) \rightarrow r.$$

5. The connective “ \rightarrow ” has a higher precedence than “ \leftrightarrow ”, therefore we write

$$p \rightarrow q \leftrightarrow r \quad \text{instead of} \quad (p \rightarrow q) \leftrightarrow r.$$

Remark: Not that the connectives “ \wedge ” and “ \vee ” have the same precedence. Therefore, the string

$$p \wedge q \vee r$$

cannot be regarded as a propositional formula. You will find some books that use a different convention. In some books, the connective “ \vee ” has a higher precedence than “ \wedge ”, while others use the opposite convention so that “ \wedge ” has a higher precedence than “*vee*”. In order to avoid any confusion, both operators have the same precedence in these lecture notes.

3.1 Applications of Propositional Logic

Before diving deeper into the subject, let us motivate our investigations by listing some industrial applications of propositional logic. The following list is by no means complete, I have just listed those applications of propositional logic that I have seen in my own professional career.

1. Analysis and design of digital circuits.

Today, complex digital circuits can easily consist of millions of logical gates¹. A logical gate is a unit that implements one of the logical connectives “*and*”, “*or*”, or “*not*”.

The complexity of modern circuits is unmanageable without the use of computer aided design tools. These tools implement algorithms that are applications of propositional logic.

On such tool that is often used in the verification of circuits is a tool for *circuit comparison*. In circuit comparison, two digital circuits are represented as propositional formulæ. The goal of circuit comparison is to prove the equivalence of these formulæ.

2. Airline crew scheduling.

When creating the time tables for crew personal, an international airline company has to comply with a number of compulsory restrictions regarding the idle time of a crew. In order to be competitive, these idle times have to be minimized in a way that the compulsory restrictions are satisfied. This is an optimization problem that can be expressed as a propositional formulæ.

¹For example, the PentiumTM IV processor that is equipped with the *Northwood* kernel consists of about 55 million logical gates.

3. Railway signaling control.

In a big train station, hundreds of switch points and signals have to be controlled in order to guarantee that trains do not collide. This control can be described using propositional formulæ.

4. There are a lot of puzzles that can be mapped to propositional formulæ. As an example, we will discuss the so called “*eight queens problem*”. This problem asks to put eight queens on a chess board in a way that no queen can attack another queen.

3.2 Semantics of Propositional Formulæ

We have defined the structure of propositional formulæ in the introduction of this chapter, but we haven’t yet defined the *truth* of a formula. In general, without further provisions, we can not speak of the truth of a formula. For example, depending on the truth of p and q , the formula

$$p \vee q$$

might be either true or false. In order to formally define the truth value of a formula, we define the set \mathbb{B} of truth values:

$$\mathbb{B} := \{\mathbf{true}, \mathbf{false}\}.$$

Using \mathbb{B} , we define the notion of a *valuation*.

Definition 17 (Propositional Valuation) A *propositional valuation* is a function

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

assigning a truth value $\mathcal{I}(p) \in \mathbb{B}$ to every propositional variable $p \in \mathcal{P}$. □

A propositional variable is sometimes called an *interpretation* of the propositional variables.

In order to evaluate a propositional formula we need an interpretation of the propositional connectives “ \neg ”, “ \wedge ”, “ \vee ”, “ \rightarrow ”, and “ \leftrightarrow ”. To this end we define the functions \neg , \wedge , \vee , \rightarrow , and \leftrightarrow on the set \mathbb{B} . These functions have the following signatures:

1. $\neg : \mathbb{B} \rightarrow \mathbb{B}$
2. $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
3. $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
4. $\rightarrow : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
5. $\leftrightarrow : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

In set theory, we have seen that functions can be regarded as binary relations. The function \neg swaps the truth values and can therefore be defined as follows:

$$\neg = \{(\mathbf{true}, \mathbf{false}), (\mathbf{false}, \mathbf{true})\}.$$

However, defining the interpretation of the other logic connectives this way is not very intuitive. It is easier to define these interpretations via a table. This table is shown below.

With the help of this table we are now able to define the truth value of a propositional formula with respect to a propositional valuation \mathcal{I} . Let us denote this value as $\widehat{\mathcal{I}}(f)$. The definition of $\widehat{\mathcal{I}}(f)$ is given by induction on f .

1. $\widehat{\mathcal{I}}(\perp) := \mathbf{false}$.
2. $\widehat{\mathcal{I}}(\top) := \mathbf{true}$.

p	q	$\neg(p)$	$\vee(p, q)$	$\wedge(p, q)$	$\oplus(p, q)$	$\ominus(p, q)$
true	true	false	true	true	true	true
true	false	false	true	false	false	false
false	true	true	true	false	true	false
false	false	true	false	false	true	true

Table 3.1: Interpretation of the logical connectives.

3. $\hat{\mathcal{I}}(p) := \mathcal{I}(p)$ for all $p \in \mathcal{P}$.
4. $\hat{\mathcal{I}}(\neg f) := \ominus(\hat{\mathcal{I}}(f))$ for all $f \in \mathcal{F}$.
5. $\hat{\mathcal{I}}(f \wedge g) := \wedge(\hat{\mathcal{I}}(f), \hat{\mathcal{I}}(g))$ for all $f, g \in \mathcal{F}$.
6. $\hat{\mathcal{I}}(f \vee g) := \vee(\hat{\mathcal{I}}(f), \hat{\mathcal{I}}(g))$ for all $f, g \in \mathcal{F}$.
7. $\hat{\mathcal{I}}(f \rightarrow g) := \ominus(\hat{\mathcal{I}}(f), \hat{\mathcal{I}}(g))$ for all $f, g \in \mathcal{F}$.
8. $\hat{\mathcal{I}}(f \leftrightarrow g) := \ominus(\hat{\mathcal{I}}(f), \hat{\mathcal{I}}(g))$ for all $f, g \in \mathcal{F}$.

In order to avoid unnecessary clutter in our notation we will not distinguish between $\hat{\mathcal{I}}$ and \mathcal{I} , that is we will just write $\mathcal{I}(f)$ instead of the more formal $\hat{\mathcal{I}}(f)$.

Example: Let us demonstrate how the truth value of the formula

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

can be calculated for the propositional valuation \mathcal{I} that is defined as

$$\mathcal{I} = \{\langle p, \text{true} \rangle, \langle q, \text{false} \rangle\}.$$

We have:

$$\begin{aligned}
\mathcal{I}\big((p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q\big) &= \ominus\big(\mathcal{I}((p \rightarrow q)), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\big) \\
&= \ominus\big(\ominus(\mathcal{I}(p), \mathcal{I}(q)), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\big) \\
&= \ominus\big(\ominus(\text{true}, \text{false}), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\big) \\
&= \ominus\big(\text{false}, \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\big) \\
&= \text{true}
\end{aligned}$$

Note that in the calculation above I did only evaluate those parts of the formula that needed to be evaluated. Nevertheless, the approach sketched above is too tedious. The easiest way to evaluate a propositional formula is using the table 3.1 on page 69. We show how the formula

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

can be evaluated for arbitrary valuations \mathcal{I} using this table. In order to do this, we construct a table that has a column for every sub-formula occurring in our formula. Table 3.2 on page 70 shows the resulting table.

Looking at the last column of table 3.2 we see that it always contains the value **true**. Therefore, we see that the evaluation of $(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$ yields **true** for every propositional valuation \mathcal{I} . A propositional formula that evaluates as **true** for any evaluation is called a *tautology*.

For complex formulæ the evaluation has to be automated. We will show how to do this later.

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

Table 3.2: Evaluation of $(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$.

3.2.1 Extensional Interpretation of Implication

The interpretation of the propositional connective “ \rightarrow ” is purely *extensional*: In order to compute the truth value of the formula

$$f \rightarrow g$$

we only have to know the truth values $\mathcal{I}(f)$ and $\mathcal{I}(g)$ of the sub-formula f and g , we do not need to know anything about the structure of the formula f and g . This leads to an interpretation of the connective “ \rightarrow ” that is different from the colloquial use of the connective “*if ... then*”. The reason is, that in colloquial use, the connective “*if ... then*” often denotes *causation*. Using the mathematical definition of “ \rightarrow ”, the statement

$$3 \cdot 3 = 8 \rightarrow \text{“it is raining”}$$

is true because the precondition “ $3 \cdot 3 = 8$ ” is obviously false. In colloquial use, the sentence

$$\text{“If } 3 \cdot 3 = 8, \text{ then it is raining.”}$$

is regarded as meaningless. Therefore, you have to keep in mind that the connective “ \rightarrow ” is not exactly the same as “*if ..., then*”. Rather, the connective “ \rightarrow ” has to be regarded as the extensional abstraction of the colloquial “*if ..., then*”.

3.2.2 Implementation in Setl2

In order to improve our understanding of the notions discussed so far we will develop a SETL program that can be used to evaluate a propositional formula. Every time we develop a program we have to decide how to represent the arguments and results. Therefore, we have to decide how to represent a propositional formula in SETL2. In SETL, compound data structures are best represented as lists. Formally, we define the representation of a propositional formula using the function

$$rep : \mathcal{F} \rightarrow \text{SETL}.$$

The formal definition is by induction on f .

1. \top will be represented a the number 1:

$$rep(\top) := 1.$$

2. \perp will be represented a the number 0:

$$rep(\perp) := 0.$$

3. A propositional variable $p \in \mathcal{P}$ is represented as a string giving the name of a variable. As the propositional variables are strings to begin with, we have

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

4. If f is a propositional formula, then $\neg f$ is represented as a list of length 2. The first element of this list is the string “-”, the second element is the representation of f :

$$\text{rep}(\neg f) := ["-", \text{rep}(f)].$$

5. If f_1 and f_2 are propositional formulæ, then we represent $f_1 \vee f_2$ as a list of three elements. The second element of this list will be the string "+", while the first element is the representation of f and the last element is the representation of g :

$$\text{rep}(f \vee g) := [\text{rep}(f), "+", \text{rep}(g)].$$

6. If f_1 and f_2 are propositional formulæ we represent $f_1 \wedge f_2$ as a list of three elements where the second element is the string "*":

$$\text{rep}(f \wedge g) := [\text{rep}(f), "*", \text{rep}(g)].$$

7. In order to represent $f_1 \rightarrow f_2$ we define:

$$\text{rep}(f \rightarrow g) := [\text{rep}(f), "->", \text{rep}(g)].$$

8. In order to represent $f_1 \leftrightarrow f_2$ we define:

$$\text{rep}(f \leftrightarrow g) := [\text{rep}(f), "<->", \text{rep}(g)].$$

The representation given above is by no means unique and we could have used any other representation. In order for a representation to be considered appropriate it should be as intuitive as possible and, furthermore, should provide easy access to the components of a formula.

Next, we discuss how a propositional valuation can be represented in SETL2. Now a propositional valuation is a function

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

mapping the set of propositional variables \mathcal{P} to the set of truth values \mathbb{B} . Therefore, we will represent a propositional interpretation \mathcal{I} as a binary relation that is left-total and right-unique:

$$\text{rep}(\mathcal{I}) := \{ \langle p, \mathcal{I}(p) \rangle : p \in \mathcal{P} \}.$$

Using this representation we can implement a simple procedure that evaluates a given propositional formula f using a given propositional valuation \mathcal{I} . The procedure is shown in figure 3.1 on page 71.

```

1  procedure eval(f, I);
2      case
3          when f = 1      => return TRUE;
4          when f = 0      => return FALSE;
5          when is_string(f) => return I(f);
6          when f(1) = "-"  => return not eval(f(2), I);
7          when f(2) = "*"  => return eval(f(1), I) and eval(f(3), I);
8          when f(2) = "+"  => return eval(f(1), I) or eval(f(3), I);
9          when f(2) = "->" => return not eval(f(1), I) or eval(f(3), I);
10         when f(2) = "<->" => return eval(f(1), I) = eval(f(3), I);
11         otherwise => print("eval: syntax error: ", f);
12     end case;
13 end eval;
```

Figure 3.1: Evaluating a propositional formula.

Let us discuss the details of this implementation:

1. If the argument f is 1, then f represents the formula \top . Therefore, the evaluation of f is always **true**.

2. If the argument f is 0, then f represents the formula \perp . Therefore, the evaluation of f is always **false**.
3. Line 5 deals with the case that the argument f represents a propositional variable. This is the case if f is a string. This can be checked with the help of the library function `is_string()`. In this case, we can apply the propositional valuation \mathcal{I} on the variable f . This works, because \mathcal{I} is represented as a binary relation and in *Setl* a binary relation can be used as a function.
4. Line 6 deals with the case that f has the form $["-", g]$ and therefore represents the formula $\neg g$. In this case, we first evaluate g using \mathcal{I} and then negate the result.
5. Line 7 deals with the case that f has the form $[g_1, "*", g_2]$ and therefore represents the formula $g_1 \wedge g_2$. In this case, we first evaluate g_1 and g_2 recursively. Next, the results are combined with the **and**-operator provided by SETL.
6. Line 8 deals with the case that f has the form $[g_1, "+", g_2]$ and represents the formula $g_1 \vee g_2$. This case is reduced to an application of the **or**-operator of SETL.
7. Line 9 deals with the case that f has the form $[g_1, "->", g_2]$ and represents the formula $g_1 \rightarrow g_2$. In this case we make use of the following equivalence:
$$(p \rightarrow q) \leftrightarrow \neg p \vee q.$$
8. In order to understand line 10 we note that the logical connective " \leftrightarrow " denotes the equality of truth values. Therefore, the evaluation of $g_1 \leftrightarrow g_2$ can be reduced to a comparison of the truth values of g_1 and g_2 .
9. Finally, if we reach line 11 there must be a syntax error.

3.2.3 An Application

We take a look at a hands-on application of the theory discussed so far. Inspector Watson is called to a jewelry that has been robbed. Three suspects, Austin, Brian, and Colin have been arrested. After all facts are evaluated, the following is known:

1. At least one of the three suspects is guilty:

$$f_1 := a \vee b \vee c.$$

2. If Austin is guilty, then he has exactly one accomplice.

In order to represent this fact, we break this statement up into two statements:

- (a) If Austin has done it, then he has at least one accomplice:

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

- (b) If Austin has done it, then he has at most one accomplice:

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

3. If Brian is not guilty, then Colin is not guilty either:

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

4. If exactly two of the suspects are guilty, then Colin is one of them.

It is not easy to see how this statement can be translated into a propositional formula. Let us therefore consider the negation of this formula. Now the statement above is wrong if Colin is not guilty and at the same time Austin and Brian are guilty. Therefore, the statement above can be formalized as follows:

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

5. If Colin is not guilty, then Austin is guilty.

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

We now have a set $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$ of formulae. We are looking for a propositional valuation \mathcal{I} such that all furthermore from f are true when evaluated with \mathcal{I} . If there is exactly one such propositional valuation, then this valuation uniquely determines the culprits. For example, if we had

$$\mathcal{I} = \{\langle a, \text{false} \rangle, \langle b, \text{false} \rangle, \langle c, \text{true} \rangle\},$$

then Colin would be the sole culprit. This propositional valuation doesn't solve the problem because it violates the third fact: As Brian would be innocent, Colin would have to be innocent too. As it would be too time consuming to try all possible evaluations manually we will implement a short program that does the necessary computations. Figure 3.2 shows this program.

Let us discuss the details of this program.

1. Lines 2 – 13 define the formulae f_1, \dots, f_6 . We have to put these formula into their SETL2 representation.
2. Next, we have to decide how we can enumerate all possible propositional valuations. We have already noticed that the propositional valuations correspond to the possible sets of culprits. However, these sets are subsets of the set

$$\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}.$$

Therefore, in line 18 we compute the set of all these subsets.

3. Now we need a way to transform a subset of propositional variables into a propositional valuation. In lines 32 – 34 we have implemented a procedure that takes a subset and converts it into a valuation. In order to understand how this procedure works we examine an example and assume that the subset to be converted is given as

$$M = \{\mathbf{a}, \mathbf{c}\}.$$

The result is

$$\mathcal{I} = \{\langle a, \text{true} \rangle, \langle b, \text{false} \rangle, \langle c, \text{true} \rangle\}.$$

The idea is, that for a propositional variable x the pair $\langle x, \text{true} \rangle$ is a member of the valuation \mathcal{I} if $x \in M$, otherwise we have $\langle x, \text{false} \rangle$ in \mathcal{I} . Therefore, the valuation that yields true for an element x iff x is a member of M is given as follows:

$$\{ [\mathbf{x}, \text{true}] : \mathbf{x} \text{ in } \mathbf{M} \} + \{ [\mathbf{x}, \text{false}] : \mathbf{x} \text{ in } \mathbf{A} \mid \text{not } \mathbf{x} \text{ in } \mathbf{M} \}$$

We can unify these two cases by requiring that the pair $\langle x, x \in M \rangle$ is an element of the valuation \mathcal{I} . This is what line 33 is about.

Observe that we had to supply the set A of all propositional variables as a second argument, for otherwise this variable would have been unknown in the procedure.

4. Line 21 collects the set B of all possible valuations.
5. Line 22 collects the set S those valuations that evaluate every formula from F as true.
6. If there is exactly one valuation that evaluates every formula from f as true, then our problem is solved. In this case, we select an arbitrary valuation (well, its not really arbitrary as there is only one) and transform this valuation into a set.

When we run the program, we get the following result:

Set of culprits: {"b", "c"}

Therefore Brian and Colin are guilty.

```

1  program main;
2      -- f1 := a ∨ b ∨ c
3      f1 := [ [ "a", "+", "b" ], "+", "c" ];
4      -- f2 := a → b ∨ c
5      f2 := [ "a", "->", [ "b", "+", "c" ] ];
6      -- f3 := a → ¬(b ∧ c)
7      f3 := [ "a", "->", [ "-", [ "b", "*", "c" ] ] ];
8      -- f4 := ¬b → ¬c
9      f4 := [ [ "-", "b" ], "->", [ "-", "c" ] ];
10     -- f5 := ¬(¬c ∧ a ∧ b)
11     f5 := [ "-", [ [ "a", "*", "b" ], "*", [ "-", "c" ] ] ];
12     -- f6 := ¬c → a
13     f6 := [ [ "-", "c" ], "->", "a" ];
14
15     FS := { f1, f2, f3, f4, f5, f6 };
16
17     A := { "a", "b", "c" };
18     P := pow A;
19     print("P = ", P);
20     -- B is the set of all propositional valuations.
21     B := { createValuation(M, A) : M in P };
22     S := { I in B | forall f in FS | eval(f, I) };
23     print("Set of all valuations satisfying all facts:", S);
24     if #S = 1 then
25         I := arb S;
26         Culprits := { x in A | I(x) };
27         print("Set of culprits: ", Culprits);
28     end if;
29
30     -- This procedure turns a subset M of A into a propositional
31     -- valuation I, such that I(x) is true iff x is an element of M.
32     procedure createValuation(M, A);
33         return { [ x, x in M ] : x in A };
34     end createValuation;
35
36     procedure eval(f, I);
37         :
38     end eval;
39 end main;

```

Figure 3.2: Who has done it?

3.3 Tautologies

Table 3.2 shows that the formula

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

is true for every propositional valuation. This example motivates the following definition.

Definition 18 If f is a propositional formula and we have

$$\mathcal{I}(f) = \text{true} \quad \text{for every propositional valuation } \mathcal{I},$$

then f is called a *tautology*. In this case we write

$$\models f.$$

□

Examples:

1. $\models p \vee \neg p$
2. $\models p \rightarrow p$
3. $\models p \wedge q \rightarrow p$
4. $\models p \rightarrow p \vee q$
5. $\models (p \rightarrow \perp) \leftrightarrow \neg p$
6. $\models p \wedge q \leftrightarrow q \wedge p$

The fact, that these formulæ are tautologies can be proved by building tables in the same way as done in table 3.2 on page 70. Although this approach is conceptually straightforward, it is inefficient if the number of propositional variables is big. The reason is that the table necessary to establish a formula f containing n variables as a tautology has 2^n rows. One aim of this chapter is to develop an algorithm that does much better in many cases.

The last two examples give rise to the following definition.

Definition 19 (Equivalent) Two formulæ f and g are equivalent iff

$$\models f \leftrightarrow g$$

□

Examples:

$\models \neg \perp \leftrightarrow \top$	$\models \neg \top \leftrightarrow \perp$	
$\models p \vee \neg p \leftrightarrow \top$	$\models p \wedge \neg p \leftrightarrow \perp$	tertium non datur
$\models p \vee \perp \leftrightarrow p$	$\models p \wedge \top \leftrightarrow p$	neutral element
$\models p \vee \top \leftrightarrow \top$	$\models p \wedge \perp \leftrightarrow \perp$	
$\models p \wedge p \leftrightarrow p$	$\models p \vee p \leftrightarrow p$	idempotency
$\models p \wedge q \leftrightarrow q \wedge p$	$\models p \vee q \leftrightarrow q \vee p$	commutativity
$\models (p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$	$\models (p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$	associativity
$\models \neg \neg p \leftrightarrow p$		elimination of $\neg \neg$
$\models p \wedge (p \vee q) \leftrightarrow p$	$\models p \vee (p \wedge q) \leftrightarrow p$	absorption
$\models p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	$\models p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$	distributivity
$\models \neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$	$\models \neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$	DeMorgan rules
$\models (p \rightarrow q) \leftrightarrow \neg p \vee q$		elimination of \rightarrow
$\models (p \leftrightarrow q) \leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p)$		elimination of \leftrightarrow

We can establish these equivalences by constructing tables that test every possible valuation. We demonstrate this approach for the first of the DeMorgan rules.

p	q	$\neg p$	$\neg q$	$p \wedge q$	$\neg(p \wedge q)$	$\neg p \vee \neg q$
true	true	false	false	true	false	false
true	false	false	true	false	true	true
false	true	true	false	false	true	true
false	false	true	true	false	true	true

Table 3.3: Proof of the first DeMorgan law.

Obviously, the last two columns of table 3.3 have identical entries. Therefore the formulæ corresponding to these columns are equivalent.

3.3.1 Checking Tautologies with Setl

In this section we develop a SETL2 program that is able to check whether a given formula is a tautology. The idea is to evaluate a given formula using all possible propositional valuations. In order to do so, we first have to compute the set of all possible propositional valuations. We have already seen that the propositional valuations correspond to sets of propositional formulæ: If M is a set of propositional variables, then we can define the propositional valuation $\mathcal{I}(M)$ as the valuation that evaluates a propositional variable p as **true** iff p is an element of M , that is we define

$$\mathcal{I}(M)(p) := \begin{cases} \text{true} & \text{if } p \in M; \\ \text{false} & \text{if } p \notin M. \end{cases}$$

On the other hand, given a propositional valuation \mathcal{I} , we can define a set $\mathcal{M}(\mathcal{I})$ of propositional variables as

$$\mathcal{M}(\mathcal{I}) := \{p \in \mathcal{P} \mid \mathcal{I}(p) = \text{true}\}.$$

The two mappings $M \mapsto \mathcal{I}(M)$ and $\mathcal{I} \mapsto \mathcal{M}(\mathcal{I})$ are inverses of each other, we have

$$\mathcal{I}(\mathcal{M}(\mathcal{I})) = \mathcal{I} \quad \text{for every propositional valuation } \mathcal{I}$$

and also

$$\mathcal{M}(\mathcal{I}(M)) = M \quad \text{for all } M \subseteq \mathcal{P}.$$

Therefore, we can compute the set of all propositional valuations of a formula f by computing the power set of all propositional variables occurring in f and then transforming the elements of this set into propositional valuations.

The procedure shown in figure 3.3 on page 77 makes use of these considerations and is able to check whether a given formula f is a tautology. This procedure uses the procedure `eval()` that has been shown in figure 3.1 on page 71.

1. Line 2 collects all propositional variables occurring in the formula f . This is done with the help of the procedure `collectVars(f)`, which is shown in lines 12 – 21. This procedure is defined by induction on the propositional formula f .
2. Line 4 computes the set of all propositional valuations. We have seen a similar computation already in the program shown in figure 3.2 on page 74.
3. Line 5 checks whether the given formula does indeed evaluate as **true** for all of the possible valuations. If this is the case, the procedure returns **true**. Otherwise, a counterexample is returned. This enables us to understand why the formula isn't a tautology.

```

1  procedure tautology(f);
2      P := collectVars(f);
3      -- A is the set of all propositional valuations.
4      A := { { [x, x in M] : x in P } : M in pow P };
5      if forall I in A | eval(f, I) then
6          return true;
7      else
8          return arb { I in A | not eval(f, I) };
9      end if;
10 end tautology;
11
12 procedure collectVars(f);
13     case
14         when f = 1          => return {};
15         when f = 0          => return {};
16         when is_string(f) => return { f };
17         when f(1) = "-"    => return collectVars( f(2) );
18         when f(2) in { "*", "+", "->", "<->" }
19             => return collectVars( f(1) ) + collectVars( f(3) );
20         otherwise => print("malformed formula: ", f);
21     end case;
22 end collectVars;

```

Figure 3.3: Testing whether a formula is a tautology.

3.3.2 Conjunctive Normal Form

Instead of using a table to check that a formula is a tautology we can also try to simplify the formula using algebraic manipulations. If we are able to show that a formula f is equivalent to the formula \top , then we have shown that f is a tautology. We demonstrate the idea with the formula

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

by using a chain of equivalences:

$(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$	(elimination of \rightarrow)
$\leftrightarrow (\neg p \vee q) \rightarrow (\neg p \rightarrow q) \rightarrow q$	(elimination of \rightarrow)
$\leftrightarrow (\neg p \vee q) \rightarrow (\neg \neg p \vee q) \rightarrow q$	(elimination of double negation)
$\leftrightarrow (\neg p \vee q) \rightarrow (p \vee q) \rightarrow q$	(elimination of \rightarrow)
$\leftrightarrow \neg(\neg p \vee q) \vee ((p \vee q) \rightarrow q)$	(DeMorgan)
$\leftrightarrow (\neg \neg p \wedge \neg q) \vee ((p \vee q) \rightarrow q)$	(elimination of double negation)
$\leftrightarrow (p \wedge \neg q) \vee ((p \vee q) \rightarrow q)$	(elimination of \rightarrow)
$\leftrightarrow (p \wedge \neg q) \vee (\neg(p \vee q) \vee q)$	(DeMorgan)
$\leftrightarrow (p \wedge \neg q) \vee ((\neg p \wedge \neg q) \vee q)$	(distributivity)
$\leftrightarrow (p \wedge \neg q) \vee ((\neg p \vee q) \wedge (\neg q \vee q))$	(Tertium-non-Datur)
$\leftrightarrow (p \wedge \neg q) \vee ((\neg p \vee q) \wedge \top)$	(neutral element)
$\leftrightarrow (p \wedge \neg q) \vee (\neg p \vee q)$	(distributivity)
$\leftrightarrow (p \vee (\neg p \vee q)) \wedge (\neg q \vee (\neg p \vee q))$	(associativity)
$\leftrightarrow ((p \vee \neg p) \vee q) \wedge (\neg q \vee (\neg p \vee q))$	(Tertium-non-Datur)
$\leftrightarrow (\top \vee q) \wedge (\neg q \vee (\neg p \vee q))$	(neutral element)
$\leftrightarrow \top \wedge (\neg q \vee (\neg p \vee q))$	(neutral element)
$\leftrightarrow \neg q \vee (\neg p \vee q)$	(associativity)
$\leftrightarrow (\neg q \vee \neg p) \vee q$	(commutativity)
$\leftrightarrow (\neg p \vee \neg q) \vee q$	(associativity)
$\leftrightarrow \neg p \vee (\neg q \vee q)$	(Tertium-non-Datur)
$\leftrightarrow \neg p \vee \top$	(neutral element)
$\leftrightarrow \top$	

The algebraic manipulations shown above have a system. In order to explain this system, we need some definitions.

Definition 20 (Literal) A propositional formula f is called a *literal* iff we have either of the following cases:

1. $f = \top$ or $f = \perp$.
2. $f = p$, where p is a propositional variable.
In this case, f is called a *positive literal*.
3. $f = \neg p$, where p is a propositional variable.
In this case, f is called a *negative literal*.

The set of all literals will be denoted as \mathcal{L} . □

Later we need the notion of the *complement* of a literal. If l is a literal, the complement of l is denoted as \overline{l} . It is defined via a case distinction:

1. $\overline{\top} = \perp$ and $\overline{\perp} = \top$.
2. $\overline{p} := \neg p$ if $p \in \mathcal{P}$.
3. $\overline{\neg p} := p$ if $p \in \mathcal{P}$.

Definition 21 (Clause) A propositional formula k is a *clause* iff k has the form

$$k = l_1 \vee \dots \vee l_r$$

where l_i is a literal for all $i = 1, \dots, r$. Therefore, a clause is a disjunction of literals. The set of all clauses is denoted as \mathcal{C} . □

We will regard clauses as sets of literals. When regarding clauses as sets we abstract from the order of the literals. This is possible because of the associativity, commutativity, and idempotency of the connective “ \vee ”. Therefore, the clause $l_1 \vee \dots \vee l_r$ will be written as

$$\{l_1, \dots, l_r\}.$$

The following example illustrates the utility of the set notation of clauses. Consider the clauses

$$p \vee q \vee \neg r \vee p \quad \text{and} \quad \neg r \vee q \vee \neg r \vee p.$$

Both clauses are equivalent, but the respective formulæ are different. If we use the set notation instead, we write these formulæ as

$$\{p, q, \neg r\} \quad \text{and} \quad \{\neg r, q, \neg r, p\}.$$

However, these sets are identical! If we transform the equivalence

$$l_1 \vee \dots \vee l_r \vee \perp \leftrightarrow l_1 \vee \dots \vee l_r$$

into set notation we arrive at

$$\{l_1, \dots, l_r, \perp\} \leftrightarrow \{l_1, \dots, l_r\}.$$

This shows that the element \perp can be safely dropped from a clause. If we set $r = 0$ in the last equivalence, we arrive at the identity

$$\{\perp\} \leftrightarrow \{\}.$$

This shows that the empty set of literals has to be interpreted as \perp .

Definition 22 A clause c is trivial iff we have one of the following cases:

1. $\top \in c$.
2. There exists a $p \in \mathcal{P}$ such that both $p \in c$ and $\neg p \in c$.
In this case, p and $\neg p$ are called complementary literals.

□

Proposition 23 A clause c is a tautology iff c is trivial.

Proof: Let us first assume that c is trivial. If $\top \in c$, then, because of the tautology

$$f \vee \top \leftrightarrow \top,$$

we know that $c \leftrightarrow \top$. If p is a propositional variable such that we have both $p \in c$ and $\neg p \in c$, then the tautology

$$p \vee \neg p \leftrightarrow \top.$$

shows $c \leftrightarrow \top$.

Let us now assume that c is a tautology. In order to show that c is trivial we assume that c is not trivial and we will derive a contradiction from this assumption. If c is not trivial, we have $\top \notin c$ and c can not contain any complementary literals. Therefore, c has the form

$$c = \{\neg p_1, \dots, \neg p_m, q_1, \dots, q_n\} \quad \text{with } p_i \neq q_j \text{ for all } i \in \{1, \dots, m\} \text{ and } j \in \{1, \dots, n\}.$$

Let us define a propositional valuation \mathcal{I} as follows:

1. $\mathcal{I}(p_i) = \text{true}$ for all $i = 1, \dots, m$ and
2. $\mathcal{I}(q_j) = \text{false}$ for all $j = 1, \dots, n$,

Using this valuation we obviously have $\mathcal{I}(c) = \text{false}$. But then c is not a tautology. This contradiction shows that, in order to be a tautology, c has to be trivial. □

Definition 24 (conjunctive normal form) A formula f is in *conjunctive normal form* (abbreviated as cnf) iff f is a conjunction of clauses, that is if f has the form

$$f = c_1 \wedge \cdots \wedge c_n,$$

where c_i is a clause for all $i = 1, \dots, n$. □

Corollary 25 If $f = k_1 \wedge \cdots \wedge k_n$ is in conjunctive normal form, then we have

$$\models f \quad \text{iff} \quad \models k_i \quad \text{for all } i = 1, \dots, n. \quad \square$$

As the connective \wedge is associative, commutative, and idempotent it is advantageous to use a set notation for clauses. Therefore, if a formula

$$f = k_1 \wedge \cdots \wedge k_n$$

is in connective normal form, we will write

$$f = \{k_1, \dots, k_n\}.$$

We provide an example: If p , q , and r are propositional variables, the formula

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

is in conjunctive normal form. In set notation this is written as

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

If $f = c_1 \wedge \cdots \wedge c_n$ is a formula in cnf then, according to proposition 23 and corollary 25 f is a tautology iff all clauses c_i are trivial. We present an algorithm capable of transforming any formula F into cnf. According to the argument given above, this algorithm can then be used to check if a formula is a tautology.

1. Eliminate all occurrences of the connective “ \leftrightarrow ” using the equivalence

$$\models (f \leftrightarrow g) \leftrightarrow (f \rightarrow g) \wedge (g \rightarrow f)$$

2. Eliminate all occurrences of the connective “ \rightarrow ” using the equivalence

$$\models (f \rightarrow g) \leftrightarrow \neg f \vee g$$

3. Simplify the formula using the following equivalences:

$$(a) \models \neg \perp \leftrightarrow \top$$

$$(b) \models \neg \top \leftrightarrow \perp$$

$$(c) \models \neg \neg f \leftrightarrow f$$

$$(d) \models \neg(f \wedge g) \leftrightarrow \neg f \vee \neg g$$

$$(e) \models \neg(f \vee g) \leftrightarrow \neg f \wedge \neg g$$

After this step, the negation symbol “ \neg ” will only appear in front of a propositional variable. A formula with the property that the negation symbol is only applied to a propositional variable is said to be in *negation normal form*.

4. Next, the formula is expanded using the following laws of distributivity:

$$\models f \vee (g \wedge h) \leftrightarrow (f \vee g) \wedge (f \vee h) \quad \text{and} \quad \models (f \wedge g) \vee h \leftrightarrow (f \vee h) \wedge (g \vee h).$$

This will push the connective “ \vee ” inside the formula so that it no longer appears on top of the connective “ \wedge ”.

5. In the last step we transform the formula into set notation by first writing the clauses as sets and then collecting the sets of clauses.

It should be noted that the size of the formula can grow in the expansion step. This is due to the fact that in the equivalence

$$f \vee (g \wedge h) \leftrightarrow (f \vee g) \wedge (f \vee h)$$

the formula f occurs twice on the right hand side but only once on the left hand side.

We illustrate the algorithm using the formula

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

1. As the formula does not contain the connective “ \leftrightarrow ”, the first step can be skipped.
2. Eliminating the connective “ \rightarrow ” results in the formula

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

3. Transforming this into negation normal form we arrive at

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

4. Expanding this we arrive at

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

5. In order to transform this into set notation we first transform the clauses $p \vee (p \vee \neg q)$ and $\neg q \vee (p \vee \neg q)$ into the sets

$$\{p, p, \neg q\} \quad \text{and} \quad \{\neg q, p, \neg q\}.$$

Now these sets are equal and therefore the conjunctive normal form is

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

You should note that the conversion to set notation has simplified the formula considerably.

3.3.3 Implementing the Algorithm

In this section, we develop a number of functions that can be used to transform a propositional formula f into conjunctive normal form. We start with the function

$$\mathbf{elimIff} : \mathcal{F} \rightarrow \mathcal{F}$$

that takes a propositional formula f and eliminates the connective “ \leftrightarrow ”. The expression $\mathbf{elimIff}(f)$ is defined by induction on the formula f as follows:

1. If $f = \top$, or $f = \perp$, or if f is a propositional variable p , then there is nothing to do:
 - (a) $\mathbf{elimIff}(\top) = \top$.
 - (b) $\mathbf{elimIff}(\perp) = \perp$.
 - (c) $\mathbf{elimIff}(p) = p$ for all $p \in \mathcal{P}$.

2. If f has the form $f = \neg g$, then we have to eliminate the connective “ \leftrightarrow ” from g recursively:

$$\mathbf{elimIff}(\neg g) = \neg \mathbf{elimIff}(g).$$

3. If f has the form $f = g_1 \wedge g_2$, then we have to eliminate the connective “ \leftrightarrow ” from g_1 and g_2 :

$$\mathbf{elimIff}(g_1 \wedge g_2) = \mathbf{elimIff}(g_1) \wedge \mathbf{elimIff}(g_2).$$

4. If f has the form $f = g_1 \vee g_2$, then we have to eliminate the connective “ \leftrightarrow ” from g_1 and g_2 :

$$\mathbf{elimIff}(g_1 \vee g_2) = \mathbf{elimIff}(g_1) \vee \mathbf{elimIff}(g_2).$$

5. If f has the form $f = g_1 \rightarrow g_2$, then we have to eliminate the connective “ \leftrightarrow ” from g_1 and g_2 :

$$\text{elimIff}(g_1 \rightarrow g_2) = \text{elimIff}(g_1) \rightarrow \text{elimIff}(g_2).$$

observe that the last three cases are very similar. We will make use of this fact in the implementation.

6. If f has the form $f = g_1 \leftrightarrow g_2$, then we will use the equivalence

$$(g_1 \leftrightarrow g_2) \leftrightarrow ((g_1 \rightarrow g_2) \wedge (g_2 \rightarrow g_1)).$$

However, we have to be aware that the connective “ \leftrightarrow ” might still occur in g_1 or g_2 . Therefore, we have

$$\text{elimIff}(g_1 \leftrightarrow g_2) = \text{elimIff}((g_1 \rightarrow g_2) \wedge (g_2 \rightarrow g_1)).$$

Figure 3.4 on page 82 shows the procedure `elimIff`. We were able to combine the cases $f = g_1 \wedge g_2$, $f = g_1 \vee g_2$, and $f = g_1 \rightarrow g_2$, as in all these cases we just have to apply the function recursively to the components g_1 and g_2 .

```

1  procedure elimIff(f);
2      case
3          when f = 1          => return 1;
4          when f = 0          => return 0;
5          when is_string(f) => return f;
6          when f(1) = "-"    => return [ "-", elimIff( f(2) ) ];
7          when f(2) in { "*", "+", "->" }
8              => return [ elimIff( f(1) ), f(2), elimIff( f(3) ) ];
9          when f(2) = "<->" => return
10              elimIff( [ [ f(1), "->", f(3) ], "*", [ f(3), "->", f(1) ] ] );
11          otherwise          => print("error in elimIff( ", f, ")" );
12      end case;
13  end elimIff;

```

Figure 3.4: Elimination of “ \leftrightarrow ”.

Next, we study the elimination of the connective “ \rightarrow ”. Figure 3.5 on page 83 provides the implementation. The idea is the same as when eliminating the connective “ \leftrightarrow ”. The only difference is that we use the equivalence

$$(g_1 \rightarrow g_2) \leftrightarrow (\neg g_1 \vee g_2).$$

Furthermore, we can make use of the fact that we have already eliminated the connective “ \leftrightarrow ”.

Next, we show how to transform a formula into negation normal form. Figure 3.6 on page 84 shows the implementation. We have defined two mutually recursive functions `nnf` and `neg`. The function `neg(f)` computes the negation normal form of $\neg f$, while `nnf(f)` computes the negation normal form of f , we have

$$\text{neg}(f) = \text{nnf}(\neg f).$$

Most of the work is done in the function `neg`, as this function implements the DeMorgan laws

$$\neg(f \wedge g) \leftrightarrow (\neg f \vee \neg g) \quad \text{and} \quad \neg(f \vee g) \leftrightarrow (\neg f \wedge \neg g).$$

Conceptually, the transformation into negation normal form is described by the following equations:

1. $\text{nnf}(\top) = \top$,

```

1  procedure elimImp(f);
2      case
3          when f = 1          => return 1;
4          when f = 0          => return 0;
5          when is_string(f) => return f;
6          when f(1) = "-"    => return [ "-", elimImp(f(2)) ];
7          when f(2) in { "*", "+" }
8              => return [ elimImp(f(1)), f(2), elimImp(f(3)) ];
9          when f(2) = "->"  => return elimImp( [ [ "-", f(1) ], "+", f(3) ] );
10         otherwise          => print("error in elimImp( ", f, ")" );
11     end case;
12 end elimImp;

```

Figure 3.5: Elimination of “ \rightarrow ”.

2. $\text{nnf}(\perp) = \perp$,
3. $\text{nnf}(\neg f) = \text{neg}(f)$,
4. $\text{nnf}(f_1 \wedge f_2) = \text{nnf}(f_1) \wedge \text{nnf}(f_2)$,
5. $\text{nnf}(f_1 \vee f_2) = \text{nnf}(f_1) \vee \text{nnf}(f_2)$.

The auxiliary function **neg** is specified as follows:

1. $\text{neg}(\top) = \text{nnf}(\neg \top) = \perp$,
2. $\text{neg}(\perp) = \text{nnf}(\neg \perp) = \top$,
3. $\text{neg}(p) = \text{nnf}(\neg p) = \neg p$ for all propositional variables p ,
4. $\text{neg}(\neg f) = \text{nnf}(\neg \neg f) = \text{nnf}(f)$,
5.
$$\begin{aligned}
 & \text{neg}(f_1 \wedge f_2) \\
 &= \text{nnf}(\neg(f_1 \wedge f_2)) \\
 &= \text{nnf}(\neg f_1 \vee \neg f_2) \\
 &= \text{nnf}(\neg f_1) \vee \text{nnf}(\neg f_2) \\
 &= \text{neg}(f_1) \vee \text{neg}(f_2),
 \end{aligned}$$
6.
$$\begin{aligned}
 & \text{neg}(f_1 \vee f_2) \\
 &= \text{nnf}(\neg(f_1 \vee f_2)) \\
 &= \text{nnf}(\neg f_1 \wedge \neg f_2) \\
 &= \text{nnf}(\neg f_1) \wedge \text{nnf}(\neg f_2) \\
 &= \text{neg}(f_1) \wedge \text{neg}(f_2).
 \end{aligned}$$

As the last step we implement the procedure to expand a formula using the equivalence

$$f \vee (g \wedge h) \leftrightarrow (f \vee g) \wedge (f \vee h)$$

This procedure will also transform the formula into set notation, so that a formula will be represented as set of sets of literals. The procedure is given in figure 3.7 on page 3.7. We explain the details of this implementation below:

```

1  procedure nnf(f);
2      case
3          when f = 1          => return 1;
4          when f = 0          => return 0;
5          when is_string(f) => return f;
6          when f(1) = "-"    => return neg( f(2) );
7          when f(2) = "*"    => return [ nnf( f(1) ), "*", nnf( f(3) ) ];
8          when f(2) = "+"    => return [ nnf( f(1) ), "+", nnf( f(3) ) ];
9          otherwise          => print("error in nnf( ", f, ")" );
10     end case;
11 end nnf;
12
13 procedure neg(f);
14     case
15         when f = 1          => return 0;
16         when f = 0          => return 1;
17         when is_string(f) => return [ "-", f ];
18         when f(1) = "-"    => return nnf( f(2) );
19         when f(2) = "*"    => return [ neg( f(1) ), "+", neg( f(3) ) ];
20         when f(2) = "+"    => return [ neg( f(1) ), "*", neg( f(3) ) ];
21         otherwise          => print("error in neg( ", f, ")" );
22     end case;
23 end neg;

```

Figure 3.6: Computing the negation normal form.

1. To begin with, we reflect how we can represent \top in set notation. As \top is the neutral element of the connective “ \wedge ” we have

$$k_1 \wedge \cdots \wedge k_n \wedge \top \leftrightarrow k_1 \wedge \cdots \wedge k_n.$$

If k_1, \dots, k_n are clauses, the equivalence given above can be transformed into set notation as follows:

$$\{k_1, \dots, k_n, \top\} \leftrightarrow \{k_1, \dots, k_n\}$$

We agree that this equivalence also holds for $n = 0$. Then we have

$$\{\top\} \leftrightarrow \{\}$$

and therefore we interpret the empty set of clauses as \top .

These considerations explain line 3 of the procedure **cnf**.

2. Next, we have to think how to represent \perp in set notation. As \perp is the neutral element of the connective “ \vee ”, we have the following equivalence:

$$L_1 \vee \cdots \vee L_n \vee \perp \leftrightarrow L_1 \vee \cdots \vee L_n.$$

If L_1, \dots, L_n are literals, this equivalence takes the following form in set notation:

$$\{L_1, \dots, L_n, \perp\} \leftrightarrow \{L_1, \dots, L_n\}$$

We agree that this equivalence is also true for $n = 0$. Therefore we have

$$\{\perp\} \leftrightarrow \{\}.$$

Hence, an empty set of literals is interpreted as \perp . This empty set is to be regarded as a single clause. Therefore, In order to represent the formula \perp in cnf in set notation, we have

This explains line 4 of the procedure `cnf`.

4. If the formula f that is to be turned into cnf set notation has the form

then g has to be a propositional variable, as we always start with a formula in negation normal form. Therefore, f is a literal and the cnf of F in set notation is given as $\{\{f\}\}$. This result is returned in line 6.

- $$\text{cnf}(f_1) = \{h_1, \dots, h_m\} \quad \text{and} \quad \text{cnf}(f_2) = \{k_1, \dots, k_n\}.$$

$$\text{cnf}(f_1 \wedge f_2) = \text{cnf}(f_1) \cup \text{cnf}(f_2).$$

6. If F has the form

$$\text{cnf}(f_1) = \{h_1, \dots, h_m\} \quad \text{and} \quad \text{cnf}(f_2) = \{k_1, \dots, k_n\}.$$

$$\begin{aligned} & f_1 \vee f_2 \\ \Leftrightarrow & (h_1 \wedge \cdots \wedge h_m) \vee (k_1 \wedge \cdots \wedge k_n) \\ \Leftarrow & (h_1 \vee k_1) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_1) \quad \wedge \\ & \qquad \vdots \qquad \qquad \qquad \qquad \qquad \vdots \\ & (h_1 \vee k_n) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_n) \\ \Leftarrow & \{h_i \vee k_j : i \in \{1, \dots, m\}, j \in \{1, \dots, n\}\} \end{aligned}$$
$$\mathbf{cnf}(f_1 \vee f_2) = \{h \cup k \mid h \in \mathbf{cnf}(f_1) \wedge k \in \mathbf{cnf}(f_2)\}.$$

Finally, figure 3.8 on page 86 shows how all the procedures discussed so far can be combined into a single procedure that takes an arbitrary propositional formula f and transforms this formula into `cnf` in set notation.

```

1  procedure cnf(f);
2      case
3          when f = 1      => return { };
4          when f = 0      => return { {} };
5          when is_string(f) => return { { f } };
6          when f(1) = "-" => return { { f } };
7          when f(2) = "*" => return cnf( f(1) ) + cnf( f(3) );
8          when f(2) = "+" => return { k1 + k2 : k1 in cnf(f(1)), k2 in cnf(f(3)) };
9          otherwise => print("error in cnf( ", f, ")" );
10     end case;
11 end cnf;

```

Figure 3.7: Computing the conjunctive normal form.

```

1  procedure normalize(f);
2      n1 := elimIff(f);
3      n2 := elimFolgt(n1);
4      n3 := nnf(n2);
5      n4 := cnf(n3);
6      return n4;
7  end normalize;

```

Figure 3.8: Normalizing a formula.

3.4 Formal Proofs

The main subject of logic is the notion of a *proof*. The aim of this section is to formalize this notion. If $\{f_1, \dots, f_n\}$ is a set of formulæ and g is another formula, then we can ask whether the formula g can be *inferred* from the formula f_1, \dots, f_n , that is we can ask whether

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

holds. There are different options to answer this question. One option would be to transform the formula $f_1 \wedge \dots \wedge f_n \rightarrow g$ into conjunctive normal form. This would result in a set $\{c_1, \dots, c_n\}$ of clauses, which is equivalent to the formula

$$f_1 \wedge \dots \wedge f_n \rightarrow g.$$

Now this formula is a tautology iff each of the clauses c_1, \dots, c_n is trivial. Unfortunately, this approach is quite inefficient. Let us investigate an example and check, whether the formula $p \rightarrow r$ can be inferred from the formulæ $p \rightarrow q$ and $q \rightarrow r$. Following the approach sketched above, we have to transform the formula

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

into conjunctive normal form. After a lengthy calculation, the conjunctive normal form of the formula above is found to be

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

and indeed all of the clauses are trivial.

In the chapter on predicate logic we will see, that the approach used above does no longer work when we deal with quantifiers. The reason is, that in general there is no algorithm to decide whether a formula involving quantifiers is a tautology. Therefore, we now present a second approach to check whether a given formula is a logical consequence of a given set of formulæ. The idea is, to use so called *inference rules* to infer a formula from other formulæ. The notion of an *inference rule* is formalized in the following definition.

Definition 26 (Inference Rule) An *inference rule* is a pair $\langle \{f_1, \dots, f_n\}, c \rangle$ where $\{f_1, \dots, f_n\}$ is a set of formulæ and c is a formulæ. The formulæ f_1, \dots, f_n are the *premises*, the formula c is the *conclusion* of the inference rule.

Notation: If the pair $\langle \{f_1, \dots, f_n\}, c \rangle$ is an inference rule, then we write this as

$$\frac{f_1 \quad \dots \quad f_n}{c}.$$

This is read as follows: “From f_1, \dots, f_n we can infer c .”

□

Examples of inference rules:

<i>Modus Ponens:</i>	<i>Modus Tollens:</i>	<i>Modus Tollendo Tollens:</i>
$\frac{p \quad p \rightarrow q}{q}$	$\frac{\neg q \quad p \rightarrow q}{\neg p}$	$\frac{\neg p \quad p \rightarrow q}{\neg q}$

The definition of inference rules given above does not restrict the formulæ that are used as premises and conclusion in any way. However, not all inference rules make sense. If we want to use an inference rule in a proof, then we should only those rules that are *correct* in the sense that the premises imply the conclusion. Therefore, the notion of *correctness* is defined as follows.

Definition 27 (Correct) The inference rule

$$\frac{f_1 \quad \cdots \quad f_n}{c}$$

is *correct* if and only if we have $\models f_1 \wedge \cdots \wedge f_n \rightarrow c$. \square

Turning back to our examples, we see that the rules denoted as “*modus ponens*” and “*modus ponendo tollens*” are correct. However, the rule “*modus tollendo tollens*” is not correct.

In the following, we will only work with clauses. This is no real restriction, as we have already seen that all formulæ can be transformed into a conjunction of clauses. Furthermore, in many applications of propositional logic, the formulæ are already given as clauses. We next present an inference rule for clauses.

Definition 28 (Cut Rule) If p is a propositional variable and c_1 and c_2 are sets of literals, then the following inference rule will be denoted as the *cut rule*:

$$\frac{c_1 \cup \{p\} \quad \{\neg p\} \cup c_2}{c_1 \cup c_2}.$$

\square

The cut rule is a very general rule. If we put $c_1 = \{\}$ and $c_2 = \{q\}$ in the definition given above, then the following rule is a special case of the cut rule:

$$\frac{\{\} \cup \{p\} \quad \{\neg p\} \cup \{q\}}{\{\} \cup \{q\}}$$

If we interpret these sets as disjunctions, then we arrive at the following rule:

$$\frac{p \quad \neg p \vee q}{q}$$

As the formula $\neg p \vee q$ is equivalent to $p \rightarrow q$, we see that the rule *modus ponens* is a special case of the cut rule. In the same way, the rule *modus tollens* can be shown to be a special case of the cut rule. If we take the definition of the cut rule and put $c_1 = \{\neg q\}$ and $c_2 = \{\}$, then we arrive at the rule *modus tollens*.

Proposition 29 *The cut rule is correct.*

Proof: We have to show the following

$$\models (c_1 \vee p) \wedge (\neg p \vee c_2) \rightarrow c_1 \vee c_2$$

can be proved. In order to do so we transform the formula given above into conjunctive normal form:

$$\begin{aligned} & (c_1 \vee p) \wedge (\neg p \vee c_2) \rightarrow c_1 \vee c_2 \\ \leftrightarrow & \neg((c_1 \vee p) \wedge (\neg p \vee c_2)) \vee c_1 \vee c_2 \\ \leftrightarrow & \neg(c_1 \vee p) \vee \neg(\neg p \vee c_2) \vee c_1 \vee c_2 \\ \leftrightarrow & (\neg c_1 \wedge \neg p) \vee (p \wedge \neg c_2) \vee c_1 \vee c_2 \\ \leftrightarrow & (\neg c_1 \vee p \vee c_1 \vee c_2) \wedge (\neg c_1 \vee \neg c_2 \vee c_1 \vee c_2) \wedge (\neg p \vee p \vee c_1 \vee c_2) \wedge (\neg p \vee \neg c_2 \vee c_1 \vee c_2) \\ \leftrightarrow & \top \wedge \top \wedge \top \wedge \top \\ \leftrightarrow & \top \end{aligned}$$

Since the conjunctive normal form is just \top , we have established the correctness of the cut rule. \square

Next, we formalize the notion of a proof.

Definition 30 (Provability, \vdash) Assume M is a set of clauses and f is a single clause. The formulæ from M are called the *assumptions*. The goal is to prove a formula f using these assumptions. In order to do so we will define inductively a relation

$$M \vdash f.$$

We will read the expression “ $M \vdash f$ ” as “ f is provable from M ”. The inductive definition of this notion is as follows:

1. A set M of assumptions proves all of its assumptions:

$$\text{If } f \in M, \text{ then } M \vdash f.$$

2. If the clauses $c_1 \cup \{p\}$ and $\{\neg p\} \cup c_2$ are provable from M , then we can use the cut rule to prove the clause $c_1 \cup c_2$:

$$\text{If } M \vdash c_1 \cup \{p\} \text{ and } M \vdash \{\neg p\} \cup c_2, \text{ then } M \vdash c_1 \cup c_2. \quad \square$$

Example: To illustrate the notion of provability, we show the following:

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

This example will also establish our way to denote proofs.

1. From $\{\neg p, q\}$ and $\{\neg q, \neg p\}$, using the cut rule, we can conclude $\{\neg p, \neg p\}$. As we have $\{\neg p, \neg p\} = \{\neg p\}$, this step is denoted as

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

This example shows that the clause $c_1 \cup c_2$ may contain less elements than the sum $\#c_1 + \#c_2$. This happens if there are literals occurring in both c_1 and in c_2 .

2. $\{\neg q, \neg p\}, \{p, \neg q\} \vdash \{\neg q\}$.
3. $\{p, q\}, \{\neg q\} \vdash \{p\}$.
4. $\{\neg p\}, \{p\} \vdash \{\}$.

As another example, we show the provability of $p \rightarrow r$ from $p \rightarrow q$ and $q \rightarrow r$. In order to do so, we first transform these formula into clauses in set notation:

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

Therefore, we have $M = \{ \{\neg p, q\}, \{\neg q, r\} \}$ and have to show

$$M \vdash \{\neg p, r\}.$$

The proof is a single application of the cut rule:

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

3.4.1 Properties of the Notion of Provability

There are two very important properties of the relation \vdash that will be formulated next.

Theorem 31 (Correctness) If $\{c_1, \dots, c_n\}$ is a set of clauses and c is a single clause, then we have:

$$\text{If } \{c_1, \dots, c_n\} \vdash c, \quad \text{then } \models c_1 \wedge \dots \wedge c_n \rightarrow c. \quad \square$$

It would be nice if we were able to invert this theorem. However, the converse of the correctness theorem is only true in a special case: It is only true if c is the empty clause, that is if $c = \perp$.

Proposition 32 (Refutation Completeness) If $\{c_1, \dots, c_n\}$ is a set of clauses, then we have the following:

$$\text{If } \models c_1 \wedge \dots \wedge c_n \rightarrow \perp, \quad \text{then } \{c_1, \dots, c_n\} \vdash \{\}. \quad \square$$

Unfortunately, we do not have the time to present proofs of these theorems.

3.5 The Algorithm of Davis and Putnam

In practical applications, it is often necessary to find a propositional valuation \mathcal{I} such that, for a given set of clauses C , the following holds true:

$$\text{eval}(c, \mathcal{I}) = \text{true} \quad \text{for all } c \in C.$$

In this case, the set of clauses is called *satisfiable* and the propositional valuation \mathcal{I} is called a *solution* of C . The notion of satisfiability is strongly connected with the notion of tautology-hood. For a given set of clauses $\{c_1, \dots, c_n\}$ we have

$$\{c_1, \dots, c_n\} \text{ is not satisfiable} \quad \text{iff} \quad \models c_1 \wedge \dots \wedge c_n \rightarrow \perp.$$

In this section, we present an algorithm that can be used to check whether a set of clauses is satisfiable. This algorithm is due to Davis and Putnam [DLL62]. Refinements of this algorithm are, for example, used to prove the correctness of digital circuits.

In order to motivate this algorithm, let us first consider those sets of clauses where the satisfiability is obvious. Consider the following example:

$$C_1 = \{ \{p\}, \{\neg q\}, \{r\}, \{\neg s\}, \{\neg t\} \}$$

The set C_1 represents the propositional formula

$$p \wedge \neg q \wedge r \wedge \neg s \wedge \neg t.$$

Obviously, C_1 is satisfiable: The propositional valuation

$$\mathcal{I} = \{ \langle p, \text{true} \rangle, \langle q, \text{false} \rangle, \langle r, \text{true} \rangle, \langle s, \text{false} \rangle, \langle t, \text{false} \rangle \}$$

is a solution of C_1 . Consider another example:

$$C_2 = \{ \{\}, \{p\}, \{\neg q\}, \{r\} \}$$

This set of clauses represents the formula

$$\perp \wedge p \wedge \neg q \wedge r$$

which is obviously not satisfiable. As a final example, consider

$$C_3 = \{ \{p\}, \{\neg q\}, \{\neg p\} \}.$$

This set of clauses represents the formula

$$p \wedge \neg q \wedge \neg p$$

and, again, is obvious that C_3 is not satisfiable, since a solution \mathcal{I} would have to evaluate p both as **true** and **false**. These examples motivate the following definitions.

Definition 33 (Unit Clause) A clause c is called a *unit clause* iff c contains just one literal, that is, if we have

$$c = \{p\} \quad \text{or} \quad c = \{\neg p\}$$

for a propositional variable p .

Definition 34 (Trivial Set of Clauses) A set of clauses C is called *trivial* iff we have one of the following cases:

1. The empty clause is a member of C :

$$\{\} \in C.$$

In this case, C is obviously unsatisfiable.

2. The set C only contains unit clauses with different propositional variables. If the set of all propositional variables is denoted as \mathcal{P} , then this requirement can be formalized as follows:

$$\forall c \in C : \text{card}(c) = 1 \quad \text{and} \quad \forall p \in \mathcal{P} : \neg(\{p\} \in C \wedge \{\neg p\} \in C).$$

Then the propositional valuation

$$\mathcal{I} := \{\langle p, \text{true} \rangle \mid \{p\} \in C\} \cup \{\langle p, \text{false} \rangle \mid \{\neg p\} \in C\}$$

is a solution of C . □

The main idea of the algorithm of Davis and Putnam is to try to simplify a given set of clauses until all clauses are unit clauses. There are three methods to simplify a given set of clauses:

1. Use of the cut rule,
2. Use of the subsumption rule,
3. Case distinction.

We discuss these methods in detail in the following.

3.5.1 Simplifications via the Cut Rule

A typical application of the cut rule takes the following form:

$$\frac{c_1 \cup \{p\} \quad \{\neg p\} \cup c_2}{c_1 \cup c_2}$$

In general, the clause $c_1 \cup c_2$ will contain more literals than the premises $c_1 \cup \{p\}$ and $\{\neg p\} \cup c_2$. If the clause $c_1 \cup \{p\}$ has $m+1$ literals and the clause $\{\neg p\} \cup c_2$ has $n+1$ literals, then the conclusion $c_1 \cup c_2$ contains up to $m+n$ literals. Of course, this number could be less than $m+n$: This is the case if there are literals that are members of both c_1 and c_2 . In most cases, $m+n$ is bigger than both $m+1$ and $n+1$. Only if either $n=0$ or $m=0$ we can be sure that $m+n$ is less than either $m+1$ or $n+1$. We have $n=0$ or $m=0$ if either of the two clauses $c_1 \cup \{p\}$ and $\{\neg p\} \cup c_2$ consists of a single literal and therefore is a unit clause. As we want to simplify our clauses, we will only use the cut rule if one of its premises is a unit clause. These restrictions of the cut rule are called *unit cuts*. In order to conduct all possible unit cuts with a given literal l , we define the function

$$\text{unitCut} : 2^{\mathcal{C}} \times \mathcal{L} \rightarrow 2^{\mathcal{C}}$$

in a way that, for a given set of clauses C and a literal l the function $\text{unitCut}(C, l)$ performs all cut rules that are possible with the unit clause $\{l\}$:

$$\text{unitCut}(C, l) = \left\{ c \setminus \{\overline{l}\} \mid c \in C \wedge \overline{l} \in c \right\}.$$

3.5.2 Simplification via Subsumption

We demonstrate the subsumption principle with the help of an example. Consider the set

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

Apparently, the clause $\{p\}$ implies the clause $\{p, q, \neg r\}$ since we have

$$\models p \rightarrow q \vee p \vee \neg r.$$

Therefore, the clause $\{p, q, \neg r\}$ can be dropped from C and C can be simplified into

$$C' := \{\{p\}\} \cup M.$$

In general, we define that a clause c is *subsumed* by a unit clause u iff

$$u \subseteq c.$$

Therefore, if C is a set of clauses such that $c \in C$ and $u \in C$ and if, furthermore, c is subsumed by u , then C can be simplified to $C - \{c\}$. To facilitate the implementation of this we define a function

$$\text{subsume} : 2^C \times \mathcal{L} \rightarrow 2^C$$

that takes a set of clauses C and a unit clause $\{l\} \in C$. The function simplifies C by elimination all clauses from C that are subsumed by $\{l\}$. Of course, the unit clause $\{l\}$ itself is retained. Hence we define

$$\text{subsume}(C, l) := (C \setminus \{c \in C \mid l \in c\}) \cup \{\{l\}\} = \{c \in C \mid l \notin c\} \cup \{\{l\}\}.$$

Observe that we have to add $\{l\}$ into the result, as the set $\{c \in C \mid l \notin c\}$ does not contain the unit clause $\{l\}$.

3.5.3 Simplification via Case Distinction

If we would only use unit cuts and unit subsumption, then our set of inference rules would be incomplete. We need another method to simplify sets of clauses. Therefore, we add the *principle of case distinction*. This principle is based on the following proposition.

Proposition 35 *If C is a set of clauses and p is a propositional variable, then C is satisfiable if and only if either $C \cup \{\{p\}\}$ or $C \cup \{\{\neg p\}\}$ is satisfiable.*

Proof:

“ \Rightarrow ”: If C is satisfied via the propositional valuation \mathcal{I} , then there are two cases for $\mathcal{I}(p)$:

1. If $\mathcal{I}(p) = \mathbf{true}$, then $C \cup \{\{p\}\}$ is satisfiable.
2. If $\mathcal{I}(\neg p) = \mathbf{true}$, then $C \cup \{\{\neg p\}\}$ is satisfiable.

“ \Leftarrow ”: As C is both a subset of $C \cup \{\{p\}\}$ and also a subset of $C \cup \{\{\neg p\}\}$, it is obvious that C is satisfiable if either of these sets is satisfiable. \square

Therefore, in order to check whether a given set of clauses C is satisfiable, we choose a propositional variable p that occurs in C . Subsequent, we form the sets

$$C_1 := C \cup \{\{p\}\} \quad \text{and} \quad C_2 := C \cup \{\{\neg p\}\}$$

and recursively investigate whether C_1 is satisfiable. If we find a solution for C_1 , then this solution is also a solution of the set C . Otherwise, we check whether C_2 is satisfiable. Again, if we find a solution for C_2 , then this is also a solution for C . On the other hand, if we neither find a solution for C_1 nor for C_2 , then C has to be unsolvable, too. The recursive investigation of C_1 and C_2 is

simpler than the investigation of C as we now have the additional unit clauses $\{p\}$ or $\{\neg p\}$ that can be used to simplify the corresponding sets via unit cuts and unit subsumptions. This leads to the following algorithm:

Given: A set C of clauses.

Wanted: A propositional valuation \mathcal{I} that solves C , that is:

$$\mathcal{I}(c) = \text{true} \quad \text{for all } c \in C.$$

Algorithm: Perform the following steps.

1. Conduct all possible unit cuts and all possible unit subsumptions.
Note that as a result of unit cuts, new unit clauses might be added to the set C . We repeatedly perform all unit cuts and unit subsumptions until we don't find any new unit clauses.
2. If C is trivial, report the result.
3. Otherwise: Choose a propositional variable p occurring in C but that does not occur in any of its unit clauses.
 - (a) Recursively solve
$$C \cup \{\{p\}\}.$$

If this succeeds, we have a solution of C .
 - (b) Otherwise, recursively solve
$$C \cup \{\{\neg p\}\}.$$

If this succeeds, we have a solution of C .
 - (c) Otherwise, C is not satisfiable.

In order to implement this algorithm, it is beneficial to combine the functions *unitCut()* and *subsume()*. We therefore define the function

$$\text{reduce} : 2^C \times \mathcal{L} \rightarrow 2^C$$

as follows

$$\text{reduce}(C, l) = \{c \setminus \{\bar{l}\} \mid c \in C \wedge \bar{l} \in c\} \cup \{c \in C \mid \bar{l} \notin c \wedge l \notin c\} \cup \{\{l\}\}.$$

According to this definition, the set $\text{reduce}(C, l)$ contains the results of all cuts with the unit clause $\{l\}$ and, furthermore, those clauses of C are retained, that are not connected with the literal l as neither l nor \bar{l} occurs in c . Finally the unit clause $\{l\}$ is added. Therefore, the sets $C \cup \{l\}$ and $\text{reduce}(C, l)$ are equivalent: If either of these sets is solvable, so is the other one.

3.5.4 An Example

Define the set C as follows:

$$C := \left\{ \{p, q, s\}, \{\neg p, r, \neg t\}, \{r, s\}, \{\neg r, q, \neg p\}, \right. \\ \left. \{\neg s, p\}, \{\neg p, \neg q, s, \neg r\}, \{p, \neg q, s\}, \{\neg r, \neg s\}, \{\neg p, \neg s\} \right\}.$$

Using the algorithm of Davis and Putnam we show that C is not satisfiable.

1. As C does not contain any unit clauses, the first step doesn't change the set of clauses.
2. Next, we check whether C is trivial. Apparently, C is not trivial, so we have to proceed.

3. We have to choose a propositional variable occurring in C . It is best to choose a variable that does occur in many clauses of C . Therefore, let us take the propositional variable p . Note that if we choose a different variable, our computation might get longer. Of course, in the end we would get the same result.

(a) Recursively, we have to check whether the set

$$C_0 := C \cup \{\{p\}\}$$

is satisfiable. Therefore, we compute

$$C_1 := \text{reduce}(C_0, p) = \left\{ \{r, \neg t\}, \{r, s\}, \{\neg r, q\}, \{\neg q, s, \neg r\}, \{\neg r, \neg s\}, \{\neg s\}, \{p\} \right\}.$$

Now the set C_1 contains the unit clause $\{\neg s\}$ that can be used to reduce this set:

$$C_2 := \text{reduce}(C_1, \neg s) = \left\{ \{r, \neg t\}, \{r\}, \{\neg r, q\}, \{\neg q, \neg r\}, \{\neg s\}, \{p\} \right\}.$$

This set contains the new unit clause $\{r\}$:

$$C_3 := \text{reduce}(C_2, r) = \left\{ \{r\}, \{q\}, \{\neg q\}, \{\neg s\}, \{p\} \right\}$$

As C_3 contains the unit clause $\{q\}$ we define:

$$C_4 := \text{reduce}(C_3, q) = \left\{ \{r\}, \{q\}, \{\}, \{\neg s\}, \{p\} \right\}.$$

Now C_4 contains the empty clause and therefore is not satisfiable.

(b) According to the algorithm, we now define

$$C_5 := C \cup \{\{\neg p\}\}$$

and check whether this set is satisfiable. To this end we define

$$C_6 = \text{reduce}(C, \neg p) = \left\{ \{q, s\}, \{r, s\}, \{\neg s\}, \{\neg q, s\}, \{\neg r, \neg s\}, \{\neg p\} \right\}.$$

The set C_6 contains the unit clause $\{\neg s\}$. Therefore, we define

$$C_7 = \text{reduce}(C_6, \neg s) = \left\{ \{q\}, \{r\}, \{\neg s\}, \{\neg q\}, \{\neg p\} \right\}.$$

The set C_7 contains the unit clause $\{q\}$. Hence we define:

$$C_8 = \text{reduce}(C_7, q) = \left\{ \{q\}, \{r\}, \{\neg s\}, \{\}, \{\neg p\} \right\}.$$

As C_8 contains the empty clause, both C_8 and the given set C are unsatisfiable. \square

In the example above, we only had to apply one case distinction. In general, any recursive invocation of the method could itself give rise to further case distinctions.

3.5.5 Implementing the Algorithm of Davis and Putnam

Next, we implement the algorithm. Figure 3.9 on page 95 shows the implementation of the procedure `DavisPutnam`. This procedure is called with two arguments: `Clauses` and `Literals`.

1. `Clauses` is a set of clauses.
2. `Literals` is a set of literals.

If the literals from `Literals` are turned into unit clauses and added to the set `Clauses`, then the procedure call `DavisPutnam(Clauses, Literals)` checks whether the resulting set of clauses is satisfiable. If a solution \mathcal{I} is found, it is returned as the result. If, on the other hand, the set is not satisfiable, then, instead, the result `false` is returned. We need the second argument `Literals` in the procedure `DavisPutnam` in order to remember the literals that have already been used for unit cuts and subsumptions.

```
1  procedure DavisPutnam( Clauses, Literals );
2      Clauses := saturate(Clauses);
3      if {} in Clauses then
4          return false;
5      end if;
6      if forall c in Clauses | #c = 1 then
7          return Clauses;
8      end if;
9      literal := selectLiteral(Clauses, Literals);
10     Result := DavisPutnam(Clauses + {{literal}}, Literals + { literal });
11     if Result /= false then
12         return Result;
13     end if;
14     notLiteral := negateLiteral(literal);
15     return DavisPutnam(Clauses + {{notLiteral}}, Literals + { notliteral } );
16 end DavisPutnam;
```

Figure 3.9: The procedure `DavisPutnam`.

The implementation that is shown in figure 3.9 works as follows:

1. Line 2 performs all possible unit cuts and unit subsumptions that are possible with the set `Clauses`.
2. Next, we test in line 3 whether the resulting set of clauses contains the empty set and is therefore not satisfiable.
3. Line 6 tests whether all clauses c from the set `Clauses` are unit clauses. In this case, the set `Clauses` is trivial and we return this set as a result.
4. Otherwise, line 9 selects a literal l from the set `Clauses` that has not been used already.
5. Line 10 checks recursively whether the set

$$\text{Clauses} \cup \{\{\text{literal}\}\}$$

is solvable.

- (a) If this set is solvable, we return the solution as result.

(b) Otherwise, we recursively check, whether the set

$$\text{Clauses} \cup \{\{\neg \text{literal}\}\}$$

is solvable.

Next, we discuss the auxiliary procedures. We start with the function **saturate**. This procedure takes a set S of clauses and performs all possible unit cuts and unit subsumptions. The implementation is shown in figure 3.10 on page 96.

```

1  procedure saturate(S);
2      Units := { c in S | #c = 1 };
3      Used := {};
4      while Units /= {} loop
5          unit := arb Units;
6          Used := Used + { unit };
7          literal := arb unit;
8          S := reduce(S, literal);
9          Units := { c in S | #c = 1 } - Used;
10     end loop;
11     return S;
12 end saturate;
```

Figure 3.10: The procedure **saturate**.

The implementation of **saturate** works as follows:

1. Line 2 computes the set **Units** of all unit clauses occurring in S .
2. Line 3 initializes the set **Used**. This set is used for remembering those unit clauses that have already been used.
3. As long as the set **Units** is not empty we choose an arbitrary unit clause in line 5.
4. Line 6 adds this clause to the set of used clauses so that we don't use it again.
5. Line 7 extracts the literal from the unit clause.
6. Line 8 performs all unit cuts and unit subsumptions that can be done with the help of the literal from the chosen unit clause.
7. As the reduction in line 8 can produce new unit clauses we have to collect these new unit clauses in line 9.
8. The loop in line 4 – 10 runs as long as we find new unit clauses. Therefore, when the procedure returns the set of clauses S in line 11, we can be sure the this set can not be reduced further by either unit cuts or unit subsumptions.

The procedure **reduce** is shown in figure 3.11. It is an implementation of the following formula:

$$\text{reduce}(C, l) = \{ c \setminus \{\bar{l}\} \mid c \in C \wedge \bar{l} \in c \} \cup \{ c \in C \mid \bar{l} \notin c \wedge l \notin c \} \cup \{ \{l\} \}.$$

Furthermore, the auxiliary procedures *selectLiteral* and *negateLiteral* are shown in figure 3.12 on page 97.

```

1  procedure reduce( S, l );
2      notL := negateLiteral(l);
3      return { c - { notL } : c in S | notL in c }
4              + { c in S | not notL in c and not l in c }
5              + { {l} };
6  end reduce;

```

Figure 3.11: The procedure `reduce`.

1. The procedure `selectLiteral(S, F)` returns an arbitrary literal from the set S of clauses that is not contained in the set F . In order to do this, the expression “ $+ S$ ” computes the union of all clauses. This is a set of all literals occurring in clauses from S . From this set, the set of literals F is subtracted. Finally, we return an arbitrary element from the resulting set of literals.
2. The procedure `negateLiteral` computes the complement \bar{l} of a given literal l . There are two cases:
 - (a) The literal is a negated variable $\neg p$. In this case, l has the form $l = ["-", p]$ and the variable p is returned.
 - (b) The literal is a propositional variable. In this case, the variable has to be negated.

```

1  procedure selectLiteral( S, F );
2      return arb (+ S - F);
3  end selectLiteral;
4
5  procedure negateLiteral(l);
6      if l(1) = "-" then
7          return l(2);
8      else
9          return [ "-", l ];
10     end if;
11 end negateLiteral;

```

Figure 3.12: The procedures `select` and `negateLiteral`.

The algorithm of Davis and Putnam can be refined in a number of ways. Unfortunately, we don't have the time to discuss these refinements. The article of Moskewicz [MMZ⁺01] describes a number of possible refinements.

Chaff: Engineering an Efficient SAT Solver

written by *M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik*

3.6 The Eight Queens Puzzle

Next, we show how certain combinatorial problems can be transformed into problems of propositional logic. These problems can then be solved using the algorithm of Davis and Putnam. To provide an example we will discuss the 8 queens puzzle. The task is to position 8 queens on a chess board in a way that no queen can attack any of the other queens. In chess, a queen can attack another piece if either of the following is true:

- the piece is in the same row as the queen,
- the piece is in the same column as the queen, or
- the piece is in the same diagonal as the queen.

Figure 3.13 on page 98 presents a chess board with a queen on it. The queen can attack all of the marked positions.

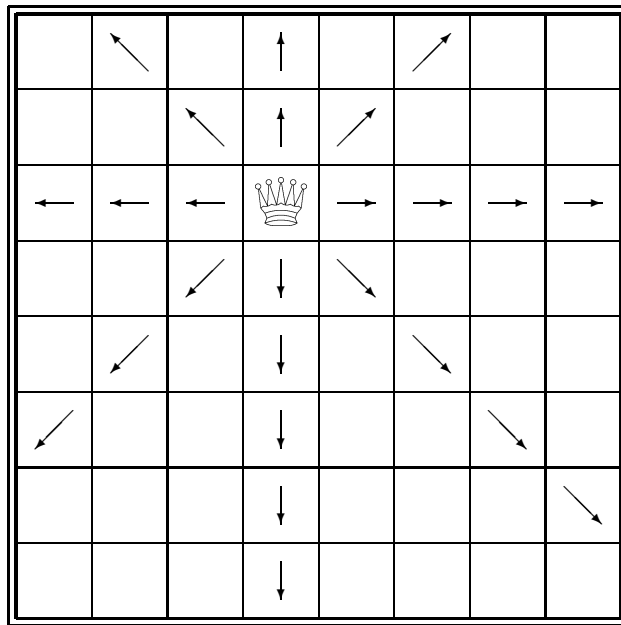


Figure 3.13: The 8 queens puzzle.

As a first step, we have to decide how to represent a chess board and the queens that are positioned on it. The idea is to have a propositional variable for every possible position. If this variable is true, then a queen is put into this position, else the position stays empty. In order to name the positions consistently, we take

$$p_{ij}$$

to represent the position in row number i and column number j . Here, both i and j are elements of the set $\{1, \dots, 8\}$. The rows are enumerated from top to bottom, while the columns are enumerated from left to right. Figure 3.14 on page 99 shows the assignment of variables to the positions on the chess board.

In order to translate this idea into SETL, we implement the procedure `createBoard()`. This procedure returns a list of lists of variables that represent the chess board shown in Figure 3.14. The implementation of this procedure is shown in Figure 3.15 on page 99. Each of the lists contained in the result represents a single column of the board.

In order to understand how this procedure works, we call the procedure with the argument 8, which gives the size of the board. The result is shown in Figure 3.16 on the next page. The

p11	p12	p13	p14	p15	p16	p17	p18
p21	p22	p23	p24	p25	p26	p27	p28
p31	p32	p33	p34	p35	p36	p37	p38
p41	p42	p43	p44	p45	p46	p47	p48
p51	p52	p53	p54	p55	p56	p57	p58
p61	p62	p63	p64	p65	p66	p67	p68
p71	p72	p73	p74	p75	p76	p77	p78
p81	p82	p83	p84	p85	p86	p87	p88

Figure 3.14: Assignments of the variables.

```

1  procedure createBoard(n);
2      return [ [ "p" + i + j : j in [1..n] ] : i in [1..n] ];
3  end createBoard;

```

Figure 3.15: The procedures `createBoard` and `createRow`.

board is represented as a list containing 8 elements. These elements are itself lists and represent the columns of the board.

```

1  [ ["p11", "p12", "p13", "p14", "p15", "p16", "p17", "p18"],
2    ["p21", "p22", "p23", "p24", "p25", "p26", "p27", "p28"],
3    ["p31", "p32", "p33", "p34", "p35", "p36", "p37", "p38"],
4    ["p41", "p42", "p43", "p44", "p45", "p46", "p47", "p48"],
5    ["p51", "p52", "p53", "p54", "p55", "p56", "p57", "p58"],
6    ["p61", "p62", "p63", "p64", "p65", "p66", "p67", "p68"],
7    ["p71", "p72", "p73", "p74", "p75", "p76", "p77", "p78"],
8    ["p81", "p82", "p83", "p84", "p85", "p86", "p87", "p88"] ]

```

Figure 3.16: Output of `createBoard(8)`.

Next, we have to translate the constraints of the eight queens puzzle into propositional logic. We have the following constraints:

- Every row has at most one queen.
- Every column has at most one queen.
- Every diagonal has at most one queen.
- There are eight queens on the board.

The first three statements can all be reduced to the same structure. We have a set of propositional variables

$$V = \{x_1, \dots, x_n\}$$

and we need to find a formula that expresses that at most one of the variables from V is **true**. However, this means that for all $x_i, x_j \in V$ such that $x_i \neq x_j$ we must have

$$\neg(x_i \wedge x_j),$$

because then two different variables cannot be **true** at the same time. According to the laws of DeMorgan we have

$$\neg(x_i \wedge x_j) \leftrightarrow \neg x_i \vee \neg x_j$$

and in set notation this can be expressed as the following clause:

$$\{\neg x_i, \neg x_j\}.$$

Therefore, the formula that expresses that no two different variables from a given set of variables V are true simultaneously, is given as the following set of clauses:

$$\{ \{\neg p, \neg q\} \mid p \in V \wedge q \in V \wedge p \neq q \}.$$

We translate these considerations into a SETL procedure. Figure 3.17 shows the procedure **atMostOne** that takes a set V of propositional variables as argument. The procedure invocation **atMostOne**(V) computes a set of clauses. These clauses are **true** if and only if at most one of the variables from V is true.

```

1  procedure atMostOne(V);
2      return { { [ "-", p ], [ "-", q ] } : p in V, q in V | p /= q };
3  end atMostOne;
```

Figure 3.17: The procedure **atMostOne**.

Using the procedure **atMostOne** we can implement the procedure **atMostOneInRow**. The procedure call

atMostOneInRow(*board*, *row*)

takes a board and a row number and computes a propositional formula that is **true** if that row contains at most one queen. The assumption is, of course, that the argument **board** is the output from the procedure **createBoard**(n). Figure 3.18 shows the implementation of **atMostOneInRow**(): We collect the set of all those variables that denote positions in the given row:

$$\{\text{board}(\text{row})(j) \mid j \in \{1, \dots, 8\}\}.$$

This set is then used as argument of the auxiliary procedure **atMostOne**().

```

1  procedure atMostOneInRow(board, row);
2      return atMostOne({ board(row)(j) : j in {1 .. 8} });
3  end atMostOneInRow;
```

Figure 3.18: The procedure **atMostOneInRow**.

Next, we ensure that we have indeed positioned eight queens on the board. Now if we put a queen in every column, we know that there will be at least eight queens on the board. As we already know that no column can contain more than one queen, we can be sure that every column must indeed contain exactly one queen, for otherwise we cannot have eight queens on the board.

Now the formula that there is a queens in the first column is given as

$$p11 \vee p21 \vee p31 \vee p41 \vee p51 \vee p61 \vee p71 \vee p81.$$

In general, if we want to express that there is a queen in column c where $c \in \{1, \dots, 8\}$, then we can use the formula

$$p1c \vee p2c \vee p3c \vee p4c \vee p5c \vee p6c \vee p7c \vee p8c.$$

Using set notation, we arrive at the following set of clauses:

$$\{\{p1c, p2c, p3c, p4c, p5c, p6c, p7c, p8c\}\}.$$

Figure 3.19 shows a SETL procedure that computes this formula for a given `column`. We have to return a set of clauses here, since the algorithm of Davis and Putnam is designed to work on sets of clauses.

```

1  procedure oneInColumn(board, column);
2      return { { board(row)(column) : row in { 1 .. 8 } } };
3  end oneInColumn;
```

Figure 3.19: The procedure `oneInColumn`.

You might expect that we develop formulæ expressing that every column contains at most one queen and formulæ expressing that every row contains at least one queen. However, these formulæ are redundant: If every column contains at least one queen and, furthermore, every row contains at most one queen, then we can conclude that there are exactly eight queens on the board. However, then there must be exactly one queen per row, for if one row would miss a queen, another row would have to contain two queens, which we have excluded already. Similarly, if one column would contain more than one queen, another column would have to be empty.

Next, we tackle the diagonals. As a first step we think how to characterize the positions of one diagonal. There are two different kinds of diagonals: Lower diagonals and upper diagonals. We start with upper diagonals. The longest upper diagonal contains the propositional variables

$$p81, p72, p63, p54, p45, p36, p27, p18.$$

Now the indices i and j of the variables p_{ij} satisfy the following linear equation:

$$i + j = 9.$$

In general, the indices of an upper diagonal satisfy the linear equation

$$i + j = k,$$

where k is an elements of the set $\{3, \dots, 15\}$. This value k is the argument of the procedure `atMostOneInUpperDiagonal`. This procedure is shown in Figure 3.20.

```

1  procedure atMostOneInUpperDiagonal(board, k);
2      S := { board(r)(c) : r in [1..8], c in [1..8] | r + c = k };
3      return atMostOne(S);
4  end atMostOneInUpperDiagonal;
```

Figure 3.20: The procedure `atMostOneInUpperDiagonal`.

In order to characterize the variables representing a lower diagonal, we inspect the lower main diagonal. This diagonal contains the following variables

$$p11, p22, p33, p44, p55, p66, p77, p88.$$

The indices i and j of the variable pij satisfy the linear equation

$$i - j = 0.$$

In general, the indices of a lower diagonal satisfy the linear equation

$$i - j = k,$$

where k is an element from the set $\{-6, \dots, 6\}$. This value k is the argument of the procedure `atMostOneInLowerDiagonal`, which is shown in Figure 3.21.

```

1  procedure atMostOneInLowerDiagonal(board, k);
2      S := { board(r)(c) : r in [1..8], c in [1..8] | r - c = k };
3      return atMostOne(S);
4  end atMostOneInLowerDiagonal;
```

Figure 3.21: The procedure `atMostOneInLowerDiagonal`.

Now we are able to compute a set of clauses that contains all constraints of the eight queen puzzle. Figure 3.22 shows the procedure `allClauses`. The procedure is called as

`allClauses(board)`.

Here *board* represents a chess board. The argument *board* is computed by the procedure *createBoard*. The procedure `allClauses` then computes a set of clauses that are satisfied if and only if the following conditions are true:

1. Every row contains at most one queen,
2. every lower diagonal contains at most one queen,
3. every upper diagonal contains at most one queen,
4. every column contains at least one queen.

The expressions in the procedure `allClauses` yield sets and the elements of these sets are sets of clauses. As the result needs to be one set of clauses and not a set of set of clauses we use the operator “+” to combine these sets of sets of clauses into sets of clauses.

```

1  procedure allClauses(board);
2      return  +/ { atMostOneInRow(board, row)          : row in {1..8}      }
3              + +/ { atMostOneInLowerDiagonal(board, c) : c in {-6..6}      }
4              + +/ { atMostOneInUpperDiagonal(board, c) : c in {3..15}     }
5              + +/ { oneInColumn(board, column)         : column in {1..8} };
6  end allClauses;
```

Figure 3.22: The procedure `allClauses`.

Now we are ready to solve the eight queens puzzle: The following commands compute the solution:

```

board    := createBoard(8);
Clauses  := allClauses(board);
I        := DavisPutnam(Clauses, {});
printBoard(I);
```

Here, `printBoard()` is a procedure that presents the solution in a readable form. However, this only works if we use a fixed width font. For completeness, the implementation of this procedure is shown in Figure 3.23, but we won't discuss this implementation.

```

1  procedure printBoard(I, board);
2      if I = om then
3          return;
4      end if;
5      n := #board;
6      print( "          " + ((8*n+1) * "-") );
7      for row in [1..n] loop
8          line := "          |";
9          for col in [1..n] loop
10             line += "          |";
11         end loop;
12         print(line);
13         line := "          |";
14         for col in [1..n] loop
15             if { board(row)(col) } in I then
16                 line += "      Q      |";
17             else
18                 line += "          |";
19             end if;
20         end loop;
21         print(line);
22         line := "          |";
23         for col in [1..n] loop
24             line += "          |";
25         end loop;
26         print(line);
27         print( "          " + ((8*n+1) * "-") );
28     end loop;
29 end printBoard;

```

Figure 3.23: The procedure `printBoard()`.

The set I that is computed by the call `DavisPutnam(Clauses, {})` is a set of literals. For any of the propositional variables p_{ij} , either the unit clause $\{p_{ij}\}$ or the unit clause $\{\neg p_{ij}\}$ is contained in I . If there is a queen in column i and row j , the set I will contain the clause $\{p_{ij}\}$, otherwise, I will contain the clause $\{\neg p_{ij}\}$. A graphical representation of the solution that is computed is shown in Figure 3.24.

The eight queens puzzle is only a toy application of propositional logic. Nevertheless, it shows the power of the algorithm of Davis and Putnam. The set of clauses produced by the procedure `allClauses` has a size of more than two screens. These clauses contain 64 different variables and the solution is computed in less than a second on computer running Mac OS X that was equipped with an Intel Core 2 Duo processor running at 2.15 GHz.

There are a number of practical problems that can be reduced to propositional formula in a similar way. For example, the creation of a time table satisfying a number of constraints can easily be reduced to a formula of propositional logic. In general, these problems are known as scheduling problems and have important practical applications.

:

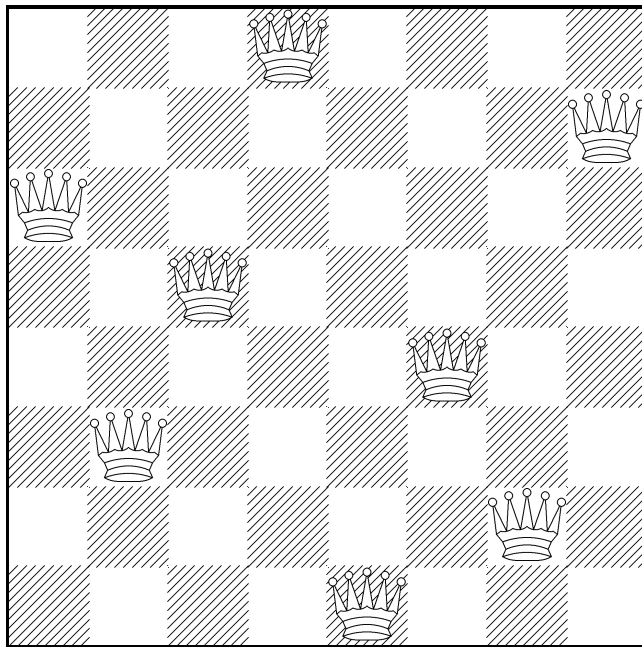


Figure 3.24: A solution of the eight queens puzzle.

Chapter 4

First-order Logic

Propositional logic discusses the propositional connectives “ \neg ”, “ \wedge ”, “ \vee ”, “ \rightarrow ”, and “ \leftrightarrow ”. In first-order logic, the set of formulæ is much richer, as we will also use the quantifiers “ \forall ” and “ \exists ”. Furthermore, the following notions are introduced:

1. *Terms* are used as a way to denote objects.
2. These terms are constructed using *variables* and *function symbols*. For example, the following are terms:

$$\text{father}(x), \quad \text{mother}(\text{isaac}), \quad x + 7.$$

Here, *father*, *mother*, *isaac*, “+”, and “7” are function symbols, while *x* is used as a variable.

3. Different objects can be put into a relation using *predicate symbols* to build atomic formulæ. For example, the following are atomic formulæ:

$$\text{isBrother}(\text{adam}, \text{father}(\text{brian})), \quad x + 7 < x \cdot 7, \quad n \in \mathbb{N}.$$

Here, we have used the predicate symbols *isBrother*, “<”, and “ \in ”.

4. First-order formulæ can be combined using propositional connectives:

$$x > 1 \rightarrow x + 7 < x \cdot 7$$

5. Furthermore, formulæ can contain *quantifiers* to define the semantics of variables:

$$\forall x \in \mathbb{R} : \exists n \in \mathbb{N} : x < n$$

The following section defines the syntax of first-order formulæ, while their interpretation is discussed in section 4.2.

4.1 Syntax

We begin with the definition of the notion of a *signature*. A signature is just a collection of symbols categorized as variables, function symbols, predicate symbols, and a specification of their arity.

Definition 36 (Signature) A signature is a 4-tuple

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

such that:

1. \mathcal{V} is the set of variables.
2. \mathcal{F} is the set of function symbols.
3. \mathcal{P} is the set of predicate symbols.
4. *arity* is a function assigning an arity to all function symbols and predicate symbols:

$$\text{arity} : \mathcal{F} \cup \mathcal{P} \rightarrow \mathbb{N}.$$

A symbol f is called an n -ary symbol iff $\text{arity}(f) = n$.

5. As we have to be able to distinguish variables, function symbols, and predicate symbols, the sets \mathcal{V} , \mathcal{F} , and \mathcal{P} have to be pairwise disjoint:

$$\mathcal{V} \cap \mathcal{F} = \{\}, \quad \mathcal{V} \cap \mathcal{P} = \{\}, \quad \text{and} \quad \mathcal{F} \cap \mathcal{P} = \{\}.$$

Terms are expressions that denote objects. They are composed from variables and function symbols. Their formal definition is as follows.

Definition 37 (Term) If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ is a signature, the set of Σ -terms \mathcal{T}_Σ is defined inductively:

1. For every variables $x \in \mathcal{V}$ we have $x \in \mathcal{T}_\Sigma$.
2. If $f \in \mathcal{F}$ is an n -ary function symbol and if $t_1, \dots, t_n \in \mathcal{T}_\Sigma$, then we also have

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

If $c \in \mathcal{F}$ is a 0-ary function symbol, we are permitted to write c instead of $c()$. In this case, the function symbol c is called a constant. \square

Example: Define the set of variables as $\mathcal{V} = \{x, y, z\}$, the set of function symbols as $\mathcal{F} = \{0, 1, +, \cdot\}$, and the set of predicate symbols as $\mathcal{P} = \{=, \leq\}$. The signature of these symbols is given by the function *arity* as follows:

$$\text{arity} = \{ \langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle +, 2 \rangle, \langle \cdot, 2 \rangle, \langle =, 2 \rangle, \langle \leq, 2 \rangle \}$$

Now, define the signature Σ_{arith} as $\Sigma_{\text{arith}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$.

Then we can construct Σ_{arith} -terms as follows:

1. $x, y, z \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
as all variables are also Σ_{arith} -terms.
2. $0, 1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
as 0 and 1 are 0-ary function symbols.
3. $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
as we have already seen that $0 \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $x \in \mathcal{T}_{\Sigma_{\text{arith}}}$ and “+” is a binary function symbol.
4. $\cdot(+(0, x), 1) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
as $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$ and “.” is a binary function symbol.

The notion of an *atomic formula* is defined next. An atomic formula is a formula that does not contain smaller formulæ. Therefore, an atomic formula cannot contain propositional connectives or quantifiers.

Definition 38 (Atomic Formulæ) If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ is a signature, the the set \mathcal{A}_Σ of all atomic Σ -formulæ is defined as follows:

If $p \in \mathcal{P}$ is an n -ary predicate symbol and $t_1, \dots, t_n \in \mathcal{T}_\Sigma$, then:

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

If p is a 0-ary predicate symbol, then we will write p instead of $p()$. In this case, the predicate symbol p is called a propositional variable. \square

Remark: Be careful not to mix up the variables of \mathcal{V} , which are also know as *object variables* and the propositional variables.

Example: If we continue the example given above, we can see that

$$=(\cdot+(0, x), 1), 0) \in \Sigma_{\text{arith}}.$$

\square

We will define first-order formulæ next. We have to be careful to distinguish between two different kinds of variables. There are so called *bound variables* and *free variables*. We introduce these notions using an example from calculus. Take a look at the following equation:

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

In this equation, the variables x and y are *free*, while the variable t is *bound* by the integral operator “ \int ”. The point is that we can substitute arbitrary values for x and y and the formula will still remain valid. For example, if we substitute 2 for x , then we get the equation

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

and this equation is true. On the other hand, substituting a number for t does not make any sense, as the integral operator needs a variable. We can try to substitute a variable t . If we substitute u for t , we would get

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

which is a valid equation. However, if we would substitute y for the variable t , we would get

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

Now this equation is no longer valid! The problem is, that by substituting y for t the variable y that already occurred inside the integral has been captured and has become a bound variable.

A similar problem arises if we substitute arbitrary terms for y . As long as these terms don't contain the variable t , we are fine. For example, if we substitute x^2 for y , we get

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

which is true. However, if we substitute t^2 for y , we get

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

and that formula is nonsense.

In first-order logic, the quantifiers “ \forall ” (*for all*) and “ \exists ” (*exists*) bind variables in a similar way as the integral operator “ \int ” does in calculus. In order to define the notions of *bound* and *free* variables precisely, we first define for a given term t the set of variables $\text{Var}(t)$ occurring in t .

Definition 39 ($\text{Var}(t)$) If t is a Σ -term, where $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$, then the set $\text{Var}(t)$ is the set of all variables occurring in t . Formally, $\text{Var}(t)$ is defined by induction on t :

1. $\text{Var}(x) := \{x\}$ for all $x \in \mathcal{V}$,
2. $\text{Var}(f(t_1, \dots, t_n)) := \text{Var}(t_1) \cup \dots \cup \text{Var}(t_n)$. □

Definition 40 (Σ -formula, bound and free variables)

Assume $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ is a signature. The set of all Σ -formulae will be denoted as \mathbb{F}_Σ . This set will be defined inductively. Furthermore, for every formula $F \in \mathbb{F}_\Sigma$ we will define the set $BV(F)$ as the set of all bound variables of F and the set $FV(F)$ will be defined as the set of all free variables of F .

1. We have $\perp \in \mathbb{F}_\Sigma$ and $\top \in \mathbb{F}_\Sigma$ and we define

$$FV(\perp) := FV(\top) := BV(\perp) := BV(\top) := \{\}$$

2. If $F = p(t_1, \dots, t_n)$ is an atomic Σ -formula, then $F \in \mathbb{F}_\Sigma$. Furthermore, we have:

- (a) $FV(p(t_1, \dots, t_n)) := \text{Var}(t_1) \cup \dots \cup \text{Var}(t_n)$.
- (b) $BV(p(t_1, \dots, t_n)) := \{\}$.

3. If $F \in \mathbb{F}_\Sigma$, then also $\neg F \in \mathbb{F}_\Sigma$. Furthermore:

- (a) $FV(\neg F) := FV(F)$.
- (b) $BV(\neg F) := BV(F)$.

4. If we have $F, G \in \mathbb{F}_\Sigma$ and if, furthermore, we have

$$FV(F) \cap BV(G) = \{\} \quad \text{and} \quad FV(G) \cap BV(F) = \{\},$$

then the following are Σ -formulae:

- (a) $(F \wedge G) \in \mathbb{F}_\Sigma$,
- (b) $(F \vee G) \in \mathbb{F}_\Sigma$,
- (c) $(F \rightarrow G) \in \mathbb{F}_\Sigma$,
- (d) $(F \leftrightarrow G) \in \mathbb{F}_\Sigma$.

Furthermore, for all sentential connectives $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ we define:

- (a) $FV(F \odot G) := FV(F) \cup FV(G)$.
- (b) $BV(F \odot G) := BV(F) \cup BV(G)$.

5. If $x \in \mathcal{V}$ and $F \in \mathbb{F}_\Sigma$ such that $x \notin BV(F)$, then we have:

- (a) $(\forall x: F) \in \mathbb{F}_\Sigma$.
- (b) $(\exists x: F) \in \mathbb{F}_\Sigma$.

Furthermore, we define:

- (a) $FV((\forall x: F)) := FV((\exists x: F)) := FV(F) \setminus \{x\}$.
- (b) $BV((\forall x: F)) := BV((\exists x: F)) := BV(F) \cup \{x\}$.

If the signature Σ is either obvious from the context or unimportant, then we will write \mathbb{F} instead of \mathbb{F}_Σ . □

In the definition given above we have been careful to guarantee that a variable cannot occur both as a free and as a bound variable in a given formula. Therefore, it can be shown that the following holds

$$FV(F) \cap BV(F) = \{\} \quad \text{for all } F \in \mathbb{F}_\Sigma.$$

Example: Continuing the example from above, we see that

$$(\exists x: \leq(+ (y, x), y)) \in \mathbb{F}_{\Sigma_{\text{arith}}}$$

holds. The set of bound variables is $\{x\}$, while the set of free variables is $\{y\}$. \square

If formulæ were always written as defined above, the readability would suffer. Therefore, we agree on certain rules that allow us to drop parenthesis. First of all, regarding the propositional connectives, we will use the same rules as discussed in the previous chapter. Furthermore, sequences of the same quantifier are combined as follows: We will write

$$\forall x, y: p(x, y) \quad \text{instead of} \quad \forall x: (\forall y: p(x, y)).$$

Furthermore, we agree that quantifiers have a higher precedence than the propositional connectives. For example,

$$(\forall x: p(x)) \wedge G \quad \text{can be written as} \quad \forall x: p(x) \wedge G.$$

Furthermore, we agree to use an infix notation for binary relation and function symbols. If the relative precedence of different binary operators is not obvious from the context, we have to use parenthesis for disambiguation. For example, the formula

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

will be written as follows:

$$\exists x: y + x \leq y.$$

In the literature, you might find expressions of the form $\forall x \in M : F$ or $\exists x \in M : F$. These are just abbreviations that are defined as follows:

$$\begin{aligned} (\forall x \in M : F) &\stackrel{\text{def}}{\iff} \forall x : (x \in M \rightarrow F), \\ (\exists x \in M : F) &\stackrel{\text{def}}{\iff} \exists x : (x \in M \wedge F). \end{aligned}$$

4.2 Semantics

Next, we fix the meaning of first-order formulæ. To this end we first define the notion of a Σ -structure a.k.a. *first-order structure*. A first-order structure gives an interpretation for the function symbols and the predicate symbols of the signature.

Definition 41 (First-order Structure) Assume a signature

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

is given. A Σ -structure \mathcal{S} is a pair $\langle \mathcal{U}, \mathcal{J} \rangle$ such that we have the following:

1. \mathcal{U} is a non-empty set. This set will be referred to as the universe of the Σ -structure. It contains all the values that can occur when we evaluate terms.
2. \mathcal{J} is an interpretation of the function symbols and predicate symbols of our signature Σ .
Formally, \mathcal{J} is defined as a mapping that has the following properties:

(a) Every function symbol $f \in \mathcal{F}$ such that $\text{arity}(f) = m$ is mapped to an m -ary function

$$f^{\mathcal{J}} : \mathcal{U} \times \cdots \times \mathcal{U} \rightarrow \mathcal{U}$$

that maps m -tuples of the universe \mathcal{U} into \mathcal{U} .

(b) Every predicate symbol $p \in \mathcal{P}$ such that $\text{arity}(p) = n$ is mapped to an n -ary relation

$$p^{\mathcal{J}} \subseteq \mathcal{U}^n.$$

(c) If the symbol “=” is an element of the predicate symbols \mathcal{P} , then we demand that the interpretation of “=” is natural, that is we must have

$$=^{\mathcal{J}} = \{ \langle u, v \rangle \mid u, v \in \mathcal{U} \wedge u = v \}.$$

Example: We provide an example of a Σ_{arith} -structure $\mathcal{S}_{\text{arith}} = \langle \mathcal{U}_{\text{arith}}, \mathcal{J}_{\text{arith}} \rangle$ by defining:

1. $\mathcal{U}_{\text{arith}} = \mathbb{N}$.
2. The interpretations $\mathcal{J}_{\text{arith}}$ are defined by requiring that the function symbols “0”, “1”, “+”, and “.” are interpreted as the corresponding functions defined on the set \mathbb{N} .

The predicate symbols \leq is defined as the following relation:

$$\leq^{\mathcal{J}_{\text{arith}}} = \{ \langle m, n \rangle \in \mathbb{N}^2 \mid \exists k \in \mathbb{N} : m + k = n \},$$

while the equality symbol is interpreted as follows:

$$=^{\mathcal{J}_{\text{arith}}} = \{ \langle m, m \rangle \mid m \in \mathbb{N} \}.$$

□

Example: The example given above is the natural way of interpreting the function and relation symbols in Σ_{arith} -structure. However, it is not the only possible σ_{arith} -structure. Let us define another Σ_{arith} -structure $\mathcal{S}_2 = \langle \mathcal{U}_2, \mathcal{J}_2 \rangle$ as follows:

1. $\mathcal{U}_2 = \{a, b\}$
2. $0^{\mathcal{J}_2} := a$
3. $1^{\mathcal{J}_2} := b$
4. $+^{\mathcal{J}_2} := \{ \langle \langle a, a \rangle, a \rangle, \langle \langle a, b \rangle, b \rangle, \langle \langle b, a \rangle, b \rangle, \langle \langle b, b \rangle, a \rangle \}$
5. $.\mathcal{J}_2 := \{ \langle \langle a, a \rangle, a \rangle, \langle \langle a, b \rangle, a \rangle, \langle \langle b, a \rangle, a \rangle, \langle \langle b, b \rangle, b \rangle \}$
6. $\leq^{\mathcal{J}_2}$ is defined as follows:
 $\leq^{\mathcal{J}_2} := \{ \langle a, a \rangle, \langle a, b \rangle, \langle b, b \rangle \}.$
7. $=^{\mathcal{J}_2}$ is the identical relation:
 $=^{\mathcal{J}_2} := \{ \langle a, a \rangle, \langle b, b \rangle \}.$

Note that we have to define $=^{\mathcal{J}_2}$ as the identical relation.

In order to evaluate terms we have to assign values to the variables first. This is done next.

Definition 42 (Variable Assignment) Assume a signature

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

and a Σ -structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ are given. An \mathcal{S} -variable assignment is a mapping

$$\mathcal{I} : \mathcal{V} \rightarrow \mathcal{U}.$$

that assigns a value from the universe \mathcal{U} to every variable.

If \mathcal{I} is an \mathcal{S} -variable assignment, x is a variable and c is an element of the universe \mathcal{U} , then we define the variable assignment $\mathcal{I}[x/c]$ as follows:

$$\mathcal{I}[x/c](y) := \begin{cases} c & \text{if } y = x; \\ \mathcal{I}(y) & \text{otherwise.} \end{cases} \quad \square$$

Definition 43 (Evaluation of Terms) If $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ is a Σ -structure and \mathcal{I} is an \mathcal{S} -variable assignment, then for every term t we can define the value of t in \mathcal{S} , which is written as $\mathcal{S}(\mathcal{I}, t)$, by induction on the structure of t :

1. If $x \in \mathcal{V}$, we define:

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

2. For a Σ -term $f(t_1, \dots, t_n)$ we define

$$\mathcal{S}(\mathcal{I}, f(t_1, \dots, t_n)) := f^{\mathcal{J}}(\mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n)). \quad \square$$

Example: Using the Σ_{arith} -structure \mathcal{S}_2 we define an \mathcal{S}_2 -variable assignment \mathcal{I} as follows:

$$\mathcal{S} := \{ \langle x, a \rangle, \langle y, b \rangle, \langle z, a \rangle \}.$$

Then we have

1. $\mathcal{I}(x) := a$,
2. $\mathcal{I}(y) := b$,
3. $\mathcal{I}(z) := a$.

Furthermore, we have

$$\mathcal{S}_2(\mathcal{I}, x + y) = b. \quad \square$$

Definition 44 (Evaluation of Atomic Σ -formulae) If \mathcal{S} is a Σ -structure and \mathcal{I} is an \mathcal{S} -variable assignment, then we can define the value of an atomic Σ -formula $p(t_1, \dots, t_n)$, written $\mathcal{S}(\mathcal{I}, p(t_1, \dots, t_n))$, as follows:

$$\mathcal{S}(\mathcal{I}, p(t_1, \dots, t_n)) := \begin{cases} \text{true} & \text{if } \langle \mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n) \rangle \in p^{\mathcal{J}}; \\ \text{false} & \text{otherwise.} \end{cases} \quad \square$$

Example: Continuing the example from above we have:

$$\mathcal{S}_2(\mathcal{I}, z \leq x + y) = \text{true}. \quad \square$$

In order to define the evaluate of arbitrary Σ -formulae we assume that we have the following functions at our disposal. These functions are defined as in the previous chapter on propositional logic.

1. $\neg: \mathbb{B} \rightarrow \mathbb{B}$,
2. $\vee: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$,
3. $\wedge: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$,
4. $\oplus: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$,
5. $\ominus: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$.

These functions have been defined in Table 3.1 on page 69.

Definition 45 (Evaluation of Σ -Formulæ) Assume \mathcal{S} is a Σ -structure and \mathcal{I} is an \mathcal{S} -variable assignment. Then, for every Σ -formula F the value $\mathcal{S}(\mathcal{I}, F)$ is defined by induction on F :

1. $\mathcal{S}(\mathcal{I}, \top) := \text{true}$ and $\mathcal{S}(\mathcal{I}, \perp) := \text{false}$.
2. $\mathcal{S}(\mathcal{I}, \neg F) := \neg(\mathcal{S}(\mathcal{I}, F))$.
3. $\mathcal{S}(\mathcal{I}, F \wedge G) := \bigwedge(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
4. $\mathcal{S}(\mathcal{I}, F \vee G) := \bigvee(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
5. $\mathcal{S}(\mathcal{I}, F \rightarrow G) := \neg(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
6. $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) := \neg(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
7. $\mathcal{S}(\mathcal{I}, \forall x: F) := \begin{cases} \text{true} & \text{if } \mathcal{S}(\mathcal{I}[x/c], F) = \text{true} \text{ for all } c \in \mathcal{U}; \\ \text{false} & \text{otherwise.} \end{cases}$
8. $\mathcal{S}(\mathcal{I}, \exists x: F) := \begin{cases} \text{true} & \text{if } \mathcal{S}(\mathcal{I}[x/c], F) = \text{true} \text{ for some } c \in \mathcal{U}; \\ \text{false} & \text{otherwise.} \end{cases} \quad \square$

Example: Continuing the last example, we have

$$\mathcal{S}_2(\mathcal{I}, \forall x: x \cdot 0 \leq 1) = \text{true}.$$

Definition 46 (Universally Valid) If F is a Σ -formula such that we have

$$\mathcal{S}(\mathcal{I}, F) = \text{true}$$

for every Σ -formula \mathcal{S} and any \mathcal{S} -variable assignment \mathcal{I} , then the formula F is called *universally valid*. This is written as

$$\models F. \quad \square$$

If F is a formula such that $FV(F) = \{\}$, then the value of $\mathcal{S}(\mathcal{I}, F)$ does not depend on the variable assignment \mathcal{I} . A formula F such that $FV(F) = \{\}$ is called a *closed* formula. If F is a closed formula we write $\mathcal{S}(F)$ instead of $\mathcal{S}(\mathcal{I}, F)$. If, furthermore, $\mathcal{S}(F) = \text{true}$, then the structure \mathcal{S} is called a *model* for F . In this case, we will write

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

The notion “*satisfiable*” and “*equivalent*” can be transferred from propositional logic to first-order logic. In order to keep our definitions readable, we will assume in the following that a fixed signature Σ is given. Then, we can just talk of terms, formulæ, and structures instead of Σ -terms, Σ -formulæ, and Σ -structures.

Definition 47 (Equivalent) Two formulæ F and G are called *equivalent* iff

$$\models F \leftrightarrow G \quad \square$$

Remark: All propositional equivalences are first-order equivalences, too.

Definition 48 (Satisfiable) A set of formulæ $M \subseteq \mathbb{F}_\Sigma$ is *satisfiable*, iff there is a structure \mathcal{S} and a variable assignment \mathcal{I} such that

$$\forall m \in M : \mathcal{S}(\mathcal{I}, m) = \text{true}$$

holds. Otherwise, M is called *unsatisfiable*. This is written as

$$M \models \perp. \quad \square$$

Our goal is to develop an algorithm that takes a given set of M of first order formulæ and that checks, whether the set M is unsatisfiable. We will see that the question whether

$$M \models \perp$$

holds, is undecidable in general. However, we will be able to develop a *calculus* \vdash such that we have

$$M \vdash \perp \quad \text{iff} \quad M \models \perp.$$

Therefore, we will be able to write a program that can be used as a so called *semi-decision procedure* for the question, whether $M \vdash \perp$ holds. By this we mean the following: If indeed M is unsatisfiable, then our program will eventually be able to discover that fact. However, if M is satisfiable, then the program might run forever. The program will be based on the calculus \vdash . The idea is to generate all possible derivations of \perp from M in a systematic way. If there is a proof of \perp , we will eventually find it and thus have shown M to be unsatisfiable. On the other hand, if there is no such proof, the program might run forever.

In order to facilitate our understanding of the notion of semi-decidability, we present a problem from number theory that exhibits a similar behaviour. A natural number n is called *perfect* iff the sum of all of its proper factors is the number n . For example, the number n is perfect as the proper factors are 1, 2, and 3 and we have

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

All known perfect numbers are even. The question, whether there is an odd perfect number is an open mathematical problem. Let us try to solve the problem by writing a procedure that checks for every odd number whether it is perfect. Figure 4.1 on page 113 shows this program. Now if there is a perfect number that is odd, and if, furthermore, we have unlimited computational resources, the program will eventually find it. However, if there are no odd numbers that are perfect, our program will never produce a result. Therefore, this program does not decide the question. Rather it is only a semi-decision procedure¹.

```

1  program main;
2      n := 1;
3      loop
4          if perfect(n) then
5              if n mod 2 = 0 then
6                  print(n);
7              else
8                  print("Heureka: ", n);
9              end if;
10         end if;
11         n := n + 1;
12     end loop;
13
14     procedure perfect(n);
15         return +/ { x in 1 .. n-1 | n mod x = 0 } = n;
16     end perfect;
17 end main;

```

Figure 4.1: Searching for an odd perfect number.

¹Before wasting countless cpu cycles running this program you should know that it has been proven that all odd perfect number have to be bigger than 10^{300} .

4.2.1 Implementing First-order Structures in Setl2

The notion of first-order structures developed in the last section is an abstract one. In order to facilitate our comprehension of this concept we will give an implementation of first-order structures in SETL2. As an concrete example we discuss *group theory*. The signature Σ_G of group theory is given as follows:

$$\Sigma_G = \langle \{x, y, z\}, \{1, *\}, \{=\}, \{\langle 1, 0 \rangle, \langle *, 2 \rangle, \langle =, 2 \rangle\} \rangle.$$

We use “1” as a nullary function symbol, “*” is a binary function symbol and “=” is a binary predicate symbol. We define a Σ_G -structure $\mathcal{S}_2 = \langle \mathcal{U}_2, \mathcal{J}_2 \rangle$ as follows:

1. $\mathcal{U}_2 = \{a, b\}$
2. $1^{\mathcal{J}_2} := a$
3. $*^{\mathcal{J}_2} := \{ \langle \langle a, a \rangle, a \rangle, \langle \langle a, b \rangle, b \rangle, \langle \langle b, a \rangle, b \rangle, \langle \langle b, b \rangle, a \rangle \}$
4. $=^{\mathcal{J}_2}$ is the identical relation:
 $=^{\mathcal{J}_2} := \{ \langle a, a \rangle, \langle b, b \rangle \}.$

In SETL2, this structure can be implemented as shown in figure 4.2.

```

1  a := "a";
2  b := "b";
3  U := { a, b }; -- the universe
4  multiply := { [ [a,a], a ], [ [a,b], b ], [ [b,a], b ], [ [b,b], a ] };
5  equal := { [ x, y ] : x in U, y in U | x = y };
6  J := { [ "1", a ], [ "*", multiply ], [ "=", equal ] };
7  S := [ U, J ];
8  I := { [ "x", a ], [ "y", b ], [ "z", a ] };

```

Figure 4.2: A First-order Structure for Group-Theory.

1. We have defined the variables a and b as the strings "a" and "b" in line 1 and 2. These variables are really nothing more than abbreviations that enable us to give the interpretation of the function symbol “*” in a concise form.
2. The universe \mathcal{U} consists of the two strings "a" and "b".
3. The interpretation of “*” is given by the binary relation `multiply` defined in line 4. The function defined by this relation satisfies the following:
$$\begin{aligned} \text{multiply}(\langle \text{"a"}, \text{"a"} \rangle) &= \text{"a"}, & \text{multiply}(\langle \text{"a"}, \text{"b"} \rangle) &= \text{"b"}, \\ \text{multiply}(\langle \text{"b"}, \text{"a"} \rangle) &= \text{"b"}, & \text{multiply}(\langle \text{"b"}, \text{"b"} \rangle) &= \text{"a"}. \end{aligned}$$
4. Line 5 defines the relation `equal` that is used as the interpretation $=^{\mathcal{J}}$ of the binary relation symbol “=”.
5. Line 6 collects the different interpretations and assigns them to their respective symbol. Therefore, for as given function symbol f , the interpretation of f , i.e. $f^{\mathcal{J}}$, is given as $J(f)$.
6. Line 7 combines the universe and U and the interpretation J to a first-order structure S .
7. Line 8 defines a variable assignment for the variables "x", "y", and "z".

```

1  procedure evalFormula(f, S, I);
2      case
3          when f = 1          => return TRUE;
4          when f = 0          => return FALSE;
5          when f(1) = "-"     => return not evalFormula(f(2), S, I);
6          when f(2) = "*"     =>
7              return evalFormula(f(1), S, I) and evalFormula(f(3), S, I);
8          when f(2) = "+"     =>
9              return evalFormula(f(1), S, I) or evalFormula(f(3), S, I);
10         when f(2) = "->"    =>
11             return not evalFormula(f(1), S, I) or evalFormula(f(3), S, I);
12         when f(2) = "<->"    =>
13             return evalFormula(f(1), S, I) = evalFormula(f(3), S, I);
14         when f(1) = "forall" =>
15             x := f(2);
16             g := f(3);
17             U := S(1);
18             return forall c in U | evalFormula(g, S, modify(I, x, c));
19         when f(1) = "exists" =>
20             x := f(2);
21             g := f(3);
22             U := S(1);
23             return exists c in U | evalFormula(g, S, modify(I, x, c));
24         otherwise => return evalAtomic(f, S, I); -- atomic formula
25     end case;
26 end evalFormula;

```

Figure 4.3: Evaluation of First-order Formulæ.

Next, we develop a procedure that is capable of evaluating first-order formulæ in a given structure. Figure 4.3 shows the implementation of the procedure $evalFormula(f, S, \mathcal{I})$. The arguments of this procedure are as follows:

1. f is a first-order formula,
2. S is a first-order structure, and
3. \mathcal{I} is a variable assignment.

Here, the formula f is represented as a list in a similar way as we have done it already in propositional logic. For example, the formula

$$\forall x : \forall y : x * y = y * x$$

is represented as

`["forall", "x", ["forall", "y", ["=", ["*", "x", "y"], ["*", "y", "x"]]]]`.

The evaluation of first-order formulæ is similar to the evaluation of propositional formulæ shown in Figure 3.1 on page 71. Only the treatment of quantifiers is new here. Line 10 to 14 deals with the evaluation of a formula f of the form $\forall x: g$. This formula is represented as the list

$$f = [\text{forall}, x, g].$$

The second element of this list is the variable x . Therefore we have $x = f(2)$. The third component is the formula g , we have $g = f(3)$. The evaluation of $\forall x: g$ is done according to the definition

$$\mathcal{S}(\mathcal{I}, \forall x: g) := \begin{cases} \text{true} & \text{if } \mathcal{S}(\mathcal{I}[x/c], g) = \text{true} \text{ for all } c \in \mathcal{U}; \\ \text{false} & \text{otherwise.} \end{cases}$$

In order to implement this we use the procedure *modify()* which changes the variable assignment \mathcal{I} for the variable x into c , we have

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

The implementation of this procedure is shown later in Figure 4.4.

The evaluation of a formula of the form $\exists x: g$ is similar and will therefore not be discussed in detail.

```

1  procedure evalAtomic(a, S, I);
2      -- we do not support nullary predicates yet
3      J      := S(2);
4      p      := a(1); -- predicate symbol
5      pJ     := J(p);
6      args   := a(2..);
7      argsVal := evalTermList(args, S, I);
8      return argsVal in pJ;
9  end evalAtomic;
10
11 procedure evalTerm(t, S, I);
12     if is_string(t) then -- it is a variable
13         return I(t);
14     end if;
15     J      := S(2);
16     f      := t(1); -- function symbol
17     fJ     := J(f);
18     args   := t(2..);
19     argsVal := evalTermList(args, S, I);
20     if #argsVal > 0 then
21         result := fJ(argsVal);
22     else
23         result := fJ; -- t is a constant
24     end if;
25     return result;
26 end evalTerm;
27
28 procedure evalTermList(tl, S, I);
29     return [ evalTerm(t, S, I) : t in tl ];
30 end evalTermList;
31
32 procedure modify(I, x, c);
33     I(x) := c;
34     return I;
35 end modify;

```

Figure 4.4: Evaluation of Terms.

Figure 4.4 shows the evaluation of atomic formulæ and of first-order terms. An atomic formula of the form

$$p(t_1, \dots, t_n)$$

is represented as the list

$$[p, t_1, \dots, t_n].$$

Therefore, the predicate symbol p is the first element of the list representing an atomic formula and the arguments t_1, \dots, t_n make up the rest of the list. We evaluate the arguments using the procedure `evalTermList()` and then we only have to check, whether the resulting list of elements is a member of the interpretation of the predicate symbol p .

Next, we discuss the procedure `evalTerm()`. The first argument t of this procedure is a term that needs to be evaluated. The second argument \mathcal{S} of the procedure `evalAtomic()` is a first-order structure and the final argument I is a variable assignment.

1. If t is a variable it will just be a string. Therefore, we just have to look up the value assigned to x in the variable assignment \mathcal{I} . This variable assignment is represented as a binary relation that can be used as a function in SETL2.
2. If the term t has the form

$$t = f(t_1, \dots, t_n),$$

then t will be represented as a list of the form

$$[f, t_1, \dots, t_n].$$

In order to evaluate this expression, we first evaluate the subterms recursively. After that, we use the interpretation $f^{\mathcal{I}}$ the function symbol f to evaluate the function for the given arguments. We have to be careful to also handle the case where the list of arguments is empty.

The implementation of the procedure `evalTermList()` applies the function `evalTerm()` on all terms of a given list.

Finally, the procedure `modify(I, x, c)` computes a new variable assignment. The implementation is very straightforward, as SETL2 provides the means to change a functional relation using an assignment of the following form:

$$\mathcal{I}(x) := c;$$

```

1  g1 := parse("forall x: =(*(x, 1), x)");
2  g2 := parse("forall x: exists y: =(*(x, y), 1)");
3  g3 := parse("forall x: forall y: forall z: =( *(*(x,y), z), *(x, *(y,z)) )");
4  GT := { g1, g2, g3 };
5  for f in GT loop
6      print( "checking ", f, ": ", evalFormula(f, S, I) );
7  end loop;
```

Figure 4.5: Axioms of Group Theory.

We conclude this section by showing how the function `evalFormula(f, S, I)` can be used to check, whether the structure shown in Figure 4.2 validates the axioms of *group theory*. These axioms are as follows:

1. The constant 1 is the right-neutral element of multiplication:

$$\forall x: x * 1 = x.$$

2. For every element x there is a right-inverse element y , such that x multiplied by y yields 1:

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

3. The law of associativity holds true in any structure that is a group:

$$\forall x: \forall y: \forall z: (x * y) * z = x * (y * z).$$

These axioms are given in line 1 to 3 of figure 4.5. I could not use infix operators here, as my parser did not support them. The loop in line 4 to 6 checks, whether all axioms are satisfied in the structure defined above.

Remark: The program discussed in this section can be used to check, whether a given first-order formula is valid in a given first-order structure. However, we cannot check whether a formula is universally valid for two reasons:

1. For one thing, we cannot use the program if the universe has an infinite number of elements. There are certain formula that evaluate as true in all finite first-order structures but that are not satisfied in certain first-order structures with an infinite universe.
2. The number of possible first-order structures is infinite, even if we restrict ourselves to finite universes. Therefore, we cannot check all universes.

4.3 Normal Forms for First-order Formulæ

In the next sections, we will define a calculus \vdash for first-order formulæ. Our task is simplified a lot if we restrict ourselves to formulæ that have a special form. Therefore, we will restrict our attention to so called first-order clauses. Note that this approach is similar to what we did in propositional logic: There, we first transformed all formulæ into conjunctive normal form. Then we used the Davis-Putnam algorithm to check whether these formulæ were satisfiable. In first-order logic, we will see that every first-order formula F can be transformed into a set of first order clauses that is satisfiable iff the original formula F is satisfiable. In order to transform a first-order formula into a first-order clause we will make use of the following equivalences.

Proposition 49 *The following equivalences are valid:*

1. $\models \neg(\forall x: f) \leftrightarrow (\exists x: \neg f)$
2. $\models \neg(\exists x: f) \leftrightarrow (\forall x: \neg f)$
3. $\models (\forall x: f) \wedge (\forall x: g) \leftrightarrow \forall x: (f \wedge g)$
4. $\models (\exists x: f) \vee (\exists x: g) \leftrightarrow \exists x: (f \vee g)$
5. $\models (\forall x: \forall y: f) \leftrightarrow (\forall y: \forall x: f)$
6. $\models (\exists x: \exists y: f) \leftrightarrow (\exists y: \exists x: f)$
7. *If x is a variable such that $x \notin FV(f)$, then we have*

$$\models (\forall x: f) \leftrightarrow f \quad \text{and} \quad \models (\exists x: f) \leftrightarrow f.$$
8. *If x is a variable such that $x \notin FV(g) \cup BV(g)$, then we have the following:*
 - (a) $\models (\forall x: f) \vee g \leftrightarrow \forall x: (f \vee g)$
 - (b) $\models g \vee (\forall x: f) \leftrightarrow \forall x: (g \vee f)$
 - (c) $\models (\exists x: f) \wedge g \leftrightarrow \exists x: (f \wedge g)$

$$(d) \models g \wedge (\exists x: f) \leftrightarrow \exists x: (g \wedge f)$$

In order to make use of the equivalences of the last group it might be necessary to rename bound variables. If f is a first-order formula and if x and y are two variables, then $f[x/y]$ is the formula that we get when we replace every occurrence of x in f by y . For example, we have

$$(\forall u : \exists v : p(u, v))[u/z] = \forall z : \exists v : p(z, v)$$

Now we are ready to formulate the last equivalence: If f is a first-order formula such that $x \in BV(f)$ and if y is a variable that does not occur in f , then we have

$$\models f \leftrightarrow f[x/y].$$

This just means that we can rename bound variables as long as the new names do not clash with other variable names.

Using the equivalences given so far, any first-order formula can be rewritten in a way that all quantifiers occur at the beginning of the formula. A formula that can be written as a string of quantifiers followed by a subformula without quantifiers is said to be in *prenex normal form*. An example will clarify the algorithm that is used to transform a formula into prenex normal form. Let us transform the formula

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

into prenex normal form:

$$\begin{aligned} & \forall x: p(x) \rightarrow \exists x: p(x) \\ \leftrightarrow & \neg(\forall x: p(x)) \vee \exists x: p(x) \\ \leftrightarrow & \exists x: \neg p(x) \vee \exists x: p(x) \\ \leftrightarrow & \exists x: (\neg p(x) \vee p(x)) \\ \leftrightarrow & \exists x: \top \\ \leftrightarrow & \top \end{aligned}$$

Here, we got lucky and ended up with a formula that does not contain any quantifier at all. Let us consider another example:

$$\begin{aligned} & \exists x: p(x) \rightarrow \forall x: p(x) \\ \leftrightarrow & \neg(\exists x: p(x)) \vee \forall x: p(x) \\ \leftrightarrow & \forall x: \neg p(x) \vee \forall x: p(x) \\ \leftrightarrow & \forall x: \neg p(x) \vee \forall y: p(y) \\ \leftrightarrow & \forall x: (\neg p(x) \vee \forall y: p(y)) \\ \leftrightarrow & \forall x: \forall y: (\neg p(x) \vee p(y)) \end{aligned}$$

This formula is now in prenex normal form.

In order to normalize first-order formulæ even more we need a stronger notion of equivalence. This is the notion of *equisatisfiability*. Let us motivate this notion by an example. Let us compare the following two first-order formulæ :

$$f_1 = \forall x: \exists y: p(x, y) \quad \text{and} \quad f_2 = \forall x: p(x, s(x)).$$

The formulæ f_1 and f_2 are not equivalent, they don't even use the same signature: The formula f_2 uses the function symbol s which does not occur in the formula f_1 . However, even although f_1 and f_2 are not equivalent, they still are related in the following sense: If S_1 is a first-order structure such that f_1 is true in S_1

$$S_1 \models f_1,$$

then the structure S_1 can be extended to a structure S_2 such that f_2 is valid in S_2 :

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

In order to do this, we just have to define the interpretation of the function symbol s in a way such that we do have

$$p(x, s(x))$$

for any x from our universe. This is possible, as the formula f_1 states that given any x we can find a value y such that $p(x, y)$ is true. The function s therefore just has to return a y such that $p(x, y)$ is true. Therefore, f_1 and f_2 are *equisatisfiable*. In general, two formulæ f_1 and f_2 are equisatisfiable if f_1 has a model if and only if f_2 has a model.

Definition 50 (Skolemization) Assume $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ is a signature and f is a closed Σ -formula of the form

$$f = \forall x_1, \dots, x_n: \exists y: g(x_1, \dots, x_n, y).$$

Then we choose a new n -ary function symbol s , i.e. we take a symbol s that does not yet occur in the set \mathcal{F} and we extend the signature Σ to the new signature

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

that has s as a new n -ary function symbol. Then, we define the Σ' -formula f' as follows:

$$f' := \text{Skolem}(f) := \forall x_1: \dots \forall x_n: g(x_1, \dots, x_n, s(x_1, \dots, x_n)).$$

As you can see, the existential quantifier $\exists y$ has been dropped. Furthermore, every occurrence of the variable y is replaced by the term $s(x_1, \dots, x_n)$. We say that the formula f' is generated from f through a *skolemization step*. \square

In order to describe how a formula f and the skolemization of f , i.e. $\text{Skolem}(f)$ are related, we need the following definition.

Definition 51 (Equisatisfiability) Two closed formula f and g are equisatisfiable if either f and g are both satisfiable or both are unsatisfiable. If f and g are equisatisfiable, then this is written as

$$f \approx_e g.$$

\square

Proposition 52 If the formula f' is generated from the formula f through a skolemization step, then f and f' are equisatisfiable.

Now we can provide a simple algorithm to eliminate the existential quantifiers from a given formula: First, the formula is put into prenex normal form. Second, all existential quantifiers are eliminated by skolemization steps. This step is called *skolemization*. According to the last proposition, the resulting formulæ are equisatisfiable. If we transform a formula F into prenex normal and then skolemize the prenex normal form, we end up with a formula of the form

$$f' = \forall x_1, \dots, \forall x_n: g$$

such that the formula g does not contain any quantifiers. The formula g is sometimes called the *matrix* of f' . As the formula g only contains propositional connectives, it can be transformed into conjunctive normal form using the algorithm shown in the previous chapter. Then we end up with a formula of the form

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

Here, the k_i are disjunctions of *literals*. (In first-order logic, a literal is either an atomic formula or the negation of an atomic formula.) If we apply the equivalence $(\forall x: f_1 \wedge f_2) \leftrightarrow (\forall x: f_1) \wedge (\forall x: f_2)$ then the universal quantifiers can be distributed onto the different k_i and the resulting formula

has the form

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

A formula f of this form is said to be in *first-order clausal-normal-form* and a formula of the form

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

where k is a disjunction of literals is called a *first-order clause*. If M is a set of formulæ and we want to know whether M is satisfiable, then we can transform M into a set of first-order clauses that is equisatisfiable. As a first-order clause only contains universal quantifiers, we can simplify our notation even further by agreeing that all formula are implicitly universally quantified. Then we can drop these quantifiers.

Now what is this all good for? The idea is to develop an algorithm that is able to check whether a given first-order formula f is universally valid or not, that is want to know whether

$$\models f$$

holds. We know that

$$\models f \quad \text{iff} \quad \{\neg f\} \models \perp$$

holds, as the formula f is universally valid iff there is no structure that satisfies the formula $\neg f$. We therefore form $\neg f$ and transform $\neg f$ into first-order clausal-normal form. Then we get something like

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

where k_1, \dots, k_n are first-order clauses. Next, we try to derive an inconsistency from the first-order clauses k_1, \dots, k_n :

$$\{k_1, \dots, k_n\} \vdash \perp$$

If we succeed, then we know that the set $\{k_1, \dots, k_n\}$ is unsatisfiable. That implies that $\neg f$ is unsatisfiable and therefore, in this case, we have shown that f is universally valid. In order to be able to derive an inconsistency from the clauses k_1, \dots, k_n we still need inference rules for first-order clauses. We will demonstrate a set of appropriate inference rules at the end of this chapter.

Let us demonstrate the algorithm sketched so far by an example. We want to investigate, whether

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

holds. We know that this holds if and only if we have

$$\left\{ \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \right\} \models \perp.$$

Let us first transform the negated formula into prenex normal form:

$$\begin{aligned} & \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \\ \leftrightarrow & \neg \left(\neg (\exists x: \forall y: p(x, y)) \vee (\forall y: \exists x: p(x, y)) \right) \\ \leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge \neg (\forall y: \exists x: p(x, y)) \\ \leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists y: \neg \exists x: p(x, y)) \\ \leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists y: \forall x: \neg p(x, y)) \end{aligned}$$

In order to proceed, we have to rename the bound variables in the second part of the conjunction. We rename x as u and y as v and get the following:

$$\begin{aligned}
& (\exists x: \forall y: p(x, y)) \wedge (\exists y: \forall x: \neg p(x, y)) \\
\leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists v: \forall u: \neg p(u, v)) \\
\leftrightarrow & \exists v: \left((\exists x: \forall y: p(x, y)) \wedge (\forall u: \neg p(u, v)) \right) \\
\leftrightarrow & \exists v: \exists x: \left((\forall y: p(x, y)) \wedge (\forall u: \neg p(u, v)) \right) \\
\leftrightarrow & \exists v: \exists x: \forall y: \left(p(x, y) \wedge (\forall u: \neg p(u, v)) \right) \\
\leftrightarrow & \exists v: \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, v) \right)
\end{aligned}$$

Now we have to skolemize in order to get rid of the existential quantifiers. In order to do so we introduce two new function symbols s_1 and s_2 . Because there are no universal quantifiers in front of the existential quantifiers, we have $\text{arity}(s_1) = 0$ and $\text{arity}(s_2) = 0$.

$$\begin{aligned}
& \exists v: \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, v) \right) \\
\approx_e & \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, s_1) \right) \\
\approx_e & \forall y: \forall u: \left(p(s_2, y) \wedge \neg p(u, s_1) \right)
\end{aligned}$$

The last formula only contains universal quantifiers which can be dropped, as we have agreed that in a first-order clause all variables are implicitly universally quantified. Then the first-order clausal normal form of this formula in set notation is

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

Let us now show that the set M is inconsistent. First, take the clause $\{p(s_2, y)\}$ and substitute the constant s_1 for the variable y . That yields the clause

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

We can substitute s_1 for y , as the clause is implicitly universally quantified.

Next, we take the clause $\{\neg p(u, s_1)\}$ and substitute s_2 for u , yielding the clause

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

Applying the cut rule to the clauses (1) and (2) gives

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

Therefore, we have derived an inconsistency and this shows that M is unsatisfiable. Therefore, the set

$$\left\{ \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \right\}$$

is unsatisfiable, too, and we have shown

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

4.4 Unification

In the example in the last section we have guessed terms s_1 and s_2 that we then substituted for the variables y and u in the clauses $\{p(s_2, y)\}$ and $\{\neg p(u, s_1)\}$. These terms were chosen with the intention to apply the cut rule later. In this section we develop a method to calculate the terms that are required to be able to apply the cut rule. In order to do so, we first introduce the notion of a *variable substitution*.

Definition 53 (Substitution) Let a signature

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

be given. A Σ -Substitution is a finite set of pairs of the form

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

such that we have the following:

1. $x_i \in \mathcal{V}$ for all $i \in \{1, \dots, n\}$,
2. $t_i \in \mathcal{T}_\Sigma$ for all $i \in \{1, \dots, n\}$,
3. if $i \neq j$, then $x_i \neq x_j$, therefore, the variables x_i are all different from each other.

If $\sigma = \{ \langle x_1, t_1 \rangle, \dots, \langle x_n, t_n \rangle \}$ is a Σ -substitution, we write this as

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

Furthermore, the *domain* of the substitution σ is defined as

$$\text{dom}(\sigma) = \{x_1, \dots, x_n\}.$$

The set of all substitutions is denoted as *Subst*. □

We will later *apply* a substitution to a term. If t is a term and σ is a substitution, then $t\sigma$ is the term that we get when we replace every variable x_i in the term t by the corresponding term t_i . The formal definition follows.

Definition 54 (Application of a Substitution)

Take a term t and a substitution $\sigma = [x_1 \mapsto t_1, \dots, x_n \mapsto t_n]$. We define the *application* of σ on t (written $t\sigma$) by induction on t :

1. If t is a variable, there are two cases:
 - (a) $t = x_i$ for an $i \in \{1, \dots, n\}$. Then we define
$$x_i\sigma := t_i.$$
 - (b) $t = y$ and $y \in \mathcal{V}$, but $y \notin \{x_1, \dots, x_n\}$. We define
$$y\sigma := y.$$

2. If t is not a variable, it must have the form $t = f(s_1, \dots, s_m)$. Then we define

$$f(s_1, \dots, s_m)\sigma := f(s_1\sigma, \dots, s_m\sigma),$$

as the expressions $s_i\sigma$ are defined by induction hypotheses. □

A substitution can also be applied to a first-order clause if predicate symbols and propositional connectives are treated similar to function symbols. Instead of giving a lengthy formal definition, we provide some examples. Let us define the substitution σ as

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

Then we have the following:

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

Next, we show how substitutions can be combined.

Definition 55 (Composition of Substitutions) Assume that

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

are two substitutions such that $\text{dom}(\sigma) \cap \text{dom}(\tau) = \{\}$. Then the *composition* $\sigma\tau$ of σ and τ is defined as follows:

$$\sigma\tau := [x_1 \mapsto s_1\tau, \dots, x_m \mapsto s_m\tau, y_1 \mapsto t_1, \dots, y_n \mapsto t_n] \quad \square$$

Example: Continuing the last example we define

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

Then we have:

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

We have the following proposition.

Proposition 56 If t is a Term and σ and τ are substitutions such that $\text{dom}(\sigma) \cap \text{dom}(\tau) = \{\}$, then we have the following:

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

This proposition can be proven by induction on the term t .

Definition 57 (Syntactical Equation) A syntactical equation is a construct of the form $s \doteq t$, where either of the following is true:

1. s and t are both terms, or
2. s and t are both atomic formula.

Furthermore, a syntactical system of equations is a set of syntactical equations. \square

When discussing syntactical equations, we will not distinguish between function symbols and relation symbols.

Definition 58 (Unifier) A substitution σ solves a syntactical equation $s \doteq t$ iff we have

$$s\sigma = t\sigma,$$

that is applying the substitution σ to both s and t yields identical objects. If E is a syntactical system of equations, then a substitution σ is called a *unifier* of E iff σ solves every syntactical equation in E . \square

If $E = \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}$ is a syntactical system of equations and σ is a substitution, then we define the application of σ on E as follows:

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

Example: Consider the following syntactical equation:

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

Define the substitution σ as

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

The substitution σ solves the syntactical equation given above, as we have

$$p(x_1, f(x_4))\sigma = p(x_2, f(x_4)) \quad \text{and}$$

$$p(x_2, x_3)\sigma = p(x_2, f(x_4))$$

and these atomic formulæ are identical. □

Next, we develop an algorithm that takes a system of syntactical equations E and calculates a unifier σ for E , provided there is one. If the system E is not solvable, the algorithm recognizes this fact. To begin, let us think of those syntactical equations, which are obviously unsolvable. There are two cases: A syntactical equation of the form

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

is obviously unsolvable if f and g are different function symbols. The reason is, that we have

$$f(s_1, \dots, s_m)\sigma = f(s_1\sigma, \dots, s_m\sigma) \quad \text{and} \quad g(t_1, \dots, t_n)\sigma = g(t_1\sigma, \dots, t_n\sigma)$$

for any substitution σ . Therefore, if $f \neq g$, then the terms $f(s_1, \dots, s_m)\sigma$ and $g(t_1, \dots, t_n)\sigma$ start with different function symbols and can therefore not be syntactical equal.

But there is another syntactical equation that is obviously unsolvable. Consider the equation

$$x \doteq f(t_1, \dots, t_n) \quad \text{where } x \in \text{Var}(f(t_1, \dots, t_n)).$$

Regardless what term we substitute for x , the right hand side of this equation will always have at least one more function symbol than the left hand side. Therefore, the right hand side can never be equal to the left hand side. This is best seen by inspecting the special case

$$x \doteq s(x).$$

We are now ready to give an algorithm that can be used to solve a system of syntactical equations. This algorithm works on pairs of the form

$$\langle F, \tau \rangle.$$

Here F is a system of syntactical equations and τ is a substitution. The algorithm is started with the pair $\langle E, [] \rangle$. Here E is the set of syntactical equations that have to be solved, while $[]$ is the empty substitution, i.e. the substitution that does not change anything. The algorithm works by applying the reductions rules given below. These rules are applied as long as possible. Then, the pair $\langle E, [] \rangle$ is either reduced to a pair of the form $\langle \{\}, \sigma \rangle$ where the substitution σ is a unifier of E , or we derive an equation that is obviously unsolvable. The reduction rules are as follows:

1. If $y \in \mathcal{V}$ is a variable that does not occur in the term t , then we apply the following reduction:

$$\langle E \cup \{y \doteq t\}, \sigma \rangle \rightsquigarrow \langle E[y \mapsto t], \sigma[y \mapsto t] \rangle \quad \text{provided } y \notin \text{Var}(t).$$

This reduction rule can be interpreted as follows: If the set of syntactical equations contains an equation of the form $y \doteq t$ where y is a variable, and if, furthermore, the variable y does not occur in t , then the equation $y \doteq t$ is solved by the substitution $[y \mapsto t]$. Therefore, we can drop the syntactical equation $y \doteq t$ in favor of the substitution $[y \mapsto t]$. However, we have to take care to apply this substitution to the remaining equations and also to the substitution σ .

2. If the variable y does occur in the term t , that is if $y \in \text{Var}(t)$, and if, furthermore, $t \neq y$, then the system of syntactical equations $E \cup \{y \doteq t\}$ is not solvable. This will be written as

$$\langle E \cup \{y \doteq t\}, \sigma \rangle \rightsquigarrow \Omega \quad \text{provided } y \in \text{Var}(t) \text{ and } t \neq y.$$

3. If $y \in \mathcal{V}$ is a variable and t isn't a variable, then we have:

$$\langle E \cup \{t \doteq y\}, \sigma \rangle \rightsquigarrow \langle E \cup \{y \doteq t\}, \sigma \rangle \quad \text{provided } y \in \mathcal{V} \text{ and } t \neq \mathcal{V}.$$

This rule is necessary to be able to apply one of the first two rules later.

4. Trivial syntactical equations can be dropped:

$$\langle E \cup \{x \doteq x\}, \sigma \rangle \rightsquigarrow \langle E, \sigma \rangle.$$

5. If f is an n -ary function symbol, then we have

$$\langle E \cup \{f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)\}, \sigma \rangle \rightsquigarrow \langle E \cup \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}, \sigma \rangle.$$

A syntactical equation of the form $f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)$ is therefore replaced by the n syntactical equations $s_1 \doteq t_1, \dots, s_n \doteq t_n$.

This rule is the reason that we have to work with sets of equations. Even if we start with just one syntactical equation, applying this rule can increase the number of equations.

A special case of this rule is

$$\langle E \cup \{c \doteq c\}, \sigma \rangle \rightsquigarrow \langle E, \sigma \rangle.$$

Here c is a nullary function symbol.

6. The set of syntactical equations $E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}$ does not have a solution if the function symbols f and g are different:

$$\langle E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}, \sigma \rangle \rightsquigarrow \Omega \quad \text{provided } f \neq g.$$

If we have been given a set of syntactical equations E , we start with the pair $\langle E, [] \rangle$. We will always be able to apply one of the reductions given above until one of the following cases occurs:

1. We can apply either the 2nd or the 6th rule. In this case, the system of syntactical equations is not solvable.
2. The pair $\langle E, [] \rangle$ is successively reduced to the pair $\langle \{\}, \sigma \rangle$. In this case, σ is a unifier of E . This is written as $\sigma = \text{mgu}(E)$. If we start with the set of equations $E = \{s \doteq t\}$, then we will write this as $\sigma = \text{mgu}(s, t)$. The abbreviation *mgu* is short for “*most general unifier*”.

Example: Let us demonstrate the algorithm and solve the syntactical equation

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

The solution is calculated as follows:

$$\begin{aligned} & \langle \{p(x_1, f(x_4)) \doteq p(x_2, x_3)\}, [] \rangle \\ & \rightsquigarrow \langle \{x_1 \doteq x_2, f(x_4) \doteq x_3\}, [] \rangle \\ & \rightsquigarrow \langle \{f(x_4) \doteq x_3\}, [x_1 \mapsto x_2] \rangle \\ & \rightsquigarrow \langle \{x_3 \doteq f(x_4)\}, [x_1 \mapsto x_2] \rangle \\ & \rightsquigarrow \langle \{\}, [x_1 \mapsto x_2, x_3 \mapsto f(x_4)] \rangle \end{aligned}$$

So in this case, the algorithm is successful and the substitution

$$[x_1 \mapsto x_2, x_3 \mapsto f(x_4)]$$

is a solution of the given syntactical equation.

Let us consider another example. We try to solve the following system of syntactical equations:

$$E = \{p(h(x_1, c)) \doteq p(x_2), q(x_2, d) \doteq q(h(d, c), x_4)\}$$

In this case, the algorithm proceeds as follows:

$$\begin{aligned} & \langle \{p(h(x_1, c)) \doteq p(x_2), q(x_2, d) \doteq q(h(d, c), x_4)\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c), d \doteq x_4\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c), x_4 \doteq d\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c)\}, [x_4 \mapsto d] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(h(d, c))\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{h(x_1, c) \doteq h(d, c)\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq d, c \doteq c\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq d, \}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{\}, [x_4 \mapsto d, x_2 \mapsto h(d, c), x_1 \mapsto d] \rangle \end{aligned}$$

Therefore, the substitution $[x_4 \mapsto d, x_2 \mapsto h(d, c), x_1 \mapsto d]$ is a solution of the given set of syntactical equations. \square

4.5 A Calculus for First-order Logic

In this chapter, we define a calculus for first-order logic. This calculus uses two inference rules that we define next.

Definition 59 (Resolution) Assume the following:

1. k_1 and k_2 are first-order clauses,
2. $p(s_1, \dots, s_n)$ and $p(t_1, \dots, t_n)$ are atomic formulæ ,
3. the syntactical equation $p(s_1, \dots, s_n) \doteq p(t_1, \dots, t_n)$ is solvable, we have

$$\mu = mgu(p(s_1, \dots, s_n), p(t_1, \dots, t_n)).$$

Then, the following is an instance of the *resolution rule*:

$$\frac{k_1 \cup \{p(s_1, \dots, s_n)\} \quad \{\neg p(t_1, \dots, t_n)\} \cup k_2}{k_1\mu \cup k_2\mu}.$$

□

The resolution rule is a combination of the *substitution rule* and the cut rule. The substitution rule has the form

$$\frac{k}{k\sigma}.$$

Here, k is a first-order clause and σ is a substitution.

In applications of the resolution rule it might be necessary to rename the variables of one clause before the resolution rule can be applied. Consider the following example. The clause set

$$M = \left\{ \{p(x)\}, \{\neg p(f(x))\} \right\}$$

is inconsistent. However, we can not use the resolution rule directly, as the syntactical equation

$$p(x) \doteq p(f(x))$$

is unsolvable. The reason is, that by a mere **coincidence**, both clauses use the same variable x . However, if we rename the variable x in the second clause into y , we arrive at the following set of clauses:

$$\left\{ \{p(x)\}, \{\neg p(f(y))\} \right\}.$$

Here, the resolution rule can be applied, as the syntactical equation

$$p(x) \doteq p(f(y))$$

has the solution $[x \mapsto f(y)]$. Using this substitution we can derive the empty clause

$$\{p(x)\}, \quad \{\neg p(f(y))\} \quad \vdash \quad \{\}$$

and thus have shown the inconsistency of the set M .

The resolution rule alone is not sufficient in order to show that a given set of clauses M is sufficient. Rather, we need a second rule. To motivate the second rule, consider the following example:

$$M = \left\{ \{p(f(x), y), p(u, g(v))\}, \{\neg p(f(x), y), \neg p(u, g(v))\} \right\}$$

We will show that M is inconsistent. However, the cut rule is not sufficient to show the inconsistency of M . The problem is that any application of the cut rule will always leave us with clauses that contain two literals. A formal proof of this claim can be done using a case distinction which is quite lengthy but nevertheless straightforward. This example motivates the introduction of the following rule.

Definition 60 (Factorization) Assume the following:

1. k is a first-order clause,
2. $p(s_1, \dots, s_n)$ and $p(t_1, \dots, t_n)$ are atomic formulæ,
3. $p(s_1, \dots, s_n) \doteq p(t_1, \dots, t_n)$ is solvable and
4. $\mu = \text{mgu}(p(s_1, \dots, s_n), p(t_1, \dots, t_n))$ is the unifier.

Then both

$$\frac{k \cup \{p(s_1, \dots, s_n), p(t_1, \dots, t_n)\}}{k\mu \cup \{p(s_1, \dots, s_n)\mu\}} \quad \text{and} \quad \frac{k \cup \{\neg p(s_1, \dots, s_n), \neg p(t_1, \dots, t_n)\}}{k\mu \cup \{\neg p(s_1, \dots, s_n)\mu\}}$$

are applications of the *factorization rule*. □

Let us show that using both resolution and factorization we can show the set M given above to be inconsistent.

1. Let us first use factorization with the first clause. In order to do so, we calculate the unifier

$$\mu = \text{mgu}(p(f(x), y), p(u, g(v))) = [y \mapsto g(v), u \mapsto f(x)].$$

Then, factorization yields

$$\{p(f(x), y), p(u, g(v))\} \vdash \{p(f(x), g(v))\}.$$

2. Now we apply factorization to the second clause. We calculate the unifier as

$$\mu = \text{mgu}(\neg p(f(x), y), \neg p(u, g(v))) = [y \mapsto g(v), u \mapsto f(x)].$$

This time, factorization yields:

$$\{\neg p(f(x), y), \neg p(u, g(v))\} \vdash \{\neg p(f(x), g(v))\}.$$

3. We finish our proof with an application of resolution. The unifier we use this time is the empty substitution, we have $\mu = []$, as the corresponding atomic formulæ are already equal:

$$\{p(f(x), g(v))\}, \{\neg p(f(x), g(v))\} \vdash \{\}.$$

If M is a set of clauses and k is a first-order clause such that k can be inferred by successive applications of resolution and factorization from clauses of M , then this is written as

$$M \vdash k.$$

We read this as M *proves* k .

Definition 61 (Universal Closure) If k is a first-order clause and $\{x_1, \dots, x_n\}$ is the set of all variables occurring in k , then we define the universal closure $\forall(k)$ of the clause k as

$$\forall(k) := \forall x_1, \dots, \forall x_n: k.$$

The essential properties of our provability relation $M \vdash k$ are given in the following theorems.

Proposition 62 (Correctness)

If $M = \{k_1, \dots, k_n\}$ is a set of clauses and is $M \vdash k$, then we have

$$\models \forall(k_1) \wedge \dots \wedge \forall(k_n) \rightarrow \forall(k).$$

Therefore, if the clause k can be proven from the set M , then k is indeed a logical consequence of the formulæ in M . \square

The converse of the preceding theorem is only valid for the empty clause.

Proposition 63 (Refutation Completeness)

If $M = \{k_1, \dots, k_n\}$ is a set of clauses and we have $\models \forall(k_1) \wedge \dots \wedge \forall(k_n) \rightarrow \perp$, then we can indeed prove the empty clause from the clauses in M

$$M \vdash \{\}.$$

\square

The preceding theorems provide us with a method to check for a given first-order formula f whether $\models f$ holds. The idea is to proceed as follows:

1. We compute the Skolem normal form of $\neg f$. This Skolem normal form is a formula of the form $\forall x_1, \dots, x_m: g$ and we know that this formula is equisatisfiable with $\neg f$:

$$\neg f \approx_e \forall x_1, \dots, x_m: g.$$

2. Next, we transform the matrix g into conjunctive normal form:

$$g \leftrightarrow k_1 \wedge \dots \wedge k_n.$$

Therefore, we now have

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

and that implies

$$\models f \quad \text{iff} \quad \{\neg f\} \models \perp \quad \text{iff} \quad \{k_1, \dots, k_n\} \models \perp.$$

3. Using the correctness and the refutation completeness we now have

$$\{k_1, \dots, k_n\} \models \perp \quad \text{iff} \quad \{k_1, \dots, k_n\} \vdash \perp.$$

Therefore we try to infer the empty clause from the set of clauses $M = \{k_1, \dots, k_n\}$ using both resolution and factorization. If we are successful, we have shown the original formula f to be universally valid.

We close this section by providing an example. We assume the following axioms:

1. Every dragon is happy if all of its children are able to fly.
2. Every red dragon can fly.
3. The children of red dragons are always red.

We want to show that these axioms imply that all red dragons are happy. As a first step, let us formalize the axioms and the claim in first-order logic. To this end, we define the following signature:

$$\Sigma_{\text{dragon}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle \quad \text{where}$$

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

2. $\mathcal{F} = \{\}$.
3. $\mathcal{P} := \{\text{red}, \text{canFly}, \text{happy}, \text{child}\}$.
4. $\text{arity} := \{\langle \text{red}, 1 \rangle, \langle \text{canFly}, 1 \rangle, \langle \text{happy}, 1 \rangle, \langle \text{child}, 2 \rangle\}$

The predicate $\text{child}(x, y)$ is true iff x is a child of y . We formalize the axioms and the claim:

1. $f_1 := \forall x : (\forall y : (\text{child}(y, x) \rightarrow \text{canFly}(y)) \rightarrow \text{happy}(x))$
2. $f_2 := \forall x : (\text{red}(x) \rightarrow \text{canFly}(x))$
3. $f_3 := \forall x : (\text{red}(x) \rightarrow \forall y : (\text{child}(y, x) \rightarrow \text{red}(y)))$
4. $f_4 := \forall x : (\text{red}(x) \rightarrow \text{happy}(x))$

We want to prove that the formula

$$f := f_1 \wedge f_2 \wedge f_3 \rightarrow f_4$$

is universally valid. Therefore, we take the formula $\neg f$ and see that we have the following equivalence

$$\neg f \leftrightarrow f_1 \wedge f_2 \wedge f_3 \wedge \neg f_4.$$

The next step is to transform the formula on the right hand side of this equivalence into a set of clauses. As we have a conjunction of several formulæ here, it is possible to transform the formulæ f_1 , f_2 , f_3 and $\neg f_4$ separately.

1. The formula f_1 is transformed as follows:

$$\begin{aligned}
f_1 &= \forall x : (\forall y : (\text{child}(y, x) \rightarrow \text{canFly}(y)) \rightarrow \text{happy}(x)) \\
&\leftrightarrow \forall x : (\neg \forall y : (\text{child}(y, x) \rightarrow \text{canFly}(y)) \vee \text{happy}(x)) \\
&\leftrightarrow \forall x : (\neg \forall y : (\neg \text{child}(y, x) \vee \text{canFly}(y)) \vee \text{happy}(x)) \\
&\leftrightarrow \forall x : (\exists y : \neg (\neg \text{child}(y, x) \vee \text{canFly}(y)) \vee \text{happy}(x)) \\
&\leftrightarrow \forall x : (\exists y : (\text{child}(y, x) \wedge \neg \text{canFly}(y)) \vee \text{happy}(x)) \\
&\leftrightarrow \forall x : \exists y : ((\text{child}(y, x) \wedge \neg \text{canFly}(y)) \vee \text{happy}(x)) \\
&\approx_e \forall x : ((\text{child}(s(x), x) \wedge \neg \text{canFly}(s(x))) \vee \text{happy}(x))
\end{aligned}$$

In the last step, we have introduced the Skolem function s where $\text{arity}(s) = 1$. The interpretation of this function is as follows: For every dragon x , that is not happy, $s(x)$ provides a child y of x such that y is not able to fly.

If we transform the matrix of the last formula into conjunctive normal form, we get the following clauses:

$$\begin{aligned}
k_1 &:= \{\text{child}(s(x), x), \text{happy}(x)\}, \\
k_2 &:= \{\neg \text{canFly}(s(x)), \text{happy}(x)\}.
\end{aligned}$$

2. The formula f_2 is transformed as follows:

$$\begin{aligned}
f_2 &= \forall x : (\text{red}(x) \rightarrow \text{canFly}(x)) \\
&\leftrightarrow \forall x : (\neg \text{red}(x) \vee \text{canFly}(x))
\end{aligned}$$

Therefore, f_2 is equivalent to the following clause:

$$k_3 := \{\neg \text{red}(x), \text{canFly}(x)\}.$$

3. For f_3 we get:

$$\begin{aligned} f_3 &= \forall x : \left(\text{red}(x) \rightarrow \forall y : (\text{child}(y, x) \rightarrow \text{red}(y)) \right) \\ &\leftrightarrow \forall x : \left(\neg \text{red}(x) \vee \forall y : (\neg \text{child}(y, x) \vee \text{red}(y)) \right) \\ &\leftrightarrow \forall x : \forall y : (\neg \text{red}(x) \vee \neg \text{child}(y, x) \vee \text{red}(y)) \end{aligned}$$

As a clause, this is written as follows:

$$k_4 := \{ \neg \text{red}(x), \neg \text{child}(y, x), \text{red}(y) \}.$$

4. Transforming the negation of f_4 we get the following:

$$\begin{aligned} \neg f_4 &= \neg \forall x : (\text{red}(x) \rightarrow \text{happy}(x)) \\ &\leftrightarrow \neg \forall x : (\neg \text{red}(x) \vee \text{happy}(x)) \\ &\leftrightarrow \exists x : \neg (\neg \text{red}(x) \vee \text{happy}(x)) \\ &\leftrightarrow \exists x : (\text{red}(x) \wedge \neg \text{happy}(x)) \\ &\approx_e \text{red}(d) \wedge \neg \text{happy}(d) \end{aligned}$$

Here, we have introduced the Skolem constant d . This constant denotes a dragon that is both red and not happy. Written as a set of clauses, the previous formula takes the following form:

$$\begin{aligned} k_5 &= \{ \text{red}(d) \}, \\ k_6 &= \{ \neg \text{happy}(d) \}. \end{aligned}$$

We therefore have to show that the set M consisting of the clauses listed below, is inconsistent:

1. $k_1 = \{ \text{child}(s(x), x), \text{happy}(x) \}$
2. $k_2 = \{ \neg \text{canFly}(s(x)), \text{happy}(x) \}$
3. $k_3 = \{ \neg \text{red}(x), \text{canFly}(x) \}$
4. $k_4 = \{ \neg \text{red}(x), \neg \text{child}(y, x), \text{red}(y) \}$
5. $k_5 = \{ \text{red}(d) \}$
6. $k_6 = \{ \neg \text{happy}(d) \}$

Define $M := \{k_1, k_2, k_3, k_4, k_5, k_6\}$. We will show $M \vdash \perp$.

1. We have

$$\text{mgu}(\text{red}(d), \text{red}(x)) = [x \mapsto d].$$

Therefore, resolution applied to the clauses k_5 and k_4 yields:

$$\{ \text{red}(d) \}, \{ \neg \text{red}(x), \neg \text{child}(y, x), \text{red}(y) \} \vdash \{ \neg \text{child}(y, d), \text{red}(y) \}.$$

2. We apply the resolution rule again for the clause derived above and the clause k_1 . To this end, we first compute

$$\text{mgu}(\text{child}(y, d), \text{child}(s(x), x)) = [y \mapsto s(d), x \mapsto d].$$

Then we have

$$\{ \neg \text{child}(y, d), \text{red}(y) \}, \{ \text{child}(s(x), x), \text{happy}(x) \} \vdash \{ \text{happy}(d), \text{red}(s(d)) \}.$$

3. Next, we apply the resolution rule to the last clause derived and k_6 . We have

$$\text{mgu}(\text{happy}(d), \text{happy}(d)) = []$$

Therefore, we conclude

$$\{\text{happy}(d), \text{red}(s(d))\}, \{\neg \text{happy}(d)\} \vdash \{\text{red}(s(d))\}.$$

4. Next, we apply resolution to the clauses $\{\text{red}(s(d))\}$ and k_3 . We have

$$\text{mgu}(\text{red}(s(d)), \neg \text{red}(x)) = [x \mapsto s(d)]$$

Therefore, resolution yields:

$$\{\text{red}(s(d))\}, \{\neg \text{red}(x), \text{canFly}(x)\} \vdash \{\text{canFly}(s(d))\}$$

5. Next, we apply resolution to $\{\text{canFly}(s(d))\}$ and k_2 . We have

$$\text{mgu}(\text{canFly}(s(d)), \text{canFly}(s(x))) = [x \mapsto d].$$

Now the resolution rule yields

$$\{\text{canFly}(s(d))\}, \{\neg \text{canFly}(s(x)), \text{happy}(x)\} \vdash \{\text{happy}(d)\}.$$

6. The clause $\{\text{happy}(d)\}$ and the clause k_6 are inconsistent, as we have:

$$\{\text{happy}(d)\}, \{\neg \text{happy}(d)\} \vdash \{\}.$$

As we have proven the empty clause, we have shown that $M \vdash \perp$ holds. Therefore, we may conclude that all communist dragons are happy.

Exercise 5: The *Russell set* R is defined as the set of all those sets that do not contain themselves. We therefore have.

$$\forall x : (x \in R \leftrightarrow \neg x \in R).$$

Show that this formula is inconsistent.

Exercise 6: Assume there is a small town with just one barber. Further assume the following:

1. The barber shaves all those villagers that do not shave themselves.
2. The barber does not shave anybody who shaves himself.

Prove, that using these axioms we are actually able to prove that the barber is gay. We are thus able to validate a common prejudice with scientific means!

Chapter 5

Prolog

Im diesem Kapitel wollen wir uns mit dem logischen Programmieren und der Sprache *Prolog* beschäftigen. Der Name *Prolog* steht für “*programming in logic*”. Die Grundidee des logischen Programmieren kann wie folgt dargestellt werden:

1. Der Software-Entwickler erstellt eine Datenbank. Diese enthält Informationen in Form von *Fakten* und *Regeln*.
2. Ein automatischer Beweiser (eine sogenannte *Inferenz-Maschine*) erschließt aus diesen Fakten und Regeln Informationen und beantwortet so Anfragen.

Das besondere an dieser Art der Problemlösung besteht darin, dass es nicht mehr notwendig ist, einen Algorithmus zu entwickeln, der ein bestimmtes Problem löst. Statt dessen wird das Problem durch logische Formeln beschrieben. Zur Lösung des Problems wird dann ein automatischen Beweiser eingesetzt, der die gesuchte Lösung berechnet. Diese Vorgehensweise folgt dem Paradigma des *deklarativen Programmierens*. In der Praxis funktioniert der deklarative Ansatz nur bei einfachen Beispielen. Um mit Hilfe von *Prolog* auch komplexere Aufgaben lösen zu können, ist es unumgänglich, die Funktionsweise des eingesetzten automatischen Beweisers zu verstehen.

Wir geben ein einfaches Beispiel, an dem wir das Grundprinzip des deklarativen Programmierens mit *Prolog* erläutern können. Abbildung 5.1 auf Seite 135 zeigt ein *Prolog*-Programm, das aus einer Ansammlung von *Fakten* und *Regeln* besteht:

1. Ein *Fakt* ist eine atomare Formel. Die Syntax ist

$$p(t_1, \dots, t_n).$$

Dabei ist p ein Prädikats-Zeichen und t_1, \dots, t_n sind Terme. Ist die Menge der Variablen, die in den Termen t_1, \dots, t_n vorkommen, durch $\{x_1, \dots, x_m\}$ gegeben, so wird der obige Fakt als die logische Formel

$$\forall x_1, \dots, x_m: p(t_1, \dots, t_n)$$

interpretiert. Das Programm in Abbildung 5.1 enthält in den Zeilen 1 – 5 Fakten. Umgangssprachlich können wir diese wie folgt lesen:

- (a) Asterix ist ein Gallier.
- (b) Obelix ist ein Gallier.
- (c) Cäsar ist ein Kaiser.
- (d) Cäsar ist ein Römer.

2. Eine *Regel* ist eine bedingte Aussage. Die Syntax ist

$$A :- B_1, \dots, B_n.$$

Dabei sind A und B_1, \dots, B_n atomare Formeln, haben also die Gestalt

$$q(s_1, \dots, s_k),$$

wobei q ein Prädikats-Zeichen ist und s_1, \dots, s_k Terme sind. Ist die Menge der Variablen, die in den atomaren Formeln A, B_1, \dots, B_n auftreten, durch $\{x_1, \dots, x_m\}$ gegeben, so wird die obige Regel als die logische Formel

$$\forall x_1, \dots, x_m : (B_1 \wedge \dots \wedge B_n \rightarrow A)$$

interpretiert. Das Programm aus Abbildung 5.1 enthält in den Zeilen 7–12 Regeln. Schreiben wir diese Regeln als prädikatenlogische Formeln, so erhalten wir:

- (a) $\forall x : (\text{gallier}(x) \rightarrow \text{stark}(x))$
Alle Gallier sind stark.
- (b) $\forall x : (\text{stark}(x) \rightarrow \text{maechtig}(x))$
Wer stark ist, ist mächtig.
- (c) $\forall x : (\text{kaiser}(x) \wedge \text{roemer}(x) \rightarrow \text{maechtig}(x))$
Wer Kaiser und Römer ist, der ist mächtig.
- (d) $\forall x : (\text{roemer}(x) \rightarrow \text{spinnt}(x))$
Wer Römer ist, spinnt.

```

1  gallier(asterix).
2  gallier(obelix).
3
4  kaiser(caesar).
5  roemer(caesar).
6
7  stark(X) :- gallier(X).
8
9  maechtig(X) :- stark(X).
10 maechtig(X) :- kaiser(X), roemer(X).
11
12 spinnt(X) :- roemer(X).
```

Figure 5.1: Ein einfaches Prolog-Programm.

Wir haben in dem Programm in Abbildung 5.1 die Zeile

`stark(X) :- gallier(X).`

als die Formel

$$\forall x : (\text{gallier}(x) \rightarrow \text{stark}(x))$$

interpretiert. Damit eine solche Interpretation möglich ist, muss klar sein, dass in der obigen Regel der String “X” eine Variable bezeichnet, während die Strings “stark” und “gallier” Prädikats-Zeichen sind. Prolog hat sehr einfache Regeln um Variablen von Prädikats- und Funktions-Zeichen unterscheiden zu können:

1. Wenn ein String mit einem großen Buchstaben oder aber mit dem Unterstrich “_” beginnt und nur aus Buchstaben, Ziffern und dem Unterstrich “_” besteht, dann bezeichnet dieser

String eine Variable. Die folgenden Strings sind daher Variablen:

`X, ABC_32, _U, Hugo, _1, _`

2. Strings, die mit einem kleinen Buchstaben beginnen und nur aus Buchstaben, Ziffern und dem Unterstrich “_” bestehen, bezeichnen Prädikats- und Funktions-Zeichen. Die folgenden Strings sind können also als Funktions- oder Prädikats-Zeichen verwendet werden:

`asterix, a1, i_love_prolog, x.`

3. Die Strings

`“+”, “-”, “*”, “/”, “.”`

bezeichnen Funktions-Zeichen. Bis auf das Funktions-Zeichen “.” können diese Funktions-Zeichen auch, wie in der Mathematik üblich, als Infix-Operatoren geschrieben werden. Das Funktions-Zeichen “.” bezeichnen wir als *Dot-Operator*. Dieser Operator wird zur Konstruktion von Listen benutzt. Die Details werden wir später diskutieren.

4. Die Strings

`“<”, “>”, “=”, “=<”, “=>”, “\=”, “==”, “\==”.`

bezeichnen Prädikats-Zeichen. Für diese Prädikats-Zeichen ist eine Infix-Schreibweise zulässig. Gegenüber den Sprachen C und Java gibt es hier die folgenden Unterschiede, die besonders Anfängern oft Probleme bereiten.

- (a) Bei dem Operator “=<” treten die Zeichen “=” und “<” nicht in der selben Reihenfolge auf, wie das in den Sprachen C oder Java der Fall ist, denn dort wird dieser Operator als “<=” geschrieben.
- (b) Der Operator “==” testet, ob die beiden Argumente gleich sind, während der Operator “\==” testet, ob die beiden Werte ungleich sind. In C und Java hat der entsprechende Operator die Form “!=”.
- (c) Der Operator “=” ist der Unifikations-Operator. Bei einem Aufruf der Form

$s = t$

versucht das Prolog-System die syntaktische Gleichung $s \doteq t$ zu lösen. (Leider wird dabei der *Occur-Check* nicht durchgeführt.) Falls dies erfolgreich ist, werden die Variablen, die in den Termen s und t vorkommen, *instantiiert*. Was dabei im Detail passiert, werden wir später sehen.

5. Zahlen bezeichnen 0-stellige Funktions-Zeichen. In Prolog können Sie sowohl ganze Zahlen als auch Fließkomma-Zahlen benutzen. Die Syntax für Zahlen ist ähnlich wie in C, die folgenden Strings stellen also Zahlen dar:

`12, -3, 2.5, 2.3e-5.`

Nachdem wir jetzt Syntax und Semantik des Programms in Abbildung 5.1 erläutert haben, zeigen wir nun, wie Prolog sogenannte *Anfragen* beantwortet. Wir nehmen an, dass das Programm in einer Datei mit dem Namen “gallier.pl” abgespeichert ist. Wir wollen herausfinden, ob es jemanden gibt, der einerseits mächtig ist und der andererseits spinnt. Logisch wird dies durch die folgende Formel ausgedrückt:

$\exists x: (\text{maechtig}(x) \wedge \text{spinnt}(x)).$

Als Prolog-Anfrage können wir den Sachverhalt wie folgt formulieren:

`maechtig(X), spinnt(X).`

Um diese Anfrage auswerten zu können, starten wir das SWI-Prolog-System¹ in einer Shell mit

¹Sie finden dieses Prolog-System im Netz unter www.swi-prolog.org.

dem Kommando:

```
pl
```

Wenn das *Prolog*-System installiert ist, begrüßt uns das System wie folgt:

```
Welcome to SWI-Prolog (Multi-threaded, Version 5.9.7)
Copyright (c) 1990-2009 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.
```

```
For help, use ?- help(Topic). or ?- apropos(Word).
```

```
?-
```

Die Zeichenfolge “?-” ist der *Prolog*-Prompt. Hier geben wir

```
consult(gallier).
```

ein (und zwar mit dem Punkt) und drücken *Return*. Damit fordern wir das System auf, die Fakten und Regeln aus der Datei “gallier.pl” zu laden. Als Ergebnis erhalten wir die Meldung

```
?- consult(gallier).
% gallier compiled 0.00 sec, 1,676 bytes
```

```
Yes
```

```
?-
```

Das Programm wurde erfolgreich geladen und übersetzt. Wir geben nun unsere Anfrage ein

```
?- maechtig(X), spinnt(X).
```

und erhalten als Antwort:

```
X = caesar
```

Wenn wir mit dieser Antwort zufrieden sind, drücken wir *Return* und erhalten einen neuen Prompt. Wenn wir statt dessen nach weiteren Personen suchen wollen, die einerseits mächtig sind und andererseits spinnen, dann geben wir das Zeichen “;” ein, bevor wir *Return* drücken. In diesem Fall erhalten wir die Antwort

```
No
```

```
?-
```

Bei den oben gegebenen Regeln und Fakten gibt es außer Caesar also niemanden, der mächtig ist und spinnt. Wenn wir das *Prolog*-System wieder verlassen wollen, dann geben wir den Befehl “halt.” ein.

5.1 Wie arbeitet *Prolog*?

Das Konzept des logischen Programmierens sieht vor, dass der Benutzer eine Datenbank mit Fakten und Regeln erstellt, die das Problem vollständig beschreiben. Anschließend beantwortet dann das *Prolog*-System mögliche Anfragen mit Hilfe einer Inferenz-Maschine. Um nicht-triviale *Prolog*-Programme erstellen zu können, ist es notwendig zu verstehen, wie das *Prolog*-System Anfragen beantwortet. Um diesen Algorithmus leichter darstellen zu können, vereinbaren wir folgendes: Ist ein Fakt der Form

```
A.
```

gegeben, so formen wir dies zu der Regel

$A :- \text{true}.$

um. Außerdem bezeichnen wir bei einer Klausel

$A :- B_1, \dots, B_n.$

die atomare Formel A als den *Kopf* und die Konjunktion B_1, \dots, B_n als den *Rumpf* der Klausel.

Wir beschreiben nun den Algorithmus, mit dem das *Prolog*-System Anfragen beantwortet.

1. Gegeben

(a) Anfrage: $G = Q_1, \dots, Q_n$

Hier sind Q_1, \dots, Q_n atomare Formeln.

(b) *Prolog*-Programm: P

Da die Reihenfolge der Klauseln für das Folgende relevant ist, fassen wir das *Prolog*-Programm P als Liste von Regeln auf.

2. **Gesucht:** Eine Substitution σ , so dass die Instanz $G\sigma$ aus den Regeln des Programms P folgt:

$$\models \forall(P) \rightarrow \forall(G\sigma).$$

Hier bezeichnet $\forall(P)$ den Allabschluß der Konjunktion aller Klauseln aus P und $\forall(G\sigma)$ bezeichnet entsprechend den Allabschluß von $G\sigma$.

Der Algorithmus selbst arbeitet wie folgt:

1. Suche (der Reihe nach) in dem Programm P alle Regeln

$A :- B_1, \dots, B_m.$

für die der Unifikator $\mu = \text{mgu}(Q_1, A)$ existiert.

2. Gibt es mehrere solche Regeln, so

(a) wählen wir die erste Regel aus, wobei wir uns an der Reihenfolge orientieren, in der die Regeln in dem Programm P auftreten.

(b) Außerdem setzen wir an dieser Stelle einen Auswahl-Punkt (*Choice-Point*), um später hier eine anderer Regel wählen zu können, falls dies notwendig werden sollte.

3. Wir setzen $G := G\mu$, wobei μ der oben berechnete Unifikator ist.

4. Nun bilden wir die Anfrage

$B_1\mu, \dots, B_m\mu, Q_2\mu, \dots, Q_n\mu.$

Jetzt können zwei Fälle auftreten:

(a) $m + n = 0$: Dann ist die Beantwortung der Anfrage erfolgreich und wir geben als Antwort $G\mu$ zurück.

(b) Sonst beantworten wir rekursiv die Anfrage $B_1\mu, \dots, B_m\mu, Q_2\mu, \dots, Q_n\mu$.

Falls die rekursive Beantwortung unter Punkt 4 erfolglos war, gehen wir zum letzten Auswahl-Punkt zurück. Gleichzeitig werden alle Zuweisungen $G := G\mu$, die wir seit diesem Auswahl-Punkt durchgeführt haben, wieder rückgängig gemacht. Anschließend versuchen wir, mit der nächsten möglichen Regel die Anfrage zu beantworten.

Um den Algorithmus besser zu verstehen, beobachten wir die Abarbeitung der Anfrage

```
maechtig(X), spinnt(X).
```

im Debugger des *Prolog*-Systems. Wir geben dazu nach dem Laden des Programms “*gallier.pl*” das Kommando *guitracer* ein.

```
?- guitracer.
```

Wir erhalten die Antwort:

```
% The graphical front-end will be used for subsequent tracing
```




```
Yes
```

```
?-
```

Jetzt starten wir die ursprüngliche Anfrage noch einmal, setzen aber das Kommando *trace* vor unsere Anfrage:

```
?- trace, maechtig(X), spinnt(X).
```


Als Ergebnis wird ein Fenster geöffnet, das Sie in Abbildung 5.2 auf Seite 140 sehen. Unter dem Menü sehen Sie hier eine Werkzeugleiste. Die einzelnen Symbole haben dabei die folgende Bedeutung:

1. Die Schaltfläche  dient dazu, einzelne Unifikationen anzuzeigen. Mit dieser Schaltfläche können wir die meisten Details der Abarbeitung beobachten.
Alternativ hat die Taste “i” die selbe Funktion.
2. Die Schaltfläche  dient dazu, einen einzelnen Schritt bei der Abarbeitung einer Anfrage durchzuführen.
Alternativ hat die Leertaste die selbe Funktion.
3. Die Schaltfläche  dient dazu, die nächste atomare Anfrage in einem Schritt durchzuführen. Drücken wir diese Schaltfläche unmittelbar, nachdem wir das Ziel

```
maechtig(X), roemer(X).
```

einggegeben haben, so würde der Debugger die Anfrage *maechtig(X)* in einem Schritt beantworten. Dabei würde dann die Variable *X* an die Konstante *asterix* gebunden.


Alternativ hat die Taste “s” (*skip*) die selbe Funktion.

4. Die Schaltfläche  dient dazu, die Prozedur, in deren Abarbeitung wir uns befinden, ohne Unterbrechung zu Ende abarbeiten zu lassen. Versuchen wir beispielsweise die atomare Anfrage “*maechtig(X)*” mit der Regel


```
maechtig(X) :- kaiser(X), roemer(X).
```

abzuarbeiten und müssen nun die Anfrage “*kaiser(X), roemer(X).*” beantworten, so würden wir durch Betätigung dieser Schaltfläche sofort die Antwort “*X = caesar*” für diese Anfrage erhalten.

Alternative hat die Taste “f” (*finish*) die selbe Funktion.

5. Die Schaltfläche  dient dazu, eine bereits beantwortete Anfrage noch einmal zu beantworten. Dies kann sinnvoll sein, wenn man beim Tracen einer Anfrage nicht aufgepaßt hat und noch einmal sehen möchte, wie das *Prolog*-System zu seiner Antwort gekommen ist.

Alternative hat die Taste “r” (*retry*) die selbe Funktion.


6. Die Schaltfläche  beantwortet die gestellte Anfrage ohne weitere Unterbrechung.

Alternative hat die Taste “n” (*nodebug*) die selbe Funktion.

Von den weiteren Schaltflächen sind fürs erste nur noch zwei interessant.

7. Die Schaltfläche  läßt das Programm bis zum nächsten Haltepunkt laufen.

Alternative hat die Taste “l” (*continue*) die selbe Funktion.

8. Die Schaltfläche  setzt einen Haltepunkt. Dazu muss vorher der Cursor an die Stelle gebracht werden, an der ein Haltepunkt gesetzt werden soll.

Alternative hat die Taste “!” die selbe Funktion.

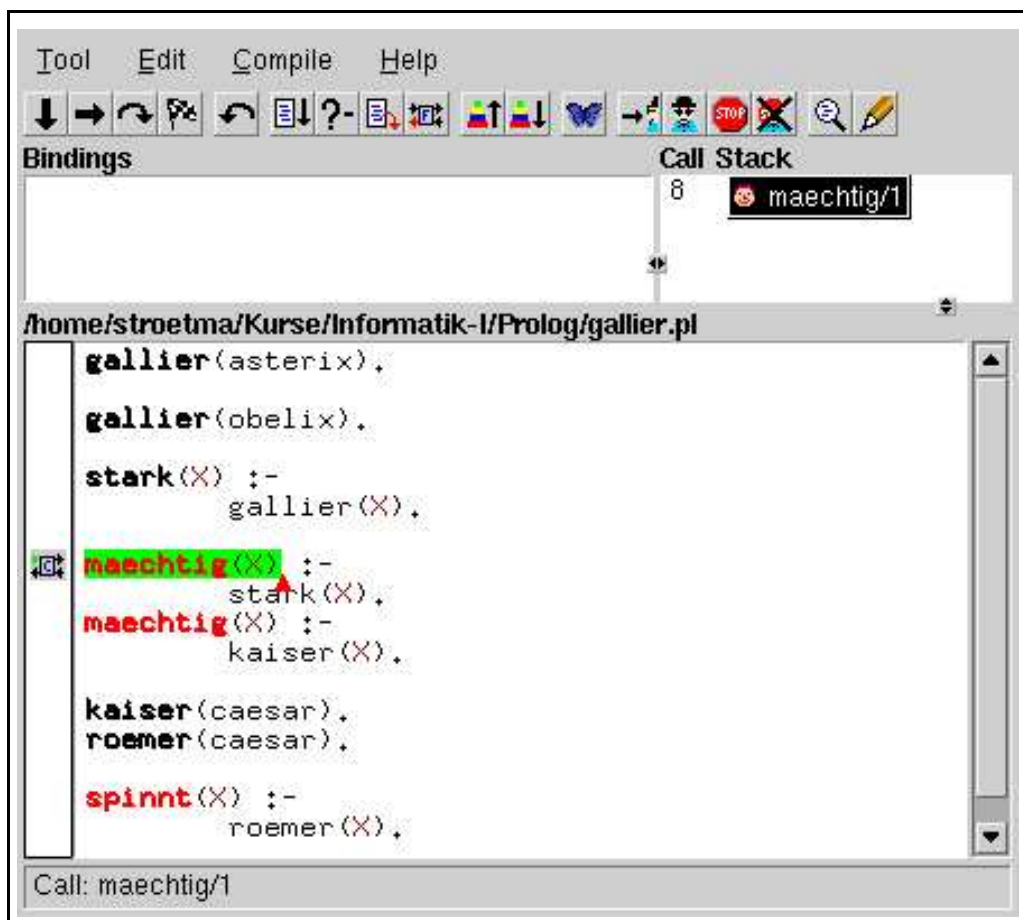


Figure 5.2: Der Debugger des *SWI-Prolog*-Systems.

Wir zeigen, wie das Ziel `maechtig(X)`, `spinnt(X)` vom *Prolog*-System beantwortet wird.

1. Zunächst wird versucht, die Anfrage “`maechtig(X)`” zu lösen. Die erste Regel, die das Prädikat `maechtig/1` definiert, ist

```
maechtig(X) :- stark(X).
```

Daher wird die Anfrage “**maechtig(X)**” reduziert zu der Anfrage “**stark(X)**”. Die aktuelle vollständige Anfrage lautet nun

stark(X), spinnt(X).

Da es noch eine zweite Regel gibt, die das Prädikat **maechtig/1** definiert, setzen wir an dieser Stelle einen Auswahl-Punkt (*Choice-Point*). Falls also die Beantwortung der Anfrage “**stark(X), spinnt(X)**” später scheitert, können wir es mit der zweiten Regel noch einmal versuchen.

2. Jetzt wird versucht, die Anfrage “**stark(X)**” zu lösen. Die erste und einzige Regel, die das Prädikat **stark/1** definiert, ist

stark(X) :- gallier(X).

Nach der Unifikation des Kopfes dieser Regel mit der Anfrage “**stark(X)**” lautet die aktuelle Anfrage

gallier(X), spinnt(X).

3. Die erste Regel, die das Prädikat **gallier/1** definiert und deren Kopf mit der Anfrage “**gallier(X)**” unifiziert werden kann, ist der Fakt

gallier(asterix).

Bei der Unifikation mit diesem Fakt wird die Variable **X** an die Konstante **asterix** gebunden. Damit lautet jetzt die aktuelle Anfrage

spinnt(asterix).

Da es noch eine zweite Regel gibt, die das Prädikat **gallier/1** definiert, setzen wir an dieser Stelle einen Auswahl-Punkt.

4. Die erste und einzige Regel, die das Prädikat **spinnt/1** definiert, lautet

spinnt(X) :- roemer(X).

Also wird nun die Variable **X** in dieser Regel mit **asterix** unifiziert und wir erhalten die Anfrage

roemer(asterix).

5. Die einzige Regel, die das Prädikat **roemer/1** definiert, ist

roemer(caesar).

Diese Regel läßt sich nicht mit der Anfrage “**roemer(asterix)**” unifizieren. Also scheitert diese Anfrage.

6. Wir schauen nun, wann wir das letzte mal einen Auswahl-Punkt gesetzt haben. Wir stellen fest, dass wir unter Punkt 3 bei der Beantwortung der Anfrage **gallier(X)** das letzte Mal einen Auswahl-Punkt gesetzt haben. Also gehen wir nun zu Punkt 3 zurück und versuchen wieder, die Anfrage

gallier(X), spinnt(X)

zu lösen. Diesmal wählen wir jedoch den Fakt

gallier(obelix).

Wir erhalten dann die neue Anfrage

spinnt(obelix).

7. Die erste und einzige Regel, die das Prädikat **spinnt/1** definiert, lautet

spinnt(X) :- roemer(X).

Also wird die Variable **X** in dieser Regel mit **obelix** unifiziert und wir erhalten die Anfrage

roemer(obelix).

8. Die einzige Regel, die das Prädikat **roemer/1** definiert, ist

roemer(caesar).

Diese Regel läßt sich nicht mit der Anfrage “**roemer(asterix)**” unifizieren. Also scheitert diese Anfrage.

9. Wir schauen wieder, wann das letzte Mal ein Auswahl-Punkt gesetzt wurde. Der unter Punkt 3 gesetzte Auswahl-Punkt wurde vollständig abgearbeitet, dieser Auswahl-Punkt kann uns also nicht mehr helfen. Aber unter Punkt 1 wurde ebenfalls ein Auswahl-Punkt gesetzt, denn für das Prädikat **maechtig/1** gibt es die weitere Regel

maechtig(X) :- kaiser(X), roemer(X).

Wenden wir diese Regel an, so erhalten wir die Anfrage

kaiser(X), roemer(X), spinnt(X).

10. Für das Prädikat **kaiser/1** enthält unsere Datenbank genau einen Fakt:

kaiser(caesar).

Benutzen wir diesen Fakt zur Reduktion unserer Anfrage, so lautet die neue Anfrage

roemer(caesar), spinnt(caesar).

11. Für das Prädikat **roemer/1** enthält unsere Datenbank genau einen Fakt:

roemer(caesar).

Benutzen wir diesen Fakt zur Reduktion unserer Anfrage, so lautet die neue Anfrage

spinnt(caesar).

12. Die erste und einzige Regel, die das Prädikat **spinnt/1** definiert, lautet

spinnt(X) :- roemer(X).

Also wird die Variable **X** in dieser Regel mit **caesar** unifiziert und wir erhalten die Anfrage

roemer(caesar).

13. Für das Prädikat **roemer/1** enthält unsere Datenbank genau einen Fakt:

roemer(caesar).

Benutzen wir diesen Fakt zur Reduktion unserer Anfrage, so ist die verbleibende Anfrage leer. Damit ist die ursprüngliche Anfrage gelöst. Die dabei berechnete Antwort erhalten wir, wenn wir untersuchen, wie die Variable **X** unifiziert worden ist. Die Variable **X** war unter Punkt 10 mit der Konstanten **caesar** unifiziert worden. Also ist

X = caesar

die Antwort, die von dem System berechnet wird.

Bei der Beantwortung der Anfrage “**maechtig(X), spinnt(X)**” sind wir einige Male in Sackgassen hineingelaufen und mussten Instantiierungen der Variable **X** wieder zurück nehmen. Dieser Vorgang wird in der Literatur als *backtracking* bezeichnet. Er kann mit Hilfe des Debuggers am Bildschirm verfolgt werden.

5.1.1 Die Tiefensuche

Der von Prolog verwendete Such-Algorithmus wird auch als *Tiefensuche* (angelsächsisch: *depth first search*) bezeichnet. Um diesen Ausdruck erläutern zu können, definieren wir zunächst den Begriff des *Suchbaums*. Die Knoten eines Suchbaums sind mit Anfrage beschriftet. Ist der Knoten u des Suchbaums mit der Anfrage

$$Q_1, \dots, Q_m$$

beschriftet und gibt es für $i = 1, \dots, k$ Regeln der Form

$$A^{(i)} :- B_1^{(i)}, \dots, B_{n(i)}^{(i)}$$

die auf die Anfrage passen, für die also $\mu_i = mgu(Q_1, A^{(i)})$ existiert, so hat der Knoten u insgesamt k verschiedene Kinder. Dabei ist das i -te Kind mit der Anfrage

$$B_1^{(i)}\mu_i, \dots, B_{n(i)}^{(i)}\mu_i, Q_2\mu_i, \dots, Q_m\mu_i$$

beschriftet. Als einfaches Beispiel betrachten wir das in Abbildung 5.3 gezeigte Programm. Der Suchbaum für die Anfrage

$$p(X)$$

ist in Abbildung 5.4 gezeigt. Den Knoten, der mit der ursprünglichen Anfrage beschriftet ist, bezeichnen wir als wie *Wurzel* des Suchbaums. Die *Lösungen* zu der ursprünglichen Anfrage finden wir an den *Blättern* des Baumes: Wir bezeichnen hier die Knoten als Blätter, die am weitesten unten stehen. Suchbäume stehen also gewissermaßen auf dem Kopf: Die Blätter sind unten und die Wurzeln sind oben².

1	$p(X) :- q1(X).$
2	$p(X) :- q2(X).$
3	
4	$q1(X) :- r1(X).$
5	$q1(X) :- r2(X).$
6	
7	$q2(X) :- r3(X).$
8	$q2(X) :- r4(X).$
9	
10	$r1(a).$
11	$r2(b).$
12	$r3(c).$
13	$r4(d).$

Figure 5.3: Die Tiefensuche in Prolog

Anhand des in Abbildung 5.4 gezeigten Suchbaums läßt sich nun die Tiefensuche erklären: Wenn der Prolog-Interpreter nach einer Lösung sucht, so wählt es immer den linken Ast und steigt dort so tief wie möglich ab. Dadurch werden die Lösungen in dem Beispiel in der Reihenfolge

$$X = a, X = b, X = c, X = d,$$

gefunden. Die Tiefensuche ist dann problematisch, wenn der linke Ast des Suchbaums unendlich tief ist. Als Beispiel betrachten wir das in Abbildung 5.5 gezeigte Prolog-Programm. Zeichnen wir hier den Suchbaum für die Anfrage

$$p(X)$$

²Daher werden diese Suchbäume auch als australische Bäume bezeichnet.

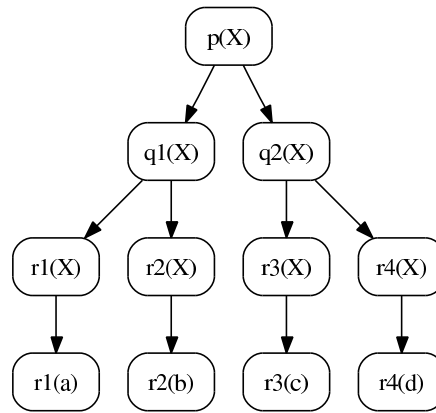


Figure 5.4: Der Suchbaum für das in Abbildung 5.3 gezeigte Programm.

so finden wir einen unendlichen Ast, in den der Prolog-Interpreter absteigt und aus dem er dann mit einem Stack-Overflow wieder zurück kommt. Vertauschen wir hingegen die Reihenfolge der Klauseln, so kann das Programm die obige Anfrage beantworten.

```

1  p(s(X)) :- p(X).
2  p(c).

```

Figure 5.5: Eine Endlos-Schleife in *Prolog*.

5.2 Ein komplexeres Beispiel

Das obige Beispiel war bewußt einfach gehalten um die Sprache *Prolog* einzuführen. Um die Mächtigkeit des Backtrackings zu demonstrieren, präsentieren wir jetzt ein komplexeres Beispiel. Es handelt sich um das folgende Rätsel:

1. Drei Freunde belegen den ersten, zweiten und dritten Platz bei einem Programmier-Wettbewerb.
2. Jeder der drei hat genau einen Vornamen, genau ein Auto und hat sein Programm in genau einer Programmier-Sprache geschrieben.
3. Michael programmiert in *Setl* und war besser als der Audi-Fahrer.
4. Julia, die einen Ford Mustang fährt, war besser als der Java-Programmierer.
5. Das Prolog-Programm war am besten.
6. Wer fährt Toyota?
7. In welcher Sprache programmiert Thomas?

Um dieses Rätsel zu lösen, überlegen wir uns zunächst, wie wir die einzelnen Daten repräsentieren können, die in dem Rätsel eine Rolle spielen. Zunächst ist dort von Personen die Rede. Jede dieser Personen hat genau einen Vornamen, ein Auto und eine Programmier-Sprache. Wir repräsentieren Personen daher durch Terme der Form

`person(Name, Car, Language).`

Dabei bezeichnen *Name*, *Car* und *Language* Konstanten, die aus den entsprechenden Mengen

gewählt werden:

$Name \in \{julia, thomas, michael\}, \quad Car \in \{ford, toyota, audi\},$
 $Language \in \{java, prolog, set1\}.$

Wenn wir Personen durch ein dreistelliges Funktions-Zeichen wie oben gezeigt repräsentieren, können wir sofort Prädikate angeben, die den Vornamen, die Auto-Marke und die Programmier-Sprache aus einem solchen Term extrahieren.

1. Das Prädikat `first_name/2` extrahiert den Vornamen:

`first_name(person(Name, Car, Language), Name).`

2. Das Prädikat `car/2` extrahiert die Auto-Marke:

`car(person(Name, Car, Language), Car).`

3. Das Prädikat `language/2` extrahiert die Programmier-Sprache:

`language(person(Name, Car, Language), Language).`

Um zu verstehen wie diese Prädikate arbeiten, zeigen wir, wie die Anfrage

`car(person(hans, seat, set1), X).`

von dem *Prolog*-System beantwortet wird. Die einzige Regel, die zur Beantwortung dieser Anfrage herangezogen werden kann, ist die Regel

`car(person(Name, Car, Language), Car) :- true.`

Um diese Regel anwenden zu können, ist die syntaktische Gleichung

`car(person(hans, seat, set1), X) \doteq car(person(Name, Car, Language), Car)`

zu lösen. Bei der Unifikation findet sich die Lösung

$\mu = [Name \mapsto \text{hans}, Car \mapsto \text{seat}, Language \mapsto \text{set1}, X \mapsto \text{seat}].$

Insbesondere wird also die Variable `X` bei dieser Anfrage an die Konstante `seat` gebunden.

Wie können wir nun die Reihenfolge repräsentieren, in der die drei Personen bei dem Wettbewerb abgeschnitten haben? Wir wählen ein dreistelliges Funktions-Zeichen `sequence` und repräsentieren die Reihenfolge durch den Term

`sequence(First, Second, Third).`

Dabei stehen *First*, *Second* und *Third* für Terme, die von dem Funktions-Zeichen `person/3` erzeugt worden sind und die Personen bezeichnen. Die Reihenfolge kann dann durch das Prädikat

`did_better(Better, Worse, Sequence)`

berechnet werden, dessen Implementierung in den Zeilen 38 – 40 der Abbildung 5.6 auf Seite 147 gezeigt ist. Wir können nun daran gehen, das Rätsel zu lösen. Abbildung 5.6 zeigt die Implementierung. Zeile 1 – 30 enthält die Implementierung einer Regel für das Prädikat `answer/2`, dass die Lösung des Rätsels berechnet. In dieser Regel haben wir das Rätsel als prädikatenlogische Formel codiert. Wir übersetzen diese Regel jetzt zurück in die Umgangssprache und zeigen dadurch, dass das Prädikat `answer/2` das Rätsel korrekt beschreibt. Die Numerierung in der folgenden Aufzählung stimmt jeweils mit der entsprechenden Zeilen-Nummer im Programm überein:

2. Falls `Sequence` eine Reihenfolge von drei Personen beschreibt und
4. `Michael` eine Person aus dieser Reihenfolge ist und
5. der Name der durch `Michael` bezeichneten Person den Wert `michael` hat und

- 6. **Michael** in SETL programmiert und
- 8. **Audi** eine Person aus der Reihenfolge **Sequence** ist und
- 9. **Michael** beim Wettbewerb besser abgeschnitten hat als die durch **Audi** bezeichnete Person
- 10. die durch **Audi** bezeichnete Person einen Audi fährt und
- :
- 24. **Toyota** eine Person aus der Reihenfolge **Sequence** ist und
- 25. die durch **Toyota** bezeichnete Person einen Toyota fährt und
- 26. **NameToyota** den Vornamen der durch **Toyota** bezeichneten Person angibt und
- 28. **Thomas** eine Person aus der Reihenfolge **Sequence** ist und
- 25. die durch **Thomas** bezeichnete Person den Vornamen Thomas hat und
- 26. **LanguageThomas** die Sprache ist, in der die durch **Thomas** bezeichnete Person programmiert, dann gilt:
- 1. **NameToyota** ist der Namen des Toyota-Fahrers und **LanguageThomas** ist die Sprache, in der Thomas programmiert.

Wenn wir die ursprüngliche Aufgabe mit der Implementierung in *Prolog* vergleichen, dann stellen wir fest, dass die in dem Rätsel gemachten Angaben eins-zu-eins in *Prolog* übersetzt werden konnten. Diese Übersetzung beschreibt nur das Rätsel und gibt keinen Hinweis, wie dieses Rätsel zu lösen ist. Für die Lösung ist dann die dem *Prolog*-System zu Grunde liegende *Inferenz-Maschine* zuständig.

```

1  answer(NameToyota, LanguageThomas) :-
2      is_sequence( Sequence ),
3      % Michael programmiert in Setl.
4      one_of_them(Michael, Sequence),
5      first_name(Michael, michael),
6      language(Michael, setl),
7      % Michael war besser als der Audi-Fahrer
8      one_of_them(Audi, Sequence),
9      did_better(Michael, Audi, Sequence),
10     car(Audi, audi),
11     % Julia fährt einen Ford Mustang.
12     one_of_them(Julia, Sequence),
13     first_name(Julia, julia),
14     car(Julia, ford),
15     % Julia war besser als der Java-Programmierer.
16     one_of_them(JavaProgrammer, Sequence),
17     language(JavaProgrammer, java),
18     did_better(Julia, JavaProgrammer, Sequence),
19     % Das Prolog-Programm war am besten.
20     one_of_them(PrologProgrammer, Sequence),
21     first(PrologProgrammer, Sequence),
22     language(PrologProgrammer, prolog),
23     % Wer fährt Toyota?
24     one_of_them(Toyota, Sequence),
25     car(Toyota, toyota),
26     first_name(Toyota, NameToyota),
27     % In welcher Sprache programmiert Thomas?
28     one_of_them(Thomas, Sequence),
29     first_name(Thomas, thomas),
30     language(Thomas, LanguageThomas).
31
32 is_sequence( sequence(_First, _Second, _Third) ).
33
34 one_of_them(A, sequence(A, _, _)).
35 one_of_them(B, sequence(_, B, _)).
36 one_of_them(C, sequence(_, _, C)).
37
38 did_better(A, B, sequence(A, B, _)).
39 did_better(A, C, sequence(A, _, C)).
40 did_better(B, C, sequence(_, B, C)).
41
42 first(A, sequence(A, _, _)).
43
44 first_name(person(Name, _Car, _Language), Name).
45
46 car(person(_Name, Car, _Language), Car).
47
48 language(person(_Name, _Car, Language), Language).

```

Figure 5.6: Wer fährt Toyota?

5.3 Listen

In Prolog wird viel mit Listen gearbeitet. Listen werden in Prolog mit dem 2-stelligen Funktions-Zeichen “.” konstruiert. Ein Term der Form

`.(s,t)`

steht also für eine Liste, die als erstes Element “s” enthält. “t” bezeichnet den Rest der Liste. Das Funktions-Zeichen “[]” steht für die leere Liste. Eine Liste, die aus den drei Elementen “a”, “b” und “c” besteht, kann also wie folgt dargestellt werden:

`.(a, .(b, .(c, [])))`

Da dies relativ schwer zu lesen ist, darf diese Liste auch als

`[a,b,c]`

geschrieben werden. Zusätzlich kann der Term “.(s,t)” in der Form

`[s | t]`

geschrieben werden. Um diese Kurzschreibweise zu erläutern, geben wir ein kurzes Prolog-Programm an, dass zwei Listen aneinander hängen kann. Das Programm implementiert das dreistellige Prädikat `concat`³. Die Intention ist, dass `concat(l1, l2, l3)` für drei Listen *l₁*, *l₂* und *l₃* genau dann wahr sein soll, wenn die Liste *l₃* dadurch entsteht, dass die Liste *l₂* hinten an die Liste *l₁* angehängt wird. Das Programm besteht aus den folgenden beiden Klauseln:

```
concat( [], L, L ).
concat( [ X | L1 ], L2, [ X | L3 ] ) :- concat( L1, L2, L3 ).
```

Wir können diese beiden Klauseln folgendermaßen in die Umgangssprache übersetzen:

1. Hängen wir eine Liste L an die leere Liste an, so ist das Ergebnis die Liste L.
2. Um an eine Liste `[X | L1]`, die aus dem Element X und dem Rest L1 besteht, eine Liste L2 anzuhängen, hängen wir zunächst an die Liste L1 die Liste L2 an und nennen das Ergebnis L3. Das Ergebnis erhalten wir, wenn wir vor L3 noch das Element X setzen. Wir erhalten dann die Liste `[X | L3]`.

Wir testen unser Programm und nehmen dazu an, dass die beiden Programm-Klauseln in der Datei “`concat.pl`” abgespeichert sind und dass wir diese Datei mit dem Befehl “`consult(concat).`” geladen haben. Dann stellen wir die Anfrage

```
?- concat( [ 1, 2, 3 ], [ a, b, c ], L ).
```

Wir erhalten die Antwort:

```
L = [1, 2, 3, a, b, c]
```

Die obige Interpretation des gegebenen Prolog-Programms ist *funktional*, dass heißt wir fassen die ersten beiden Argumente des Prädikats `concat` als *Eingaben* auf und interpretieren das letzte *Argument* als Ausgabe. Diese Interpretation ist aber keineswegs die einzig mögliche Interpretation. Um das zu sehen, geben wir als Ziel

```
concat(L1, L2, [1,2,3]).
```

ein und drücken, nachdem das System uns die erste Antwort gegeben hat, nicht die Taste *Return* sondern die Taste “;”. Wir erhalten:

³In dem SWI-Prolog-System gibt es das vordefinierte Prädikat `append/3`, das genau das selbe leistet wie unsere Implementierung von `concat/3`.

```
?- concat(L1, L2, [1, 2, 3]).
```

```
L1 = []  
L2 = [1, 2, 3] ;
```

```
L1 = [1]  
L2 = [2, 3] ;
```

```
L1 = [1, 2]  
L2 = [3] ;
```

```
L1 = [1, 2, 3]  
L2 = [] ;
```

No

In diesem Fall hat das Prolog-System durch Backtracking alle Möglichkeiten bestimmt, die es gibt, um die Liste “[1, 2, 3]” in zwei Teile zu zerlegen.

5.3.1 Sortieren durch Einfügen

Wir entwickeln nun einen einfachen Algorithmus zum Sortieren von Listen von Zahlen. Die Idee ist folgende: Um eine Liste aus n Zahlen zu sortieren, sortieren wir zunächst die letzten $n - 1$ Zahlen und fügen dann das erste Element in die sortierte Liste an der richtigen Stelle ein. Mit dieser Idee besteht das Programm aus zwei Prädikaten:

1. Das Prädikat `insert/3` erwartet als erstes Argument eine Zahl x und als zweites Argument eine Liste von Zahlen l , die zusätzlich noch in aufsteigender Reihenfolge sortiert sein muss. Das Prädikat fügt die Zahl x so in die Liste l ein, dass die resultierende Liste wiederum in aufsteigender Reihenfolge sortiert ist. Das so berechnete Ergebnis wird als letztes Argument des Prädikats zurück gegeben.

Um die obigen Ausführungen über die verwendeten Typen und die Bestimmung von Ein- und Ausgabe prägnanter formulieren zu können, führen wir den Begriff einer *Typ-Spezifikation* ein. Für das Prädikat `insert/3` hat diese Typ-Spezifikation die Form

```
insert(+Number, +List(Number), -List(Number)).
```

Das Zeichen “+” legt dabei fest, dass das entsprechende Argument eine Eingabe ist, während “-” verwendet wird um ein Ausgabe-Argument zu spezifizieren.

2. Das Prädikat `insertion_sort/2` hat die Typ-Spezifikation

```
insertion_sort(+ List(Number), -List(Number)).
```

Der Aufruf `insertion_sort(List, Sorted)` sortiert *List* in aufsteigender Reihenfolge.

Abbildung 5.7 zeigt das *Prolog*-Programm.

Nachfolgend diskutieren wir die einzelnen Klauseln der Implementierung des Prädikats `insert`.

1. Die erste Klausel des Prädikats `insert` greift, wenn die Liste, in welche die Zahl x eingefügt werden soll, leer ist. In diesem Fall wird als Ergebnis einfach die Liste zurück gegeben, die als einziges Element die Zahl x enthält.
2. Die zweite Klausel greift, wenn die Liste, in die die Zahl x eingefügt werden soll nicht leer ist und wenn außerdem x kleiner oder gleich dem ersten Element dieser Liste ist. In diesem Fall kann x an den Anfang der Liste gestellt werden. Dann erhalten wir die Liste

```

1  % insert( +Number, +List(Number), -List(Number) ).
2
3  insert( X, [], [ X ] ).
4
5  insert( X, [ Head | Tail ], [ X, Head | Tail ] ) :-
6      X <= Head.
7
8  insert( X, [ Head | Tail ], [ Head | New_Tail ] ) :-
9      X > Head,
10     insert( X, Tail, New_Tail ).
11
12 % insertion_sort( +List(Number), -List(Number) ).
13
14 insertion_sort( [], [] ).
15
16 insertion_sort( [ Head | Tail ], Sorted ) :-
17     insertion_sort( Tail, Sorted_Tail ),
18     insert( Head, Sorted_Tail, Sorted ).

```

Figure 5.7: Sortieren durch Einfügen.

[X, Head | Tail].

Diese Liste ist sortiert, weil einerseits schon die Liste [Head | Tail] sortiert ist und andererseits X kleiner als Head ist.

3. Die dritte Klausel greift, wenn die Liste, in die die Zahl X eingefügt werden soll nicht leer ist und wenn außerdem X größer als das erste Element dieser Liste ist. In diesem Fall muss X rekursiv in die Liste Tail eingefügt werden. Dabei bezeichnet Tail den Rest0 der Liste, in die wir X einfügen wollen. Weiter bezeichnet New_Tail die Liste, die wir erhalten, wenn wir die Zahl X in die Liste Tail einfügen. An den Anfang der Liste New_Tail setzen wir nun noch den Kopf Head der als Eingabe gegebenen Liste.

Damit können wir nun auch die Wirkungsweise des Prädikats `insertion_sort` erklären.

1. Ist die zu sortierende Liste leer, so ist das Ergebnis die leere Liste.
2. Ist die zu sortierende Liste nicht leer und hat die Form [Head | Tail], so sortieren wir zunächst die Liste Tail und erhalten als Ergebnis die sortierte Liste Sorted_Tail. Fügen wir hier noch das Element Head mit Hilfe von `insert` ein, so erhalten wir als Endergebnis die sortierte Liste.

Viele *Prolog*-Prädikate sind *funktional*. Wir nennen ein Prädikat funktional, wenn die einzelnen Argumente klar in Eingabe- und Ausgabe-Argumente unterschieden werden können und wenn außerdem zu jeder Eingabe höchstens eine Ausgabe berechnet wird. Zum Beispiel sind die oben angegebenen Prädikate zum Sortieren einer Liste von Zahlen funktional. Bei einem funktionalen Programm können wir die Semantik oft dadurch am besten verstehen, dass wir das Programm in *bedingte Gleichungen* umformen. Für das oben angegebene Programm erhalten wir dann die folgenden Gleichungen:

1. $\text{insert}(X, []) = [X]$.
2. $X \leq \text{Head} \rightarrow \text{insert}(X, [\text{Head}|\text{Tail}]) = [X, \text{Head}|\text{Tail}]$.
3. $X > \text{Head} \rightarrow \text{insert}(X, [\text{Head}|\text{Tail}]) = [\text{Head}|\text{insert}(X, \text{Tail})]$.

4. `insertion_sort([]) = []`.

5. `insertion_sort([Head|Tail]) = insert(Head, insertion_sort(Tail))`.

Die Korrespondenz zwischen dem *Prolog*-Programm und den Gleichungen sollte augenfällig sein. Außerdem ist offensichtlich, dass die obigen Gleichungen den Sortier-Algorithmus in sehr prägnanter Form wiedergeben. Wir werden diese Beobachtung im zweiten Semester benutzen und die meisten dort vorgestellten Algorithmen durch bedingte Gleichungen spezifizieren.

5.3.2 Sortieren durch Mischen

Der im letzten Abschnitt vorgestellte Sortier-Algorithmus hat einen Nachteil: Die Rechenzeit, die dieser Algorithmus verbraucht, wächst im ungünstigsten Fall quadratisch mit der Länge der zu sortierenden Liste. Den Beweis dieser Behauptung werden wir im nächsten Semester liefern. Wir werden nun einen Algorithmus vorstellen der effizienter ist: Ist n die Länge der Liste, so wächst bei diesem Algorithmus der Verbrauch der Rechenzeit nur mit dem Faktor $n * \log_2(n)$. Den Nachweis dieser Behauptung erbringen wir im zweiten Semester. Wenn es sich bei der zu sortierenden Liste beispielsweise um ein Telefonbuch mit einer Millionen Einträgen handelt, dann ist der relative Unterschied des Rechenzeit-Verbrauchs durch einen Faktor der Größe 50 000 gegeben.

Wir werden den effizienteren Algorithmus zunächst durch bedingte Gleichungen beschreiben und anschließend die Umsetzung dieser Gleichungen in *Prolog* angeben. Der Algorithmus wird in der Literatur als *Sortieren durch Mischen* bezeichnet (engl. *merge sort*) und besteht aus drei Phasen:

1. In der ersten Phase wird die zu sortierende Liste in zwei etwa gleich große Teile aufgeteilt.
2. In der zweiten Phase werden diese Teile rekursiv sortiert.
3. In der dritten Phase werden die sortierten Teillisten so zusammen gefügt (gemischt), dass die resultierende Liste ebenfalls sortiert ist.

Wir beginnen mit dem Aufteilen einer Liste in zwei Teile. Bei der Aufteilung orientieren wir uns an den Indizes der Elemente. Zur Illustration zunächst ein Beispiel: Wir teilen die Liste

$[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8]$ auf in $[a_1, a_3, a_5, a_7]$ und $[a_2, a_4, a_6, a_8]$.

Elemente, deren Index gerade ist, werden in der ersten Teilliste aufgesammelt und die Elemente mit ungeradem Index sammeln wir in der zweiten Teilliste. Als Namen für die Funktionen, die diese Teillisten berechnen, wählen wir **even** und **odd**:

odd : $List(Number) \rightarrow List(Number)$,

even : $List(Number) \rightarrow List(Number)$.

Die Funktion **odd**(L) berechnet die Liste aller Elemente aus L mit ungeradem Index, während **even**(L) die Liste aller Elemente mit geradem Index berechnet. Die beiden Funktionen können durch die folgenden Gleichungen spezifiziert werden:

1. **odd**($[]$) = $[]$.

2. **odd**($[h|t]$) = $[h|\text{even}(t)]$,

denn das erste Element einer Liste hat den Index 1, was offenbar ein ungerader Index ist und alle Elemente, die in der Liste t einen geraden Index haben, haben in der Liste $[h|t]$ einen ungeraden Index.

3. **even**($[]$) = $[]$.

```

1  % odd( +List(Number), -List(Number) ).
2  odd( [], [] ).
3  odd( [ X | Xs ], [ X | L ] ) :-
4      even( Xs, L ).
5
6  % even( +List(Number), -List(Number) ).
7  even( [], [] ).
8  even( [ _X | Xs ], L ) :-
9      odd( Xs, L ).
10
11 % merge( +List(Number), +List(Number), -List(Number) ).
12 mix( [], Xs, Xs ).
13 mix( Xs, [], Xs ).
14 mix( [ X | Xs ], [ Y | Ys ], [ X | Rest ] ) :-
15     X <= Y,
16     mix( Xs, [ Y | Ys ], Rest ).
17 mix( [ X | Xs ], [ Y | Ys ], [ Y | Rest ] ) :-
18     X > Y,
19     mix( [ X | Xs ], Ys, Rest ).
20
21 % merge_sort( +List(Number), -List(Number) ).
22 merge_sort( [], [] ).
23 merge_sort( [ X ], [ X ] ).
24 merge_sort( [ X, Y | Rest ], Sorted ) :-
25     odd( [ X, Y | Rest ], Odd ),
26     even( [ X, Y | Rest ], Even ),
27     merge_sort( Odd, Odd_Sorted ),
28     merge_sort( Even, Even_Sorted ),
29     mix( Odd_Sorted, Even_Sorted, Sorted ).

```

Figure 5.8: Sortieren durch Mischen.

4. $\text{even}([h|t]) = \text{odd}(t)$,

denn alle Elemente, die in der Liste t einen ungeraden Index haben, haben in der Liste $[h|t]$ einen geraden Index.

Als nächstes entwickeln wir eine Funktion

$$\text{mix} : \text{List}(\text{Number}) \times \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number})$$

die zwei aufsteigend sortierte Listen so mischt, dass die resultierende Liste ebenfalls aufsteigend sortiert ist. Durch rekursive Gleichungen kann diese Funktion wie folgt spezifiziert werden:

1. $\text{mix}([], l) = l$.

2. $\text{mix}(l, []) = l$.

3. $x \leq y \rightarrow \text{mix}([x|s], [y|t]) = [x|\text{mix}(s, [y|t])]$.

Falls $x \leq y$ ist, so ist x sicher das kleinste Element der Liste die entsteht, wenn wir die Listen $[x|s]$ und $[y|t]$ mischen. Also mischen wir rekursiv die Listen s und $[y|t]$ und setzen x an den Anfang dieser Liste.

4. $x > y \rightarrow \text{mix}([x|s], [y|t]) = [y|\text{mix}([x|s], t)]$.

Damit können wir jetzt die Funktion

$$\text{merge_sort} : \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number}),$$

die eine Liste von Zahlen sortiert, durch bedingte Gleichungen spezifizieren.

1. $\text{merge_sort}([]) = []$.
2. $\text{merge_sort}([x]) = [x]$.
3. $\text{length}(l) \geq 2 \rightarrow \text{merge_sort}(l) = \text{mix}(\text{merge_sort}(\text{odd}(l)), \text{merge_sort}(\text{even}(l)))$.

Falls die Liste l aus 2 oder mehr Elementen besteht, teilen wir diese Liste in die beiden Listen $\text{odd}(l)$ und $\text{even}(l)$ auf, sortieren diese Listen und mischen anschließend die sortierten Teillisten.

Die oben angegebenen Gleichungen lassen sich nun unmittelbar in ein *Prolog*-Programm umsetzen. Abbildung 5.8 auf Seite 152 zeigt das resultierende *Prolog*-Programm. Da es in *SWI-Prolog* bereits vordefinierte Prädikate mit den Namen `merge/3` und `sort/2` gibt, haben wir statt dessen die Namen `mix/2` und `merge_sort/3` gewählt.

5.3.3 Symbolisches Differenzieren

Die Sprache *Prolog* wird gerne für Anwendungen benutzt, bei denen symbolische Rechnungen eine wesentliche Rolle spielen, denn symbolische Rechnungen sind in *Prolog* dadurch, dass die zu manipulierenden Objekte in der Regel unmittelbar als Prolog-Terme dargestellt werden können, sehr einfach zu implementieren. Zur Verdeutlichung zeigen wir ein Programm, mit dem es möglich ist, symbolisch zu differenzieren. Im Rahmen einer Übung haben wir ein SETL-Programm entwickelt, das arithmetische Ausdrücke symbolisch differenziert. Damals war es notwendig gewesen, die zu differenzierenden Terme durch geeignete SETL-Objekte zu repräsentieren. An dieser Stelle ist die *Prolog*-Implementierung einfacher, denn arithmetische Ausdrücke können unmittelbar durch Terme dargestellt werden.

Die Methodik, mit der wir das *Prolog*-Programm entwickeln, besteht aus zwei Schritten:

1. Als erstes legen wir fest, was genau wir unter einem arithmetischen Ausdruck verstehen wollen und wie ein solcher Ausdruck in *Prolog* repräsentiert werden soll. Dazu definieren wir die Menge der *Prolog*-Terme *Expr*, die einen arithmetischen Ausdruck darstellen.
2. Dann stellen wir bedingte Gleichungen auf, die eine Funktion

$$\text{diff} : \text{Expr} \times \text{Var} \rightarrow \text{Expr}$$

beschreiben. Diese Gleichungen sind nichts anderes als die mathematischen Regeln, die Sie in der Schule für das Differenzieren gelernt haben.

3. Im letzten Schritt implementieren wir diese Gleichungen in Prolog.

Induktive Definition der Menge *Expr*.

1. Variablen sind arithmetische Ausdrücke.

Variablen stellen wir durch nullstellige Funktionszeichen dar. Nullstellige Funktionszeichen werden in Prolog auch als *Atome* bezeichnet. Damit gilt

$$c \in \text{Expr} \quad \text{für jedes Prolog-Atom } c.$$

2. Zahlen sind arithmetische Ausdrücke.

Sowohl die ganzen Zahlen als auch die reellen Zahlen sind Bestandteil der Sprache *Prolog* und können damit durch sich selbst dargestellt werden:

$$n \in Expr \quad \text{für alle } n \in \mathbb{Z},$$

$$r \in Expr \quad \text{für alle } r \in \mathbb{R}.$$

3. Das Negative eines arithmetischen Ausdrucks ist ein arithmetischer Ausdruck. In *Prolog* kann das Negative durch den unären Operator “-” dargestellt werden, also haben wir

$$-t \in Expr \quad \text{falls } t \in Expr.$$

4. Die Summe, die Differenz, das Produkt, und der Quotient zweier arithmetischen Ausdrücke ist ein arithmetischer Ausdruck. In *Prolog* können Summe, Differenz, Produkt und Quotient respektive durch die binären Operatoren “+”, “-”, “*” und “/” dargestellt werden, also setzen wir

$$s + t \in Expr \quad \text{falls } s, t \in Expr.$$

$$s - t \in Expr \quad \text{falls } s, t \in Expr.$$

$$s * t \in Expr \quad \text{falls } s, t \in Expr.$$

$$s / t \in Expr \quad \text{falls } s, t \in Expr.$$

5. Die Potenz zweier arithmetischer Ausdrücke ist ein arithmetischer Ausdruck. In *Prolog* kann die Potenz durch den binären Operator “**” dargestellt werden, also setzen wir

$$s ** t \in Expr \quad \text{falls } s, t \in Expr.$$

6. Bei der Behandlung spezieller Funktionen beschränken wir uns auf die Exponential-Funktion und den natürlichen Logarithmus:

$$\exp(t) \in Expr \quad \text{falls } t \in Expr,$$

$$\ln(t) \in Expr \quad \text{falls } t \in Expr.$$

Aufstellen der bedingten Gleichungen Den Wert von $\text{diff}(t, x)$ definieren wir nun durch Induktion nach dem Aufbau des arithmetischen Ausdrucks t .

1. Bei der Ableitung einer Variablen müssen wir unterscheiden, ob wir die Variable nach sich selbst oder nach einer anderen Variablen ableiten.

- (a) Die Ableitung einer Variablen nach sich selbst gibt den Wert 1:

$$y = x \rightarrow \frac{dy}{dx} = 1.$$

Also haben wir

$$y = x \rightarrow \text{diff}(x, x) = 1.$$

- (b) Die Ableitung einer Variablen y nach einer anderen Variablen x ergibt den Wert 0:

$$y \neq x \rightarrow \frac{dy}{dx} = 0$$

Also haben wir

$$y \neq x \rightarrow \mathbf{diff}(x, x) = 1.$$

2. Die Ableitung einer Zahl n ergibt 0:

$$\frac{dn}{dx} = 0.$$

Damit haben wir

$$\mathbf{diff}(n, x) = 0.$$

3. Die Ableitung eines Ausdrucks mit negativen Vorzeichen ist durch

$$\frac{d}{dx}(-f) = -\frac{df}{dx}$$

gegeben. Die rekursive Gleichung lautet

$$\mathbf{diff}(-f, x) = -\mathbf{diff}(f, x).$$

4. Die Ableitung einer Summe ergibt sich als Summe der Ableitungen der Summanden:

$$\frac{d}{dx}(f + g) = \frac{df}{dx} + \frac{dg}{dx}$$

Als Gleichung schreibt sich dies

$$\mathbf{diff}(f + g, x) = \mathbf{diff}(f, x) + \mathbf{diff}(g, x).$$

5. Die Ableitung einer Differenz ergibt sich als Differenz der Ableitung der Operanden:

$$\frac{d}{dx}(f - g) = \frac{df}{dx} - \frac{dg}{dx}$$

Als Gleichung schreibt sich dies

$$\mathbf{diff}(f - g, x) = \mathbf{diff}(f, x) - \mathbf{diff}(g, x).$$

6. Die Ableitung eines Produktes wird durch die Produkt-Regel beschrieben:

$$\frac{d}{dx}(f * g) = \frac{df}{dx} * g + f * \frac{dg}{dx}.$$

Dies führt auf die Gleichung

$$\mathbf{diff}(f * g, x) = \mathbf{diff}(f, x) * g + f * \mathbf{diff}(g, x).$$

7. Die Ableitung eines Quotienten wird durch die Quotienten-Regel beschrieben:

$$\frac{d}{dx}(f/g) = \frac{\frac{df}{dx} * g - f * \frac{dg}{dx}}{g * g}.$$

Dies führt auf die Gleichung

$$\mathbf{diff}(f/g, x) = (\mathbf{diff}(f, x) * g - f * \mathbf{diff}(g, x)) / (g * g).$$

8. Zur Ableitung eines Ausdrucks der Form $f ** g$ verwenden wir die folgende Gleichung:

$$f ** g = \exp(g * \ln(f)).$$

Das führt auf die Gleichung

$$\text{diff}(f ** g, x) = \text{diff}(\exp(g * \ln(f)), x).$$

9. Bei der Ableitung der Exponential-Funktion benötigen wir die Ketten-Regel:

$$\frac{d}{dx} \exp(f) = \frac{df}{dx} * \exp(f).$$

Das führt auf die Gleichung

$$\text{diff}(\exp(f), x) = \text{diff}(f, x) * \exp(f).$$

10. Für die Ableitung des natürlichen Logarithmus finden wir unter Berücksichtigung der Ketten-Regel

$$\frac{d}{dx} \ln(f) = \frac{1}{f} * \frac{df}{dx}.$$

Das führt auf die Gleichung

$$\text{diff}(\exp(f), x) = \text{diff}(f, x) / f.$$

Implementierung in Prolog Abbildung 5.9 zeigt die Implementierung in *Prolog*. An Stelle der zweistelligen Funktion *diff()* haben wir nun ein dreistelliges Prädikat **diff/3**, dessen letztes Argument das Ergebnis berechnet. Wir diskutieren die einzelnen Klauseln.

- Die beiden Klauseln in den Zeilen 3 – 9 zeigen, wie eine Variable differenziert werden kann. Das Prädikat **atom(X)** prüft, ob *X* ein nullstelliges Funktions-Zeichen ist. Solche Funktions-Zeichen werden im *Prolog*-Jargon auch als *Atome* bezeichnet. Wir prüfen also in Zeile 4 und 8, ob es sich bei dem abzuleitenden Ausdruck um eine Variable handelt. Anschließend überprüfen wir in den Zeilen 5 bzw. 9, ob diese Variable mit der Variablen, nach der differenziert werden soll, übereinstimmt oder nicht.
- In der Klausel in den Zeilen 11 – 12 behandeln wir den Fall, dass es sich bei dem zu differenzierenden Ausdruck um eine Zahl handelt. Um dies überprüfen zu können, verwenden wir das Prädikat **number(X)**, das überprüft, ob das Argument *X* eine Zahl ist.
In dieser Klausel haben wir die Variable, nach der abgeleitet werden soll, mit “*X*” bezeichnet. Der Grund ist, dass das *Prolog*-System für Variablen, die in einer Klausel nur einmal vorkommen, eine Warnung ausgibt. Diese Warnung kann vermieden werden, wenn vorne an den Variablennamen ein Unterstrichs “_” angefügt wird.
- Am Beispiel der Ableitung des Ausdrucks $-f$ zeigen wir, wie rekursive Gleichungen in *Prolog* umgesetzt werden können. Die Gleichung, die in den Zeilen 14 – 15 umgesetzt wird, lautet

$$\text{diff}(-f, x) = -\text{diff}(f, x).$$

Um den Ausdruck $-F$ nach *x* zu differenzieren, müssen wir zunächst den Ausdruck *F* nach *x* ableiten. Das passiert in Zeile 15 und liefert das Ergebnis *Fs*. Das Endergebnis erhalten wir dadurch, dass wir vor *Fs* ein Minuszeichen setzen.

- Die restlichen Klausel setzen die oben gefundenen bedingten Gleichungen unmittelbar um und werden daher hier nicht weiter diskutiert.

```

1  % diff( +Expr, +Atom, -Expr).
2
3  diff(F, X, 1) :-
4      atom(F),
5      F == X.
6
7  diff(F, X, 0) :-
8      atom(F),
9      F \== X.
10
11 diff(N, _X, 0) :-
12     number(N).
13
14 diff(-F, X, -Fs) :-
15     diff(F, X, Fs).
16
17 diff(F + G, X, Fs + Gs) :-
18     diff(F, X, Fs),
19     diff(G, X, Gs).
20
21 diff(F - G, X, Fs - Gs) :-
22     diff(F, X, Fs),
23     diff(G, X, Gs).
24
25 diff(F * G, X, Fs * G + F * Gs) :-
26     diff(F, X, Fs),
27     diff(G, X, Gs).
28
29 diff(F / G, X, (Fs * G - F * Gs) / (G * G)) :-
30     diff(F, X, Fs),
31     diff(G, X, Gs).
32
33 diff( F ** G, X, D ) :-
34     diff( exp(F * ln(G)), X, D ).
35
36 diff( exp(F), X, Fs * exp(F) ) :-
37     diff(F, X, Fs).
38
39 diff( ln(F), X, Fs / F ) :-
40     diff(F, X, Fs).

```

Figure 5.9: Ein Programm zum symbolischen Differenzieren

5.4 Negation in *Prolog*

In diesem Abschnitt besprechen wir die Implementierung des Negations-Operators in *Prolog*. Wir zeigen zunächst an Hand eines einfachen Beispiels die Verwendung dieses Operators, besprechen dann seine Semantik und zeigen abschließend, in welchen Fällen die Verwendung des Negations-Operators problematisch ist.

5.4.1 Berechnung der Differenz zweier Listen

In *Prolog* wird der Negations-Operator als “\+” geschrieben. Wir erläutern die Verwendung dieses Operators am Beispiel einer Funktion, die die Differenz zweier Mengen berechnen soll, wobei die Mengen durch Listen dargestellt werden. Wir werden die Funktion

$$\text{difference} : \text{List}(\text{Number}) \times \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number})$$

durch bedingte Gleichungen spezifizieren. Der Ausdruck

$$\text{difference}(l_1, l_2)$$

berechnet die Liste aller der Elemente aus l_1 , die nicht Elemente der Liste l_2 sind. In SETL könnten wir diese Funktion wie in Abbildung 5.10 gezeigt implementieren.

```
1  procedure difference(l1, l2);
2      return [ x in l1 | not x in l2 ];
3  end difference;
```

Figure 5.10: Implementierung der Prozedur **difference** in SETL.

In *Prolog* erfolgt die Implementierung dieser Funktion durch Rekursion im ersten Argument. Dazu stellen wir zunächst bedingte Gleichungen auf:

1. $\text{difference}([], l) = []$.
2. $\neg \text{member}(h, l) \rightarrow \text{difference}([h|t], l) = [h|\text{difference}(t, l)]$,
denn wenn das Element h in der Liste l nicht vorkommt, so bleibt dieses Element im Ergebnis erhalten.
3. $\text{member}(h, l) \rightarrow \text{difference}([h|t], l) = \text{difference}(t, l)$.

```
1  % difference( +List(Number), +List(Number), -List(Number) ).
2  difference( [], _L, [] ).
3
4  difference( [ H | T ], L, [ H | R ] ) :-
5      \+ member( H, L ),
6      difference( T, L, R ).
7
8  difference( [ H | T ], L, R ) :-
9      member( H, L ),
10     difference( T, L, R ).
```

Figure 5.11: Berechnung der Differenz zweier Listen

5.4.2 Semantik des Negations-Operators in Prolog

Es bleibt zu klären, wie das *Prolog*-System eine Anfrage der Form

`\+ A`

beantwortet, wie also der `not`-Operator implementiert ist.

1. Zunächst versucht das System, die Anfrage “A” zu beantworten.
2. Falls die Beantwortung der Anfrage “A” scheitert, ist die Beantwortung der Anfrage “\+ A” erfolgreich. In diesem Fall werden keine Variablen instanziiert.
3. Falls die Beantwortung der Anfrage “A” erfolgreich ist, so scheitert die Beantwortung der Anfrage “\+ A”.

Wichtig ist zu sehen, dass bei der Beantwortung einer negierten Anfrage in keinem Fall Variablen instanziiert werden. Eine negierte Anfrage

`\+ A`

funktioniert daher nur dann wie erwartet, wenn die Anfrage *A* keine Variablen mehr enthält. Zur Illustration betrachten wir das Programm in Abbildung 5.12. Versuchen wir mit diesem Programm die Anfrage

`smart1(X)`

zu beantworten, so wird diese Anfrage reduziert zu der Anfrage

`\+ roemer(X), gallier(X).`

Um die Anfrage “\+ roemer(X)” zu beantworten, versucht das *Prolog*-System rekursiv, die Anfrage “roemer(X)” zu beantworten. Dies gelingt und die Variable *X* wird dabei an den Wert “caesar” gebunden. Da die Beantwortung der Anfrage “roemer(X)” gelingt, scheitert die Anfrage

`\+ roemer(X)`

und damit gibt es auch auf die ursprüngliche Anfrage “smart1(X)” keine Antwort.

```
1  gallier(miraculix).
2
3  roemer(caesar).
4
5  smart1(X) :- \+ roemer(X), gallier(X).
6
7  smart2(X) :- gallier(X), \+ roemer(X).
```

Figure 5.12: Probleme mit der Negation

Wenn wir voraussetzen, dass das Programm das Prädikate `roemer/1` vollständig beschreibt, dann ist dieses Verhalten nicht korrekt, denn dann folgt die Konjunktion

`¬roemer(miraculix) ∧ gallier(miraculix)`

ja aus unserem Programm. Wenn der dem *Prolog*-System zu Grunde liegende automatische Beweiser anders implementiert wäre, dann könnte er dies auch erkennen. Wir können uns in diesem Beispiel damit behelfen, dass wir die Reihenfolge der Formeln im Rumpf umdrehen, so wie dies bei der Klausel in Zeile 7 der Abbildung 5.12 geschehen ist. Die Anfrage

`smart2(X)`

liefert für *X* den Wert “miraculix”. Die zweite Anfrage funktioniert, weil zu dem Zeitpunkt, an dem die negierte Anfrage “\+ roemer(X)” aufgerufen wird, ist die Variable *X* bereits an den Wert

`miraculix` gebunden und die Anfrage “`roemer(miraculix)`” scheitert. Generell sollte in *Prolog*-Programmen der Negations-Operator “`\+`” nur auf solche Prädikate angewendet werden, die zum Zeitpunkt keine freien Variablen mehr enthalten.

5.5 Die Tiefen-Suche in *Prolog*

Wenn das *Prolog*-System eine Anfrage beantwortet, wird dabei als Such-Strategie die sogenannte Tiefen-Suche (engl. *depth first search*) angewendet. Wir wollen diese Strategie nun an einem weiteren Beispiel verdeutlichen. Wir implementieren dazu ein *Prolog*-Programm mit dessen Hilfe es möglich ist, in einem Graphen eine Verbindung von einem gegebenen Start-Knoten zu einem Ziel-Knoten zu finden. Als Beispiel betrachten wir den Graphen in Abbildung 5.13. Die Kanten können durch ein *Prolog*-Prädikat `edge/2` wie folgt dargestellt werden:

```

1  edge(a, b).
2  edge(a, c).
3  edge(b, e).
4  edge(e, f).
5  edge(c, f).
```

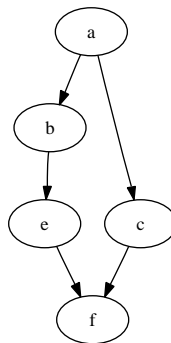


Figure 5.13: Ein einfacher Graph ohne Zykeln

Wir wollen nun ein *Prolog*-Programm entwickeln, mit dem es möglich ist für zwei vorgegebene Knoten x und y zu entscheiden, ob es einen Weg von x nach y gibt. Außerdem soll dieser Weg dann als Liste von Knoten berechnet werden. Unser erster Ansatz besteht aus dem Programm, das in Abbildung 5.14 gezeigt ist. Die Idee ist, dass der Aufruf

`find_path(Start, Goal, Path)`

einen Pfad *Path* berechnet, der von *Start* nach *Goal* führt. Wir diskutieren die Implementierung.

```

1  % find_path( +Point, +Point, -List(Point) ).
2  find_path( X, X, [ X ] ).
3
4  find_path( X, Z, [ X | Path ] ) :-
5      edge( X, Y ),
6      find_path( Y, Z, Path ).
```

Figure 5.14: Berechnung von Pfaden in einem Graphen

1. Die erste Klausel sagt aus, dass es trivialerweise einen Pfad von X nach X gibt. Dieser Pfad enthält genau den Knoten X .
2. Die zweite Klausel sagt aus, dass es einen Weg von X nach Z gibt, wenn es zunächst eine direkte Verbindung von X zu einem Knoten Y gibt und wenn es dann von diesem Knoten Y eine Verbindung zu dem Knoten Z gibt. Wir erhalten den Pfad, der von X nach Z führt, dadurch, dass wir vorne an den Pfad, der von Y nach Z führt, den Knoten X anfügen.

Stellen wir an das *Prolog*-System die Anfrage `find_path(a,f,P)`, so erhalten wir die Antwort

```

1  ?- find_path(a,f,P).
2
3  P = [a, b, e, f] ;
4  P = [a, c, f] ;
5  No

```

Durch Backtracking werden also alle möglichen Wege von a nach f gefunden. Als nächstes testen wir das Programm mit dem in Abbildung 5.15 gezeigten Graphen. Diesen Graphen stellen wir wie folgt in *Prolog* dar:

```

1  edge(a, b).
2  edge(a, c).
3  edge(b, e).
4  edge(e, a).
5  edge(e, f).
6  edge(c, f).

```

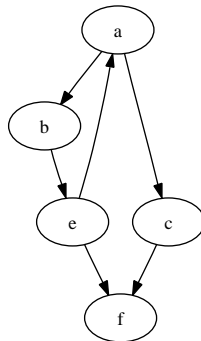


Figure 5.15: Ein Graph mit einem Zykel

Jetzt erhalten wir auf die Anfrage `find_path(a,f,P)` die Antwort

```

1  ?- find_path(a,f,P).
2  ERROR: Out of local stack

```

Die Ursache ist schnell gefunden.

1. Wir starten mit der Anfrage
`find_path(a,f,P).`
2. Nach Unifikation mit der zweiten Klausel haben wir die Anfrage reduziert auf
`edge(a, Y1), find_path(Y1, f, P1).`
3. Nach Unifikation mit dem Fakt `edge(a,b)` haben wir die neue Anfrage
`find_path(b, f, P1).`
4. Nach Unifikation mit der zweiten Klausel haben wir die Anfrage reduziert auf
`edge(b, Y2), find_path(Y2, f, P2).`
5. Nach Unifikation mit dem Fakt `edge(b,e)` haben wir die neue Anfrage
`find_path(e, f, P2).`
6. Nach Unifikation mit der zweiten Klausel haben wir die Anfrage reduziert auf
`edge(e, Y3), find_path(Y3, f, P3).`
7. Nach Unifikation mit dem Fakt `edge(e,a)` haben wir die neue Anfrage
`find_path(a, f, P3).`

Die Anfrage “`find_path(a, f, P3)`” unterscheidet sich aber von der ursprünglichen Anfrage “`find_path(a,f,P)`” nur durch den Namen der Variablen. Wenn wir jetzt weiterrechnen würden, würde sich die Rechnung nur wiederholen, ohne dass wir vorwärts kommen. Das Problem ist, das *Prolog* immer die erste Klausel nimmt, die paßt. Wenn später die Reduktion der Anfrage scheitert, wird zwar nach Backtracking die nächste Klausel ausprobiert, aber wenn das Programm in eine Endlos-Schleife läuft, dann gibt es eben kein Backtracking, denn das Programm weiß ja nicht, dass es in einer Endlos-Schleife ist.

Es ist aber leicht das Programm so umzuschreiben, dass keine Endlos-Schleife mehr auftreten kann. Die Idee ist, dass wir uns merken, welche Knoten wir bereits besucht haben und diese nicht mehr auswählen. In diesem Sinne implementieren wir nun ein Prädikat `find_path/4`. Die Idee ist, dass der Aufruf

```
find_path(Start, Goal, Visited, Path)
```

einen Pfad berechnet, der von *Start* nach *Goal* führt und der zusätzlich keine Knoten benutzt, die bereits in der Liste *Visited* aufgeführt sind. Diese Liste füllen wir bei den rekursiven Aufrufen nach und nach mit den Knoten an, die wir bereits besucht haben. Mit Hilfe dieser Liste vermeiden wir es, einen Knoten zweimal zu besuchen. Abbildung 5.16 zeigt die Implementierung.

```

1  % find_path( +Point, +Point, +List(Point), -List(Point) )
2
3  find_path( X, X, _Visited, [ X ] ).
4
5  find_path( X, Z, Visited, [ X | Path ] ) :-
6      edge( X, Y ),
7      \+ member( Y, Visited ),
8      find_path( Y, Z, [ Y | Visited ], Path ).

```

Figure 5.16: Berechnung von Pfaden in zyklischen Graphen

1. In der ersten Klausel spielt das zusätzliche Argument noch keine Rolle, denn wenn wir das Ziel erreicht haben, ist es uns egal, welche Knoten wir schon besucht haben.

2. In der zweiten Klausel überprüfen wir in Zeile 7, ob der Knoten *Y* in der Liste *Visited*, die die Knoten enthält, die bereits besucht wurden, auftritt. Nur wenn dies nicht der Fall ist, versuchen wir rekursiv von *Y* einen Pfad nach *Z* zu finden. Bei dem rekursiven Aufruf erweitern wir die Liste *Visited* um den Knoten *Y*, den diesen Knoten wollen wir in Zukunft ebenfalls vermeiden.

Mit dieser Implementierung ist es jetzt möglich, auch in dem zweiten Graphen einen Weg von *a* nach *f* zu suchen, wir erhalten

```
1  ?- find_path(a,f,[a],P).
2  P = [a, b, e, f] ;
3  P = [a, c, f] ;
4  No
```

5.5.1 Missionare und Kannibalen

Als spielerische Anwendung zeigen wir nun, wie sich mit Hilfe des oben definierten Prädikats `find_path/4` bestimmte Rätsel lösen lassen und lösen exemplarisch das folgende Rätsel:

Drei Missionare und drei Kannibalen wollen zusammen einen Fluß überqueren. Sie haben nur ein Boot, indem maximal zwei Passagiere fahren können. Sowohl die Kannibalen als auch die Missionare können rudern. Die Kannibalen sind hungrig, wenn die Missionare an einem der Ufer in der Unterzahl sind, haben sie ein Problem. Die Aufgabe besteht darin, einen Fahrplan zu erstellen, so dass hinterher alle das andere Ufer erreichen und die Missionare zwischendurch kein Problem haben.

Die Idee ist, das Rätsel, durch einen Graphen zu modellieren. Die Knoten dieses Graphen sind dann die Situationen, die während des Übersetzens auftreten. Wir repräsentieren diese Situationen durch Terme der Form

`side(M, K, B)`.

Ein solcher Term repräsentiert eine Situation, bei der auf der linken Seite des Ufers M Missionare, K Kannibalen und B Boote sind. Unsere Aufgabe besteht nun darin, das Prädikat `edge/2` so zu implementieren, dass

`edge(side(M1, K1, B1), side(M2, K2, B2))`

genau dann wahr ist, wenn die Situation `side(M1, K1, B1)` durch eine Boots-Überfahrt in die Situation `side(M2, K2, B2)` überführt werden kann und wenn zusätzlich die Missionare in der neuen Situation kein Problem bekommen. Abbildung 5.18 auf Seite 165 zeigt ein *Prolog*-Programm, was das Rätsel löst. Den von diesem Programm berechneten Fahrplan finden Sie in Abbildung 5.17 auf Seite 164. Wir diskutieren dieses Programm nun Zeile für Zeile.

1	MMM	KKK	B	~~~~~			
2				> KK >			
3	MMM	K		~~~~~	B	KK	
4				< K <			
5	MMM	KK	B	~~~~~		K	
6				> KK >			
7	MMM			~~~~~	B	KKK	
8				< K <			
9	MMM	K	B	~~~~~		KK	
10				> MM >			
11	M	K		~~~~~	B	KK	MM
12				< M K <			
13	MM	KK	B	~~~~~		K	M
14				> MM >			
15		KK		~~~~~	B	K	MMM
16				< K <			
17		KKK	B	~~~~~			MMM
18				> KK >			
19		K		~~~~~	B	KK	MMM
20				< K <			
21		KK	B	~~~~~		K	MMM
22				> KK >			
23				~~~~~	B	KKK	MMM

Figure 5.17: Fahrplan für Missionare und Kannibalen

```

1  solve :-
2      find_path( side(3,3,1), side(0,0,0), [ side(3,3,1) ], Path ),
3      nl, write('Lösung:') , nl, nl,
4      print_path(Path).
5
6  % edge( +Point, -Point ).
7  % This clause describes rowing from the left side to the right side.
8  edge( side( M, K, 1 ), side( MN, KN, 0 ) ) :-
9      between( 0, M, MB ),      % MB missionaries in the boat
10     between( 0, K, KB ),      % KB cannibals in the boat
11     MB + KB >= 1,             % boat must not be empty
12     MB + KB <= 2,             % no more than two passengers
13     MN is M - MB,             % missionaries left on the left side
14     KN is K - KB,             % cannibals left on the left
15     \+ problem( MN, KN ).     % no problem may occur
16
17 % This clause describes rowing from the right side to the left side.
18 edge( side( M, K, 0 ), side( MN, KN, 1 ) ) :-
19     otherSide( M, K, MR, KR ),
20     edge( side( MR, KR, 1 ), side( MRN, KRN, 0 ) ),
21     otherSide( MRN, KRN, MN, KN ).
22
23 % otherSide( +Number, +Number, -Number, -Number ).
24 otherSide( M, K, M_Other, K_Other ) :-
25     M_Other is 3 - M,
26     K_Other is 3 - K.
27
28 % problem( +Number, +Number ).
29 problem(M, K) :-
30     problemSide(M, K).
31
32 problem(M, K) :-
33     otherSide( M, K, M_Other, K_Other ),
34     problemSide(M_Other, K_Other).
35
36 % problem( +Number, +Number ).
37 problemSide(Missionare, Kannibalen) :-
38     Missionare > 0,
39     Missionare < Kannibalen.
40
41 % find_path( +Point, +Point, +List(Point), -List(Point) )
42 find_path( X, X, _Visited, [ X ] ).
43
44 find_path( X, Z, Visited, [ X | Path ] ) :-
45     edge( X, Y ),
46     \+ member( Y, Visited ),
47     find_path( Y, Z, [ Y | Visited ], Path ).

```

Figure 5.18: Missionare und Kannibalen

1. Wir beginnen mit dem Hilfs-Prädikat `otherSide/4`, das in den Zeilen 27 – 29 implementiert ist. Für eine vorgegebene Situation `side(M, K, B)` berechnet der Aufruf

`otherSide(side(M, K, B), $OtherSide$)`

einen Term, der die Situation am gegenüberliegenden Ufer beschreibt. Wenn an einen Ufer M Missionare sind, so sind am anderen Ufer die restlichen Missionare und da es insgesamt 3 Missionare gibt, sind das $3 - M$. Die Anzahl der Kannibalen am gegenüberliegenden Ufer wird analog berechnet.

2. Das Prädikat `problem/2` in den Zeilen 32 – 37 überprüft, ob es bei einer vorgegeben Anzahl von Missionaren und Kannibalen zu einem Problem kommt. Da das Problem entweder am linken oder am rechten Ufer auftreten kann, besteht die Implementierung aus zwei Klauseln. Die erste Klausel prüft, ob es auf der Seite, an der M Missionare und K Kannibalen sind, zum Problem kommt. Die zweite Klausel überprüft, ob es auf dem gegenüberliegenden Ufer zu einem Problem kommt. Als Hilfs-Prädikat verwenden wir hier das Prädikat `problemSide/2`. Dieses Prädikat ist in Zeile 40 implementiert und überprüft die Situation an einer Seite: Falls sich auf einer Seite M Missionare und K Kannibalen befinden, so gibt es dann ein Problem, wenn die Zahl M von 0 verschieden ist und wenn zusätzlich $M < K$ ist.

3. Bei der Implementierung des Prädikats `edge/2` verwenden wir in den Zeilen 12 und 13 das Prädikat `between/3`, das in dem *SWI-Prolog*-System vordefiniert ist. Beim Aufruf

`between($Low, High, N$)`

sind Low und $High$ ganze Zahlen mit $Low \leq High$. Der Aufruf instantiiert die Variable N nacheinander mit den Zahlen

$Low, Low + 1, Low + 2, \dots, High$.

Beispielsweise gibt die Anfrage

`between(1,3,N), write(N), nl, fail.`

nacheinander die Zahlen 1, 2 und 3 am Bildschirm aus.

4. Die Implementierung des Prädikats `edge/2` besteht aus zwei Klauseln. In der ersten Klausel betrachten wir den Fall, dass das Boot am linken Ufer ist. In der Zeilen 12 generieren wir die Zahl der Missionare MB , die im Boot übersetzen sollen. Diese Zahl MB ist durch M beschränkt, denn es können nur die Missionare übersetzen, die sich am linken Ufer befinden. Daher benutzen wir das Prädikat `between/3` um eine Zahl zwischen 0 und M zu erzeugen. Analog generieren wir in Zeile 13 die Zahl KB der Kannibalen, die im Boot übersetzen. In Zeile 14 testen wir, dass es mindestens einen Passagier gibt, der mit dem Boot übersetzt und in Zeile 15 testen wir, dass es höchstens zwei Passagiere sind. In Zeile 16 und 17 berechnen wir die Zahl MN der Missionare und die Zahl KN der Kannibalen, die nach der Überfahrt auf dem linken Ufer verbleiben und testen dann in Zeile 18, dass es für diese Zahlen kein Problem gibt.

Die zweite Klausel befaßt sich mit dem Fall, dass das Boot am rechten Ufer liegt. Wir hätten diese Klausel mit *Copy & Paste* aus der vorhergehenden Klausel erzeugen können, aber es ist eleganter, diesen Fall auf den vorhergehenden Fall zurück zu führen. Da da Boot nun auf der rechten Seite liegt, berechnen wir daher in Zeile 22 die Zahl MR der Missionare auf der rechten Seite und die Zahl KR der Kannibalen auf der rechten Seite. Dann untersuchen wir die Situation `side($MR, KR, 1$)`, bei der MR Missionare und KR Kannibalen am linken Ufer stehen. Wenn diese so übersetzen können, dass nachher MRN Missionare und KRN Kannibalen am linken Ufer stehen, dann können wir in Zeile 24 berechnen, wieviele Missionare und Kannibalen sich dann am gegenüberliegenden Ufer befinden.

5. In den Zeilen 1 – 6 definieren wir nun das Prädikat `solve/0`, dessen Aufruf das Problem löst. Dazu wird zunächst das Prädikat `find_path/4` mit dem Start-Knoten `side(3,3,1)` und dem Ziel-Knoten `side(0,0,0)` aufgerufen. Der berechnete Pfad wird dann ausgegeben mit dem Prädikat `print_path/1`, dessen Implementierung wir hier aus Platzgründen nicht angegeben haben.

5.6 Das 8-Damen-Problem in *Prolog*

In diesem Abschnitt zeigen wir, wie sich das 8-Damen-Problem in *Prolog* lösen läßt. Wir beginnen mit einer naiven Implementierung, bei der es zunächst darum geht, ein möglichst einfaches Programm zu erstellen, bei dem die Effizienz noch keine Rolle spielt. Die Grundidee bei dem Programm ist es, zunächst alle möglichen Lösungen zu generieren und anschließend zu testen, ob es sich tatsächlich um eine Lösung handelt. Dazu müssen wir zunächst entscheiden, wie wir eine mögliche Lösung des n -Damen-Problems in Prolog repräsentieren wollen. Wir entscheiden uns dafür, eine Lösung als eine Liste der Länge n darzustellen, wobei die einzelnen Elemente Zahlen aus der Menge $\{1, \dots, n\}$ sind. Diese Elemente interpretieren wir als Positionen der Damen. Konkret vereinbaren wir, dass eine Liste der Form

$$[c_1, c_2, \dots, c_n]$$

spezifiziert, dass die Dame, die sich in der ersten Zeile befindet, in der Spalte c_1 steht, die Dame aus der zweiten Zeile steht in der Spalte c_2 und allgemein steht die Dame in der i -ten Zeile in der Spalte c_i . Beispielsweise repräsentieren wir die in Abbildung 5.19 gezeigte Lösung des 8-Damen-Problems durch die Liste

$$[4, 8, 1, 3, 6, 2, 7, 5].$$

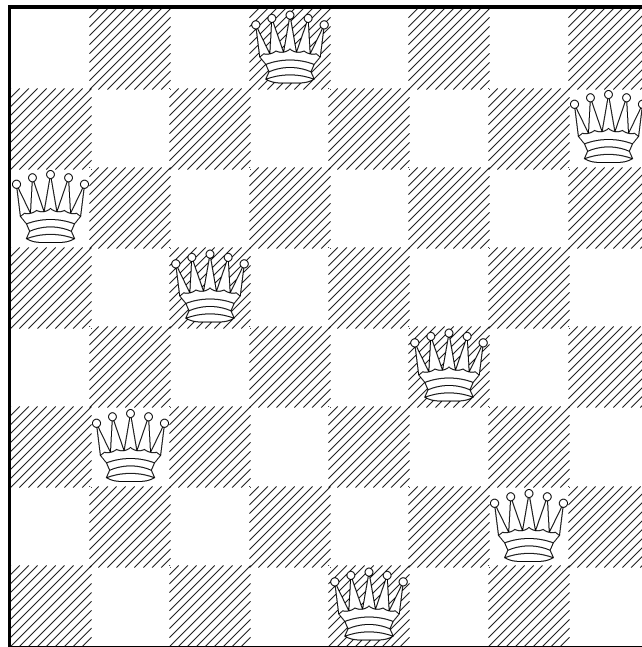


Figure 5.19: A solution of the eight queens puzzle.

Abbildung 5.20 zeigt nun unser Programm zur Lösung des n -Damen-Problems.

1. Das Prädikat `queens/2` bekommt als erstes Argument die Anzahl N der zu platzierenden Damen übergeben. Es ruft zunächst das Prädikat `generate_list/3` auf, dass alle möglichen Listen der Länge N erzeugt, deren Elemente Zahlen aus der Menge $\{1, \dots, N\}$ sind. Abschließend überprüft das Prädikat `is_solution/1`, ob es sich bei der erzeugten Liste tatsächlich um eine Lösung des N -Damen-Problems handelt.
2. Der Aufruf `generate_list(L, N, T)` setzt voraus, dass die Argumente L und N mit natürlichen Zahlen instanziiert sind. Hierbei gibt L die Länge der zu erzeugenden Liste an und N spezifiziert, dass die einzelnen Elemente der zu erzeugenden Liste Elemente der Menge $\{1, \dots, N\}$

sein sollen. Die Variable T enthält dann nach einem erfolgreichen Aufruf des Prädikats die erzeugte Liste.

Die Implementierung erfolgt durch Rekursion in der Länge L der zu erzeugenden Liste. Solange diese Länge positiv ist, wird zunächst durch den Aufruf von **between/3** eine Zahl C aus der Menge $\{1, \dots, N\}$ erzeugt. Diese Zahl ist später das erste Element der erzeugten Liste. Anschließend wird durch einen rekursiven Aufruf eine Liste der Länge $L - 1$ erzeugt, der im Ergebnis die vorher erzeugte Zahl C vorangestellt wird.

3. Das Prädikat **is_solution(L)** überprüft, ob die als Argument übergebene Liste L tatsächlich das n -Damen-Problem löst. Diese Überprüfung erfolgt rekursiv: Zunächst wird überprüft, ob die Dame aus der ersten Zeile eine der anderen Damen angreifen kann. Abschließend wird rekursiv überprüft, ob auch die Damen in den weiteren Zeilen sicher stehen.
4. Das Prädikat **attack(C, X, L)** erhält drei Argumente. Das erste Argument spezifiziert eine Zeile, in der eine Dame steht. Das zweite Argument X spezifiziert, wieviele Zeilen Abstand zwischen der durch C spezifizierten Dame und der ersten Dame der Liste L liegen. Falls die durch C spezifizierte Dame in der Zeile r_C steht und die Dame, die durch das erste Element der Liste L spezifiziert wird, in der Zeile r_1 steht, so gilt

$$X = r_1 - r_C.$$

Das dritte Argument L ist eine Liste von Positionen von Damen. Die einzelnen Klauseln der Implementierung sind jetzt wie folgt zu verstehen:

- (a) Falls die durch C spezifizierte Dame in der selben Spalte steht wie die erste Dame der Liste L , dann kann die durch C spezifizierte Dame offenbar diese Dame angreifen und der Aufruf von **attack/3** liefert **true**.
- (b) Die nächste Klausel behandelt den Fall, dass die erste Dame der Liste L in der selben fallenden Diagonalen steht wie die durch C spezifizierte Dame. Um die Implementierung zu verstehen ist nur zu beachten, dass die Spalten der fallenden Diagonale, die von C ausgeht, sich dadurch ergeben, dass wir die Spalte für jede neue Zeile um 1 hochzählen müssen. Das erklärt den Term $C + X$.
- (c) Analog behandelt die dritte Klausel den Fall, dass die erste Dame der Liste L in der selben steigenden Diagonalen steht wie die durch C spezifizierte Dame. Für eine steigende Diagonale müssen wir die Spalte für jede weitere Zeile um 1 dekrementieren. Das erklärt den Term $C - X$.
- (d) Die letzte Klausel prüft schließlich rekursiv, ob die Dame in Spalte C eine der restlichen Damen angreifen kann.

Das Programm ist ineffizient, weil die Tests und die Überprüfungen zu stark getrennt sind. So wird bei der Lösung des 8-Damen-Problems beispielsweise zunächst die Liste $[1, 1, 1, 1, 1, 1, 1, 1]$ erzeugt und geprüft, ob diese Liste das Problem löst. Nachdem dieser Versuch scheitert, wird anschließend die Liste $[1, 1, 1, 1, 1, 1, 1, 2]$ erzeugt, obwohl eigentlich schon klar ist, dass diese Liste keine Lösung bringen kann, denn schon die ersten beiden Positionen der ursprünglichen Liste verhindern eine Lösung. Die wesentliche Idee die Effizienz dieses Programms zu steigern besteht darin, so früh wie möglich zu testen ob eine bestimmte Position überhaupt in Frage kommt. In dem in Abbildung 5.21 gezeigten Programm wird diese Idee dadurch umgesetzt, dass wir ein dreistelliges Prädikat **queens/3** implementieren, das in der Form

queens(N, Positioned, Solution)

aufgerufen wird. Hier bezeichnet N die Anzahl der zu positionierenden Damen, während **Positioned** eine Liste von Damen angibt, die bereits gesetzt sind und von der bekannt ist, dass die dort gesetzten Damen sich nicht gegenseitig angreifen können. Die Implementierung dieses Prädikats besteht aus zwei Klauseln.

```

1  % queens(+Number, -List(Number)).
2  queens(N, S) :-
3      generate_list(N, N, S),
4      is_solution(S).
5
6  % generate_list(+Number, +Number, -List(Number)).
7  generate_list(L, N, [ C | T ]) :-
8      L > 0,
9      between(1, N, C),
10     L1 is L - 1,
11     generate_list(L1, N, T).
12
13 generate_list(0, _, []).
14
15 % is_solution(+List(Number)).
16 is_solution([ H | T ]) :-
17     \+ attack(H, 1, T),
18     is_solution(T).
19
20 is_solution([]).
21
22 % attack(+Number, +Number, +List(Number)).
23 attack(C, _, [ C | _T ]). % same column
24
25 attack(C, X, [ H | _T ]) :-
26     H is C + X.
27
28 attack(C, X, [ H | _T ]) :-
29     H is C - X.
30
31 attack(C, X, [ _H | T ]) :-
32     X1 is X + 1,
33     attack(C, X1, T).

```

Figure 5.20: Ein naives Programm zur Lösung des n -Damen-Problems.

1. In der ersten Klausel der Implementierung von `queens/3` überprüfen wir zunächst, ob überhaupt noch weitere Damen zu setzen sind. Ist dies der Fall, so liefert das Prädikat `between/3` per Backtracking alle möglichen Positionen für die nächste zu setzende Dame. Der Aufruf von `attack/3` überprüft dann, ob die neue Dame in Spalte C eine der bereits in `Positioned` gesetzten Damen angreifen kann. Falls dies nicht der Fall ist, wird diese Dame den Damen aus der Liste `Positioned` vorangestellt und es wird rekursiv versucht das Problem zu lösen.
2. Die zweite Klausel greift, wenn die Liste `Positioned` der bereits gesetzten Damen das Problem löst. Dies ist der Fall wenn diese Liste bereits N Damen enthält.

Die Implementierung ist insgesamt effizienter weil jedesmal, wenn eine Dame gesetzt wird, überprüft wird, ob dies zu Problemen mit den bereits gesetzten Damen führen kann. Dadurch werden Teile des Suchbaums, die keine Lösung enthalten können, frühzeitig abgeschnitten.

```

1  % queens(+Number, -List(Number)).
2  queens(N, S) :-
3      queens(N, [], S).
4
5  % queens(+Number, +List(Number), -List(Number)).
6  queens(N, Positioned, Solution) :-
7      length(Positioned, M),
8      M < N,
9      between(1, N, C),
10     \+ attack(C, 1, Positioned),
11     queens(N, [C | Positioned], Solution).
12
13 queens(N, Positioned, Positioned) :-
14     length(Positioned, M),
15     M == N.

```

Figure 5.21: Eine effizientere Lösung für das n -Damen-Problem.

5.7 Der Cut-Operator

Wir haben ein Prädikat als *funktional* definiert, wenn wir die einzelnen Argumente klar in Eingabe- und Ausgabe-Argumente aufteilen können. Wir nennen ein Prädikat *deterministisch* wenn es funktional ist und wenn außerdem zu jeder Eingabe höchstens eine Ausgabe berechnet wird. Diese zweite Forderung ist durchaus nicht immer erfüllt. Betrachten wir die ersten beiden Fakten zur Definition des Prädikats `mix/3`:

```

1  mix( [], Xs, Xs ).
2  mix( Xs, [], Xs ).

```

Für die Anfrage “`mix([], [], L)`” können beide Fakten verwendet werden. Das Ergebnis ist zwar immer das selbe, nämlich `L = []`, es wird aber zweimal ausgegeben:

```

1  ?- mix([], [], L).
2
3  L = [] ;
4
5  L = []

```

Dies kann zu Ineffizienz führen. Aus diesem Grunde gibt es in *Prolog* den Cut-Operator “`!`”. Mit diesem Operator ist es möglich, redundante Lösungen aus dem Suchraum heraus zu schneiden. Schreiben wir die ersten beiden Klauseln der Implementierung von `mix/3` in der Form

```

1  mix( [], Xs, Xs ) :- !.
2  mix( Xs, [], Xs ) :- !.

```

so wird auf die Anfrage “ $\text{mix}([], [], L)$ ” die Lösung $L = []$ nur noch einmal generiert. Ist allgemein eine Regel der Form

$$P :- Q_1, \dots, Q_m, !, R_1, \dots, R_k$$

gegeben, und gibt es weiter eine Anfrage A , so dass A und P unifizierbar sind, so wird die Anfrage A zunächst zu der Anfrage

$$Q_1\mu, \dots, Q_m\mu, !, R_1\mu, \dots, R_k\mu$$

reduziert. Außerdem wird ein Auswahl-Punkt gesetzt, wenn es noch weitere Klauseln gibt, deren Kopf mit A unifiziert werden könnte. Bei der weiteren Abarbeitung dieser Anfrage gilt folgendes:

1. Falls bereits die Abarbeitung einer Anfrage der Form

$$Q_i\sigma, \dots, Q_m\sigma, !, R_1\sigma, \dots, R_k\sigma$$

für ein $i \in \{1, \dots, m\}$ scheitert, so wird der Cut nicht erreicht und hat keine Wirkung.

2. Eine Anfrage der Form

$$!, R_1\sigma, \dots, R_k\sigma$$

wird reduziert zu

$$R_1\sigma, \dots, R_k\sigma.$$

Dabei werden alle Auswahl-Punkte, die bei der Beantwortung der Teilanfragen Q_1, \dots, Q_m gesetzt worden sind, gelöscht. Außerdem wird ein eventuell bei der Reduktion der Anfrage A auf die Anfrage

$$Q_1\mu, \dots, Q_m\mu, !, R_1\mu, \dots, R_k\mu$$

gesetzter Auswahl-Punkte gelöscht.

3. Sollte später die Beantwortung der Anfrage

$$R_1\sigma, \dots, R_k\sigma.$$

scheitern, so scheitert auch die Beantwortung der Anfrage A .

Zur Veranschaulichung betrachten wir ein Beispiel.

```

1  q(Z) :- p(Z).
2  q(1).
3
4  p(X) :- a(X), b(X), !, c(X,Y), d(Y).
5  p(3).
6
7  a(1).    a(2).    a(3).
8
9  b(2).    b(3).
10
11 c(2,2).  c(2,4).
12
13 d(3).
```

Wir verfolgen die Beantwortung der Anfrage $q(U)$.

1. Zunächst wird versucht $q(U)$ mit dem Kopf der ersten Klausel des Prädikats $q/1$ zu unifizieren. Dabei wird Z mit U instantiiert und die Anfrage wird reduziert zu

$$p(U).$$

Da es noch eine weitere Klausel für das Prädikat $q/1$ gibt, die zur Beantwortung der Anfrage $q(U)$ in Frage kommt, setzen wir Auswahl-Punkt Nr. 1.

2. Jetzt wird versucht $p(U)$ mit $p(X)$ zu unifizieren. Dabei wird die Variable X an U gebunden und die ursprüngliche Anfrage wird reduziert zu der Anfrage

$$a(U), b(U), !, c(U, Y), d(Y).$$

Außerdem wird an dieser Stelle Auswahl-Punkt Nr. 2 gesetzt, denn die zweite Klausel des Prädikats $p/1$ kann ja ebenfalls mit der ursprünglichen Anfrage unifiziert werden.

3. Um die Teilanfrage $a(U)$ zu beantworten, wird $a(U)$ mit $a(1)$ unifiziert. Dabei wird U mit 1 instantiiert und die Anfrage wird reduziert zu

$$b(1), !, c(1, Y), d(Y).$$

Da es für das Prädikat $a/1$ noch weitere Klauseln gibt, wird Auswahl-Punkt Nr. 3 gesetzt.

4. Jetzt wird versucht, die Anfrage

$$b(1), !, c(1, Y), d(Y).$$

zu lösen. Dieser Versuch scheitert jedoch, da sich die für das Prädikat $b/1$ vorliegenden Fakten nicht mit $b(1)$ unifizieren lassen.

5. Also springen wir zurück zum letzten Auswahl-Punkt (das ist Auswahl-Punkt Nr. 3) und machen die Instantiierung $U \mapsto 1$ rückgängig. Wir haben jetzt also wieder das Ziel

$$a(U), b(U), !, c(U, Y), d(Y).$$

6. Diesmal wählen wir das Fakt $a(2)$ um es mit $a(U)$ zu unifizieren. Dabei wird U mit 2 instantiiert und wir haben die Anfrage

$$b(2), !, c(2, Y), d(Y).$$

Da es noch eine weitere Klausel für das Prädikat $a/1$ gibt, setzen wir Auswahl-Punkt Nr. 4 an dieser Stelle.

7. Jetzt unifizieren wir die Teilanfrage $b(2)$ mit der ersten Klausel für das Prädikat $b/1$. Die verbleibende Anfrage ist

$$!, c(2, Y), d(Y).$$

8. Diese Anfrage wird reduziert zu

$$c(2, Y), d(Y).$$

Außerdem werden bei diesem Schritt die Auswahl-Punkte Nr. 2 und Nr. 4 gelöscht.

9. Um diese Anfrage zu beantworten, unifizieren wir $c(2, Y)$ mit dem Kopf der ersten Klausel für das Prädikat $c/2$, also mit $c(2, 2)$. Dabei erhalten wir die Instantiierung $Y \mapsto 2$. Die Anfrage ist damit reduziert zu

$$d(2).$$

Außerdem setzen wir an dieser Stelle Auswahl-Punkt Nr. 5, denn das Prädikat $c/2$ hat ja noch eine weitere Klausel, die in Frage kommt.

10. Die Anfrage " $d(2)$ " scheitert. Also springen wir zurück zum Auswahl-Punkt Nr. 5 und machen die Instantiierung $Y \mapsto 2$ rückgängig. Wir haben also wieder die Anfrage

$$c(2, Y), d(Y).$$

11. Zur Beantwortung dieser Anfrage nehmen wir nun die zweite Klausel der Implementierung von $c/2$ und erhalten die Instantiierung $Y \mapsto 4$. Die verbleibende Anfrage lautet dann

$$d(4).$$

12. Da sich $d(4)$ und $d(3)$ nicht unifizieren lassen, scheitert diese Anfrage. Wir springen jetzt zurück zum Auswahl-Punkt Nr. 1 und machen die Instantiierung $U \mapsto Z$ rückgängig. Die Anfrage lautet also wieder

$$p(U).$$

13. Wählen wir nun die zweite Klausel der Implementierung von $q/1$, so müssen wir $q(U)$ und $q(1)$ unifizieren. Diese Unifikation ist erfolgreich und wir erhalten die Instantiierung $U \mapsto 1$, die die Anfrage beantwortet.

5.7.1 Verbesserung der Effizienz von *Prolog*-Programmen durch den Cut-Operator

In der Praxis wird der Cut-Operator eingesetzt, um überflüssige Auswahl-Punkte zu entfernen und dadurch die Effizienz eines Programms zu steigern. Als Beispiel betrachten wir eine Implementierung des Algorithmus “Sortieren durch Vertauschen” (engl. *bubble sort*). Wir spezifizieren diesen Algorithmus zunächst durch bedingte Gleichungen. Dabei benutzen wir die Funktion

$$\text{append} : \text{List}(\text{Number}) \times \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number})$$

Der Aufruf $\text{append}(l_1, l_2)$ liefert eine Liste, die aus allen Elementen von l_1 gefolgt von den Elementen aus l_2 besteht. In dem *SWI-Prolog*-System ist ein entsprechendes Prädikat $\text{append}/3$ implementiert. Die Implementierung dieses Prädikats deckt sich mit der Implementierung des Prädikats $\text{concat}/3$, die wir in einem früheren Abschnitt vorgestellt hatten.

Außerdem benutzen wir noch das Prädikat

$$\text{ordered} : \text{List}(\text{Number}) \rightarrow \mathbb{B},$$

das überprüft, ob eine Liste geordnet ist. Die bedingten Gleichungen zur Spezifikation der Funktion

$$\text{bubble_sort} : \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number})$$

lauten nun:

1. $\text{append}(l_1, [x, y|l_2], l) \wedge x > y \rightarrow \text{bubble_sort}(l) = \text{bubble_sort}(\text{append}(l_1, [y, x|l_2]))$
Wenn die Liste l in zwei Teile l_1 und $[x, y|l_2]$ zerlegt werden kann und wenn weiter $x > y$ ist, dann vertauschen wir die Elemente x und y und sortieren die so entstandene Liste rekursiv.
2. $\text{ordered}(l) \rightarrow \text{bubble_sort}(l) = l$
Wenn die Liste l bereits sortiert ist, dann kann die Funktion bubble_sort diese Liste unverändert als Ergebnis zurück geben.

Die Gleichungen um das Prädikat ordered zu spezifizieren lauten:

1. $\text{ordered}([]) = \text{true}$
Die leere Liste ist offensichtlich sortiert.
2. $\text{ordered}([x]) = \text{true}$
Eine Liste, die nur aus einem Element besteht, ist ebenfalls sortiert.
3. $x \leq y \rightarrow \text{ordered}([x, y|r]) = \text{ordered}([y|r])$.
Eine Liste der Form $[x, y|r]$ ist sortiert, wenn $x \leq y$ ist und wenn außerdem die Liste $[y|r]$ sortiert ist.

Abbildung 5.22 zeigt die Implementierung des Bubble-Sort Algorithmus in *Prolog*. In Zeile 2 wird die als Eingabe gegebene Liste L in die beiden Liste $L1$ und $[X, Y | L2]$ zerlegt. Da es im Allgemeinen mehrere Möglichkeiten gibt, eine Liste in zwei Teillisten zu zerlegen, wird hierbei ein Auswahl-Punkt gesetzt. Anschließend wird geprüft, ob Y kleiner als X ist. Wenn dies der Fall ist, wird mit $\text{append}/3$ die neue Liste

$$\text{append}(L1, [Y, X|L2])$$

gebildet und diese Liste wird rekursiv sortiert. Wenn es nicht möglich ist, die Liste L so in zwei Listen $L1$ und $[X, Y | L2]$ zu zerlegen, dass Y kleiner als X ist, dann muss die Liste L schon sortiert sein. In diesem Fall greift die zweite Klausel, die allerdings noch Überprüfen muss, ob

```

1  bubble_sort( L, Sorted ) :-
2      append( L1, [ X, Y | L2 ], L ),
3      X > Y,
4      append( L1, [ Y, X | L2 ], Cs ),
5      bubble_sort( Cs, Sorted ).
6
7  bubble_sort( Sorted, Sorted ) :-
8      is_ordered( Sorted ).
9
10
11  is_ordered( [] ).
12
13  is_ordered( [ _ ] ).
14
15  is_ordered( [ X, Y | Ys ] ) :-
16      X < Y,
17      is_ordered( [ Y | Ys ] ).

```

Figure 5.22: Der Bubble-Sort Algorithmus

L tatsächlich sortiert ist, denn sonst könnte beim Backtracking eine falsche Lösung berechnet werden.

Das Problem bei dem obigen Programm ist die Effizienz. Aufgrund der vielen Möglichkeiten eine Liste zu zerlegen, wird beim Backtracking immer wieder die selbe Lösung generiert. Beispielsweise liefert die Anfrage

```
bubble_sort( [ 4, 3, 2, 1 ], L ), write(L), nl, fail.
```

16 mal die selbe Lösung. Abbildung 5.23 zeigt eine Implementierung, bei der nur eine Lösung berechnet wird. Dies wird durch den Cut-Operator in Zeile 4 erreicht. Ist einmal eine Zerlegung der Liste L in L1 und [X, Y | L2] gefunden, bei der Y kleiner als X ist, so bringt es nichts mehr, nach anderen Zerlegungen zu suchen, denn die ursprünglich gegebene Liste L läßt sich ja auf jeden Fall dadurch sortieren, dass rekursiv die Liste

```
append(L1, [Y, X|L2])
```

sortiert wird. Dann kann auch der Aufruf der Prädikats `ordered/1` im Rumpf der zweiten Klausel des Prädikats `bubble_sort` entfallen, denn diese wird beim Backtracking ja nur dann erreicht, wenn es keine Zerlegung der Liste L in L1 und [X, Y | L2] gibt, bei der Y kleiner als X ist. Dann muß aber die Liste L schon sortiert sein.

```

1  bubble_sort( List, Sorted ) :-
2      append( L1, [ X, Y | L2 ], List ),
3      X > Y,
4      !,
5      append( L1, [ Y, X | L2 ], Cs ),
6      bubble_sort( Cs, Sorted ).
7
8  bubble_sort( Sorted, Sorted ).

```

Figure 5.23: Effiziente Implementierung des Bubble-Sort Algorithmus

Wenn wir bei der Entwicklung eines *Prolog*-Programms von bedingten Gleichungen ausgehen, dann gibt es ein einfaches Verfahren, um das entstandene *Prolog*-Programm durch die Einführung

von Cut-Operator effizienter zu machen: Der Cut-Operator sollte nach den Tests, die vor dem Junktor “ \rightarrow ” stehen, gesetzt werden. Wir erläutern dies durch ein Beispiel: Unten sind noch einmal die Gleichungen zur Spezifikation des Algorithmus “*Sortieren durch Mischen*” wiedergegeben.

1. $\text{odd}([]) = []$.
2. $\text{odd}([h|t]) = [h|\text{even}(t)]$.
3. $\text{even}([]) = []$.
4. $\text{even}([h|t]) = \text{odd}(t)$.
5. $\text{merge}([], l) = l$.
6. $\text{merge}(l, []) = l$.
7. $x \leq y \rightarrow \text{merge}([x|s], [y|t]) = [x|\text{merge}(s, [y|t])]$.
8. $x > y \rightarrow \text{merge}([x|s], [y|t]) = [y|\text{merge}([x|s], t)]$.
9. $\text{sort}([]) = []$.
10. $\text{sort}([x]) = [x]$.
11. $\text{sort}([x, y|t]) = \text{merge}(\text{sort}(\text{odd}([x, y|t])), \text{sort}(\text{even}([x, y|t])))$.

Das *Prolog*-Programm mit Cut-Operatoren sieht dann so aus wie in Abbildung 5.24 gezeigt. Nur die Gleichungen 7. und 8. haben Bedingungen, bei allen anderen Gleichungen gibt es keine Bedingungen. Bei den Gleichungen 7. und 8. wird der Cut-Operator daher nach dem Test der Bedingungen gesetzt, bei allen anderen Klauseln wird der Cut-Operator dann am Anfang des Rumpfes gesetzt.

Analysieren wir das obige Programm genauer, so stellen wir fest, dass viele der Cut-Operatoren im Grunde überflüssig sind. Beispielsweise kann immer nur eine der beiden Klauseln, die das Prädikat `odd/2` implementieren, greifen, denn entweder ist die eingegebene Liste leer oder nicht. Also sind die Cut-Operatoren in den Zeilen 2 und 4 redundant. Andererseits stören sie auch nicht, so dass es für die Praxis das einfachste sein dürfte Cut-Operatoren stur nach dem oben angegebenen Rezept zu setzen.

5.8 Literaturhinweise

Für eine umfangreiche und dem Thema angemessene Darstellung der Sprache *Prolog* fehlt in der einführenden Vorlesung leider die Zeit. Daher wird dieses Thema in einer späteren Vorlesung auch wieder aufgegriffen. Den Lesern, die ihre Kenntnisse jetzt schon vertiefen wollen, möchte ich auf die folgende Hinweise auf die Literatur geben:

1. *The Art of Prolog* von Leon Sterling und Ehud Shapiro [SS94]. Dieses Werk ist ein ausgezeichnetes Lehrbuch, das auch für den Anfänger gut lesbar ist.
2. *Prolog Programming for Artificial Intelligence* von Ivan Bratko [Bra90]. Neben der Sprache *Prolog* führt dieses Buch auch in die künstliche Intelligenz ein.
3. *Foundations of Logic Programming* von J. W. Lloyd [Llo87] beschreibt die theoretischen Grundlagen der Sprache *Prolog*.
4. *Prolog: The Standard* von Pierre Deransart, Abdel Ali Ed-Dbali und Laurent Cervoni [DEDC96] gibt den ISO-Standard für die Sprache *Prolog* wieder.

```

1  % odd( +List(Number), -List(Number) ).
2  odd( [], [] ) :- !.
3  odd( [ X | Xs ], [ X | L ] ) :-
4      !,
5      even( Xs, L ).
6
7  % even( +List(Number), -List(Number) ).
8  even( [], [] ) :- !.
9  even( [ _X | Xs ], L ) :-
10     !,
11     odd( Xs, L ).
12
13 % merge( +List(Number), +List(Number), -List(Number) ).
14 mix( [], Xs, Xs ) :- !.
15 mix( Xs, [], Xs ) :- !.
16 mix( [ X | Xs ], [ Y | Ys ], [ X | Rest ] ) :-
17     X =< Y,
18     !,
19     mix( Xs, [ Y | Ys ], Rest ).
20 mix( [ X | Xs ], [ Y | Ys ], [ Y | Rest ] ) :-
21     X > Y,
22     !,
23     mix( [ X | Xs ], Ys, Rest ).
24
25 % merge_sort( +List(Number), -List(Number) ).
26 merge_sort( [], [] ) :- !.
27 merge_sort( [ X ], [ X ] ) :- !.
28 merge_sort( [ X, Y | Rest ], Sorted ) :-
29     !,
30     odd( [ X, Y | Rest ], Odd ),
31     even( [ X, Y | Rest ], Even ),
32     merge_sort( Odd, Odd_Sorted ),
33     merge_sort( Even, Even_Sorted ),
34     mix( Odd_Sorted, Even_Sorted, Sorted ).

```

Figure 5.24: Sortieren durch Mischen mit Cut-Operatoren.

5. *SWI-Prolog 5.6 Reference Manual* von Jan Wielemaker [Wie06] beschreibt das SWI-Prolog-System. Dieses Dokument ist im Internet unter der Adresse <http://www.swi-prolog.org/dl-doc.html> verfügbar.

Bibliography

- [Bra90] Ivan Bratko. *PROLOG Programming for Artificial Intelligence*. Addison-Wesley, 2nd edition, 1990.
- [Can95] Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Mathematische Annalen*, 46:481–512, 1895.
- [DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.
- [DH90] Robert B. K. Dewar and Robert Hummel. A gentle introduction to the SETL2 programming language. Technical report, 1990. Available at: <ftp://cs.nyu.edu/pub/languages/setl2/doc/2.2/intro.ps>.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [Lip98] Seymour Lipschutz. *Set Theory and Related Topics*. McGraw-Hill, New York, 1998.
- [Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1987.
- [MMZ⁺01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
- [Sch03] Jack Schwarz. *Programming in SETL*. 2003. Available at <http://www.settheory.com>.
- [Sny90] W. Kirk Snyder. The Setl2 programming language: Update on current developments. Technical report, 1990. Available at <ftp://cs.nyu.edu/pub/languages/setl2/doc/2.2/update.ps>.
- [SS94] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.
- [Wie06] J. Wielemaker. SWI-Prolog 5.6 reference manual, 2006. Online available at <http://www.swi-prolog.org/dl-doc.html>.