

Informatik I: Logik
— Wintersemester 2011/2012 —
DHBW Stuttgart

Prof. Dr. Karl Stroetmann

3. Mai 2012

Inhaltsverzeichnis

1	Einleitung	4
1.1	Überblick über den Inhalt der Vorlesung:	4
1.2	Motivation	5
2	Die Programmier-Sprache <i>SetIX</i>	6
2.1	Einführende Beispiele	6
2.2	Darstellung von Mengen	10
2.3	Paare, Relationen und Funktionen	13
2.4	Allgemeine Tupel	14
2.5	Spezielle Funktionen und Operatoren auf Mengen	15
2.5.1	Anwendung: <i>Sortieren durch Auswahl</i>	16
2.6	Kontroll-Strukturen	18
2.6.1	Schleifen	20
2.6.2	Fixpunkt-Algorithmen	23
2.6.3	Verschiedenes	24
2.7	Fallstudie: Berechnung von Wahrscheinlichkeiten	24
2.8	Fallstudie: Berechnung von Pfaden	26
2.8.1	Berechnung des transitiven Abschlusses einer Relation	27
2.8.2	Berechnung der Pfade	30
2.8.3	Der Bauer mit dem Wolf, der Ziege und dem Kohl	33
2.8.4	Ausblick	34
3	Aussagenlogik	36
3.1	Motivation	36
3.2	Anwendungen der Aussagenlogik	37
3.3	Formale Definition der aussagenlogischen Formeln	38
3.3.1	Syntax der aussagenlogischen Formeln	38
3.3.2	Semantik der aussagenlogischen Formeln	39
3.3.3	Extensionale und intensionale Interpretationen der Aussagenlogik	41
3.3.4	Implementierung in <i>SetIX</i>	42
3.3.5	Eine Anwendung	44
3.4	Tautologien	47
3.4.1	Testen der Allgemeingültigkeit in <i>SetIX</i>	48
3.4.2	Nachweis der Allgemeingültigkeit durch Äquivalenz-Umformungen	50

3.4.3	Berechnung der konjunktiven Normalform in <i>SetlX</i>	54
3.5	Der Herleitungs-Begriff	60
3.5.1	Eigenschaften des Herleitungs-Begriffs	63
3.6	Das Verfahren von Davis und Putnam	64
3.6.1	Vereinfachung mit der Schnitt-Regel	65
3.6.2	Vereinfachung durch Subsumption	65
3.6.3	Vereinfachung durch Fallunterscheidung	66
3.6.4	Der Algorithmus	66
3.6.5	Ein Beispiel	67
3.6.6	Implementierung des Algorithmus von Davis und Putnam	68
3.7	Das 8-Damen-Problem	71
4	Prädikatenlogik	79
4.1	Syntax der Prädikatenlogik	79
4.2	Semantik der Prädikatenlogik	83
4.2.1	Implementierung prädikatenlogischer Strukturen in SETLX	87
4.3	Normalformen für prädikatenlogische Formeln	93
4.4	Unifikation	97
4.5	Ein Kalkül für die Prädikatenlogik	102
4.6	<i>Prover9</i> und <i>Mace4</i>	108
4.6.1	Der automatische Beweiser <i>Prover9</i>	108
4.6.2	<i>Mace4</i>	109
5	<i>Prolog</i>	112
5.1	Wie arbeitet <i>Prolog</i> ?	115
5.1.1	Die Tiefensuche	121
5.2	Ein komplexeres Beispiel	122
5.3	Listen	124
5.3.1	Sortieren durch Einfügen	126
5.3.2	Sortieren durch Mischen	128
5.3.3	Symbolisches Differenzieren	131
5.4	Negation in <i>Prolog</i>	136
5.4.1	Berechnung der Differenz zweier Listen	136
5.4.2	Semantik des Negations-Operators in PROLOG	137
5.5	Die Tiefen-Suche in <i>Prolog</i>	138
5.5.1	Missionare und Kannibalen	141
5.6	Der Cut-Operator	145
5.6.1	Verbesserung der Effizienz von <i>Prolog</i> -Programmen durch den Cut-Operator	147
5.7	Literaturhinweise	151

6	Der Hoare-Kalkül	152
6.1	Vor- und Nachbedingungen	152
6.1.1	Spezifikation von Zuweisungen	153
6.1.2	Die Abschwächungs-Regel	155
6.1.3	Zusammengesetzte Anweisungen	155
6.1.4	Alternativ-Anweisungen	157
6.1.5	Schleifen	158
6.2	Der Euklid'sche Algorithmus	158
6.2.1	Nachweis der Korrektheit des Euklid'schen Algorithmus	158
6.2.2	Maschinelle Programm-Verifikation	163
6.3	Symbolische Programm-Ausführung	165

Kapitel 1

Einleitung

Die erste Informatik-Vorlesung legt die Grundlagen, die für das weitere Studium der Informatik benötigt werden. Bei diesen Grundlagen handelt es sich im wesentlichen um die *mathematische Logik*. Als Anwendung dieser Grundlagen werden zwei Programmiersprachen vorgestellt:

1. *SetIX* (set language extended) ist eine mengenbasierte Programmiersprache, in der dem Programmierer die Operationen der Mengenlehre zur Verfügung gestellt werden. Zusätzlich unterstützt die Sprache *funktionales Programmieren*.
2. *Prolog* (programming in logic) basiert auf prädikatenlogischen Konzepten und ist die erste Programmiersprache der *fünften Generation*. Diese Sprache wurde früher vor allem bei KI-Projekten eingesetzt.

1.1 Überblick über den Inhalt der Vorlesung:

Die Begriffs-Bildungen der Mengenlehre sind nicht sehr kompliziert, dafür aber um so abstrakter. Um diese Begriffs-Bildungen konkreter werden zu lassen und darüber hinaus den Studenten ein Gefühl für die Nützlichkeit der Mengenlehre zu geben, stellen wir im zweiten Kapitel die Sprache *SetIX* vor. Dies ist eine Programmier-Sprache, die auf der Mengenlehre aufgebaut ist. Neben den klassischen Datentypen wie Zahlen und Strings gibt es hier als Datentypen zusätzlich Mengen. Dadurch ist es in *SetIX* möglich, Algorithmen in der Sprache der Mengenlehre zu formulieren. Solche Algorithmen sind zwar meistens nicht so effizient wie Implementierungen in einer klassischen Programmier-Sprache, aber dafür in der Regel wesentlich klarer (und damit schneller zu implementieren) als beispielsweise ein entsprechendes C-Programm. Zusätzlich hat *SetIX* eine konzeptuelle Ähnlichkeit mit Datenbank-Abfrage-Sprachen wie beispielsweise SQL, so dass sich eine Vertrautheit mit den Konzepten dieser Sprache auch später noch als nützlich erweist.

In dem dritten Kapitel widmen wir uns der *Aussagen-Logik*. Die Handhabung aussagenlogischer Formeln ist einfacher als die Handhabung prädikatenlogischer Formeln. Daher bietet sich die Aussagen-Logik gewissermassen als Trainings-Objekt an um mit den Methoden der Logik vertraut zu werden. Die Aussagen-Logik hat gegenüber der Prädikaten-Logik noch einen weiteren Vorteil: Sie ist *entscheidbar*, d.h. wir können ein Programm schreiben, dass als Eingabe eine aussagenlogische Formel verarbeitet und welches dann entscheidet, ob diese Formel gültig ist. Ein solches Programm existiert für beliebige Formeln der Prädikaten-Logik nicht. Darüber hinaus gibt es in der Praxis eine Reihe von Problemen, die bereits mit Hilfe der Aussagenlogik gelöst werden können. Beispielsweise lässt sich die Frage nach der Korrektheit kombinatorischer digitaler Schaltungen auf die Entscheidbarkeit einer aussagenlogischen Formel zurückführen. Außerdem gibt es eine Reihe kombinatorischer Probleme, die sich auf aussagenlogische Probleme reduzieren lassen. Als ein Beispiel zeigen wir, wie sich das 8-Damen-Problem mit Hilfe der Aussagenlogik lösen lässt.

Im vierten Kapitel behandeln wir die Prädikatenlogik und analysieren den Begriff des prädikatenlogischen Beweises mit Hilfe eines *Kalküls*. Ein *Kalkül* ist dabei ein formales Verfahren, einen mathematischen Beweis zu führen. Ein solches Verfahren läßt sich programmieren. Wir stellen zu diesem Zweck den *Resolutions-Kalkül* vor, mit dem sich konstruktive Beweise führen lassen.

Der Resolutions-Kalkül bildet die Grundlage für die Programmier-Sprache *Prolog*, die wir im fünften Kapitel diskutieren.

1.2 Motivation

Da sowohl die mathematische Logik und auch die in der Mathematik eingeführte Mengenlehre sehr abstrakt sind, bereiten sie erfahrungsgemäß vielen Studenten Schwierigkeiten. Im Rahmen der Mathematik-Vorlesung habe ich bereits Gründe gegeben, warum wir uns trotzdem mit diesen beiden Gebieten beschäftigen müssen. Ich möchte diese Gründe hier nicht wiederholen sondern statt dessen auf den Anfang des Mathematik-Skripts verweisen.

Zum Schluß möchte ich hier noch ein Paar Worte zum Gebrauch von neuer und alter Rechtschreibung und der Verwendung von Spell-Checkern in diesem Skript sagen. Dieses Skript wurde unter Verwendung strenger marktwirtschaftlicher Kriterien erstellt. Im Klartext heißt das: Zeit ist Geld und als Dozent an der DHBW hat man weder das eine noch das andere. Daher ist es sehr wichtig zu wissen, wo eine zusätzliche Investition von Zeit noch einen für die Studenten nützlichen Effekt bringt und wo dies nicht der Fall ist. Ich habe mich an aktuellen Forschungs-Ergebnissen zum Nutzen der Rechtschreibung orientiert. Diese zeigen, daß es nicht wichtig ist, in welcher Reihenfolge die Buchstaben in einem Wort stehen, das einzige was wichtig ist, ist dass der erste und der letzte Buchstabe an der richtigen Position steht. Der Rest kann ein toller Böldisn sein, trotzdem kann man ihn ohne Probleme lesen. Das ist so, weil wir nicht jeden Buchstaben einzeln lesen, sondern das Wort als Gesamtes. Wie sie sehen, ist das tatsächlich der Fall. ☺

Kapitel 2

Die Programmier-Sprache *SetlX*

Die im letzten Kapitel vorgestellten Begriffs-Bildungen aus der Mengenlehre bereiten erfahrungsgemäß dem Anfänger aufgrund ihrer Abstraktheit gewisse Schwierigkeiten. Um diese Begriffe vertrauter werden zu lassen, stelle ich daher nun eine Programmier-Sprache vor, die mit diesen Begriffen arbeitet. Dies ist die Sprache *SetlX*. Diese Sprache basiert auf der Ende der siebziger Jahre von Jack T. Schwartz eingeführten Sprache *Setl* [SDSD86]. Die Sprache *SetlX* lehnt sich in ihrer Syntax stark an die Programmiersprache C an, ist ansonsten aber als Derivat von *Setl* zu sehen. Sie finden auf der Webseite

<http://www.lehre.dhbw-stuttgart.de/~stroetma/SetlX/setlX.php>

eine Anleitung zur Installation von *SetlX*.

2.1 Einführende Beispiele

Wir wollen in diesem Abschnitt die Sprache *SetlX* an Hand einiger einfacher Beispiele vorstellen, bevor wir dann in den folgenden Abschnitten auf die Details eingehen. Die Sprache *SetlX* ist eine interaktive Sprache, Sie können diese Sprache also unmittelbar über einen Interpreter aufrufen. Falls Sie *SetlX* auf Ihrem Rechner installiert haben, können Sie den Befehl

```
setlx
```

in einer Kommando-Zeile eingeben. Anschließend meldet sich der Interpreter dann wie in Abbildung 2.1 gezeigt. Die Zeichenfolge “=>” ist der Prompt, der Ihnen signalisiert, dass der Interpreter auf eine Eingabe wartet. Geben Sie dort den Text

```
print("Hallo");
```

ein und drücken anschließend zweimal auf die Eingabe-Taste¹, so erhalten Sie die folgende Ausgabe:

```
1  Hallo
2  Result: om
3
4  =>
```

Hier hat der Interpreter zunächst den Befehl `print("Hallo")` ausgeführt und dabei den Text “Hallo” ausgegeben. In der Zeile darunter wird der Wert des zuletzt ausgegebenen Ausdrucks

¹Eine zweimalige Betätigung der Eingabe-Taste ist erforderlich, weil es durchaus Befehle gibt, die sich über mehrere Zeilen erstrecken. Daher wertet der Interpreter die Eingabe erst dann aus, wenn er zwei aufeinander folgende Zeilenumbrüche sieht.

```

1  =====setlX=====v0.2.1==
2
3  Welcome to the setlX interpreter!
4
5  You can display some helpful information by using '--help' as parameter
6  when launching this program.
7
8  Interactive-Mode:
9      Two newline characters execute previous input.
10     The 'exit;' statement terminates the interpreter.
11
12  =====Interactive=Mode=====
13
14  =>

```

Abbildung 2.1: *SetlX*-Interpreter nach dem Start.

angezeigt. Da die Funktion `print()` kein Ergebnis berechnet, ist der Rückgabe-Wert undefiniert. Ein undefinierter Wert wird in *SetlX* mit Ω bezeichnet, was in der Ausgabe durch den String “om” dargestellt wird.

Die Funktion `print()` akzeptiert beliebig viele Argumente. Wenn Sie beispielsweise das Ergebnis der Rechnung $36 * 37 / 2$ ausgeben wollen, so können Sie dies über den Befehl

```
print("36 * 37 / 2 = ", 36 * 37 / 2);
```

erreichen. Wenn Sie nur an der Auswertung dieses Ausdrucks interessiert sind, so können Sie diesen Ausdruck auch unmittelbar hinter dem Prompt eingeben und mit einem Semikolon “;” abschließen. Wenn Sie nun zweimal die Eingabe-Taste betätigen, wird der Ausdruck ausgewertet und das Ergebnis angezeigt.

Der *SetlX*-Interpreter läßt sich nicht nur interaktiv betreiben, sondern er kann auch vollständige Programme ausführen. Speichern wir das in Abbildung 2.2 gezeigte Programm in einer Datei mit dem Namen “`sum.stlx`” ab, so können wir in der Kommando-Zeile den Befehl

```
setlX sum.stlx
```

eingeben. Dann wird zunächst der Text “**Type an natural number ...**” gefolgt von einem Doppelpunkt als Prompt ausgegeben. Geben wir dann eine Zahl n ein und betätigen die Eingabe-Taste, so wird als Ergebnis die Summe $\sum_{i=1}^n i$ ausgegeben.

```

1  // This program reads a number n and computes the sum 1 + 2 + ... + n.
2  print("Type a natural number and press return.");
3  n := read();
4  s := +/ { 1 .. n };
5  print("The sum 1 + 2 + ... + ", n, " is equal to ", s, ".");

```

Abbildung 2.2: Ein einfaches Programm zur Berechnung der Summe $\sum_{i=1}^n i$.

Wir diskutieren nun das in Abbildung 2.2 auf Seite 7 gezeigte Programm Zeile für Zeile. Die Zeilen-Nummern in dieser und den folgenden Abbildungen von *SetlX*-Programmen sind nicht Bestandteil der Programme sondern wurden hinzugefügt um in den Diskussionen dieser Programme besser auf einzelne Zeilen Bezug nehmen zu können.

1. Die erste Zeile enthält einen Kommentar. In *SetlX* werden Kommentare durch den String

“//” eingeleitet. Aller Text zwischen diesem String und dem Ende der Zeile wird von dem *SetLX*-Compiler ignoriert. Neben einzeiligen Kommentaren unterstützt *SetLX* auch mehrzeilige Kommentare, die wie in der Sprache *C* durch die Strings “/*” und “*/” begrenzt werden.

- Die zweite Zeile enthält einen Ausgabe-Befehl. Diese Zeile zeigt auch den ersten Datentyp der Sprache *SetLX*, die *Strings*. Diese werden in *SetLX* in doppelte Anführungszeichen eingeschlossen.

Die Zeile wird mit dem Zeichen “;” abgeschlossen. In *SetLX* werden alle Befehle mit einem Semikolon abgeschlossen.

- Die dritte Zeile enthält eine Zuweisung. Die Funktion *read()* liest einen vom Benutzer eingegebenen String, erkennt, dass es sich um eine Zahl handelt und weist mit Hilfe des Zuweisungs-Operators “:=” der Variablen *n* die gelesene Zahl zu. An dieser Stelle gibt es einen wichtigen Unterschied zu Sprache *C*, denn dort ist der Zuweisungs-Operator der String “=”.

Im Gegensatz zu der Sprache *C* ist die Sprache *SetLX* *ungetypt*. Daher ist es weder notwendig noch möglich, die Variable *n* zu deklarieren. Würde der Benutzer an Stelle einer Zahl einen String eingeben, so würde das Programm später mit einer Fehlermeldung abbrechen.

- Die vierte Zeile zeigt zunächst, wie sich Mengen als Aufzählungen definieren lassen. Sind *a* und *b* ganze Zahlen mit $a < b$, so berechnet der Ausdruck

$$\{ a \dots b \}$$

die Menge

$$\{x \in \mathbb{Z} \mid a \leq x \wedge x \leq b\}.$$

Der Operator “+” berechnet dann die Summe aller Elemente der Menge

$$\{i \in \mathbb{N} \mid 1 \leq i \wedge i \leq n\}$$

Das ist natürlich genau die Summe

$$1 + 2 + \dots + n = \sum_{i=1}^n i.$$

Diese Summe wird der Variablen *s* zugewiesen.

- In der letzten Zeile wird diese Summe ausgegeben.

Als nächstes betrachten wir das in Abbildung 2.3 auf Seite 9 gezeigte Programm `sum-recursive.stlx`, das die Summe $\sum_{i=0}^n i$ mit Hilfe einer Rekursion berechnet.

- Zeile 1 bis Zeile 7 enthalten die Definition der Prozedur `sum`. Die Definition einer Prozedur wird in *SetLX* durch das Schlüsselwort “**procedure**” eingeleitet. Hinter diesem Schlüsselwort folgt zunächst eine öffnende Klammer “(”, dann eine Liste von Argumenten, welche durch “,” voneinander getrennt sind, und danach eine schließende Klammer “)”. Darauf folgt der Rumpf der Prozedur, der, genau wie in der Sprache *C*, durch die Klammern “{” und “}” begrenzt wird. Im Allgemeinen besteht der Rumpf aus einer Liste von Kommandos. In unserem Fall haben wir hier nur ein einziges Kommando. Dieses Kommando ist allerdings ein zusammengesetztes Kommando und zwar eine Fallunterscheidung. Die allgemeine Form einer Fallunterscheidung ist wie folgt:

```

1  sum := procedure(n) {
2      if (n == 0) {
3          return 0;
4      } else {
5          return sum(n-1) + n;
6      }
7  };
8
9  n      := read();
10 total := sum(n);
11 print("Sum 0 + 1 + 2 + ... + ", n, " = ", total);

```

Abbildung 2.3: Ein rekursives Programm zur Berechnung der Summe $\sum_{i=0}^n i$.

```

1      if ( test ) {
2          body1
3      } else {
4          body2
5      }

```

Die Semantik der Fallunterscheidung ist wie folgt:

- Zunächst wird der Ausdruck *test* ausgewertet. Bei der Auswertung muß sich entweder der Wert “true” oder “false” ergeben.
- Falls sich “true” ergibt, werden anschließend die Kommandos in *body₁* ausgeführt. Dabei ist *body₁* eine Liste von Kommandos.
- Andernfalls werden die Kommandos in der Liste *body₂* ausgeführt.

Beachten Sie, dass die Definition der Prozedur durch ein Semikolon abgeschlossen wird.

- Nach der Definition der Prozedur **sum** wird in Zeile 9 ein Wert in die Variable **n** eingelesen.
- Dann wird in Zeile 10 für den eben eingelesenen Wert von **n** die oben definierte Prozedur **sum** aufgerufen.

Zusätzlich enthält Zeile 10 eine Zuweisung: Der Wert, den der Prozedur-Aufruf **sum(n)** zurück liefert, wird in die Variable **total**, die auf der linken Seite des *Zuweisungs-Operators* “:=” steht, geschrieben.

Die Prozedur **sum** in dem obigen Beispiel ist *rekursiv*, d.h. sie ruft sich selber auf. Die Logik, die hinter der Implementierung steht, läßt sich am einfachsten durch die beiden folgenden bedingten Gleichungen erfassen:

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

Die Korrektheit dieser Gleichungen wird unmittelbar klar, wenn wir für $sum(n)$ die Definition

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

einsetzen, denn offenbar gilt:

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

Die erste Gleichung behandelt den Fall, dass die Prozedur sich nicht selbst aufruft. Einen solchen Fall muss es in jeder rekursiv definierten Prozedur geben, denn sonst würde die Prozedur in einer Endlos-Schleife stecken bleiben.

2.2 Darstellung von Mengen

Der wichtigste Unterschied zwischen der Sprache SETLX und der Sprache C besteht darin, dass *SetlX* die Verwendung von Mengen und Listen unmittelbar unterstützt. Um zu zeigen, wie wir in *SetlX* mit Mengen umgehen können, zeigen wir ein einfaches Programm, das Vereinigung, Schnitt und Differenz zweier Mengen berechnet. Abbildung 2.4 zeigt die Datei `simple.stlx`. Das in dieser Abbildung gezeigte Programm zeigt die Verwendung der elementaren Mengen-Operatoren in *SetlX*.

```

1  a := { 1, 2, 3 };
2  b := { 2, 3, 4 };
3  // Berechnung der Vereinigungs-Menge a ∪ b
4  c := a + b;
5  print(a, " + ", b, " = ", c);
6  // Berechnung der Schnitt-Menge      a ∩ b
7  c := a * b;
8  print(a, " * ", b, " = ", c);
9  // Berechnung der Mengen-Differenz   a \ b
10 c := a - b;
11 print(a, " - ", b, " = ", c);
12 // Berechnung der Potenz-Menge      2a
13 c := pow(a);
14 print("pow(", a, ") = ", c);
15 // Überprüfung einer Teilmengen-Beziehung a ⊆ b
16 print("(", a, " <= ", b, ") = ", (a <= b));
```

Abbildung 2.4: Berechnung von \cup , \cap , \setminus und Potenz-Menge

In Zeile 1 und 2 sehen wir, dass wir Mengen ganz einfach durch explizite Aufzählung ihrer Argumente angeben können. In den Zeilen 4, 7 und 10 berechnen wir dann nacheinander Vereinigung, Schnitt und Differenz dieser Mengen. Hier ist zu beachten, dass dafür in *SetlX* die Operatoren “+”, “*”, “-” verwendet werden. In Zeile 13 berechnen wir die Potenz-Menge mit Hilfe der Funktion `pow()`. Schließlich überprüfen wir in Zeile 17 mit dem Operator “<=”, ob *a* eine Teilmenge von *b* ist. Führen wir dieses Programm aus, so erhalten wir die folgende Ausgabe:

```

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

Um interessantere Programme zeigen zu können, stellen wir jetzt weitere Möglichkeiten vor, mit denen wir in *SetlX* Mengen definieren können.

Definition von Mengen durch arithmetische Aufzählung

In dem letzten Beispiel hatten wir Mengen durch explizite Aufzählung definiert. Das ist bei großen Mengen viel zu mühsam. Eine Alternative ist daher die Definition einer Menge durch eine *arithmetische Aufzählung*. Wir betrachten zunächst ein Beispiel:

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

Die Menge, die hier der Variablen `a` zugewiesen wird, ist die Menge aller natürlichen Zahlen von 1 bis 100. Die allgemeine Form einer solchen Definition ist

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

Mit dieser Definition würde `a` die Menge aller ganzen Zahlen von `start` bis `stop` zugewiesen, formal gilt

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

Es gibt noch eine Variante der arithmetischen Aufzählung, die wir ebenfalls durch ein Beispiel einführen.

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

Die Menge, die hier der Variablen `a` zugewiesen wird, ist die Menge aller ungeraden natürlichen Zahlen von 1 bis 100. Die Zahl 100 liegt natürlich nicht in dieser Menge, denn sie ist ja gerade. Die allgemeine Form einer solchen Definition ist

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

Definieren wir $\text{step} = \text{second} - \text{start}$ und ist step positiv, so läßt sich diese Menge formal wie folgt definieren:

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

Beachten Sie, dass `stop` nicht unbedingt ein Element der Menge

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

ist. Beispielsweise gilt

$$\{ 1, 3 .. 6 \} = \{ 1, 3, 5 \}.$$

Definition von Mengen durch Iteratoren

Eine weitere Möglichkeit, Mengen zu definieren, ist durch die Verwendung von *Iteratoren* gegeben. Wir geben zunächst ein einfaches Beispiel:

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

Nach dieser Zuweisung enthält `p` die Menge aller *nicht-trivialen* Produkte, deren Faktoren ≤ 10 sind. (Ein Produkt der Form $a \cdot b$ heißt dabei *trivial* genau dann, wenn einer der Faktoren a oder b den Wert 1 hat.) In der Schreibweise der Mathematik gilt für die oben definierte Menge `p`:

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

Wie ausdrucksstark Iteratoren sind, läßt sich an dem Programm `primes-difference.stl` erkennen, das in Abbildung 2.5 auf Seite 12 gezeigt ist. Das Programm berechnet die Menge der Primzahlen bis zu einer vorgegebenen Größe n und ist so kurz wie eindrucksvoll. Die zugrunde liegende Idee ist, dass eine Zahl genau dann eine Primzahl ist, wenn Sie von 1 verschieden ist und sich nicht als Produkt zweier von 1 verschiedener Zahlen schreiben läßt. Um also die Menge der Primzahlen kleiner gleich n zu berechnen, ziehen wir einfach von der Menge $\{2, \dots, n\}$ die Menge aller Produkte ab. Genau dies passiert in Zeile 2 des Programms.

Die allgemeine Form der Definition einer Menge mit Iteratoren ist durch den Ausdruck

```

1  n := 100;
2  primes := {2 .. n} - { p * q : p in {2..n}, q in {2..n} };
3  print(primes);

```

Abbildung 2.5: Programm zur Berechnung der Primzahlen bis n .

$$\{expr : x_1 \text{ in } S_1, \dots, x_n \text{ in } S_n\}$$

gegeben. Hierbei ist $expr$ ein Term, in dem die Variablen x_1 bis x_n auftreten. Weiterhin sind S_1 bis S_n Ausdrücke, die bei ihrer Auswertung Mengen ergeben. Die Ausdrücke $x_i \text{ in } S_i$ werden dabei als *Iteratoren* bezeichnet, weil die Variablen x_i über alle Werte der entsprechenden Menge S_i laufen (wir sagen auch: *iterieren*). Die mathematische Interpretation der obigen Menge ist dann durch

$$\{expr \mid x_1 \in S_1 \wedge \dots \wedge x_n \in S_n\}$$

gegeben. Die Definition einer Menge über Iteratoren entspricht der Definition einer Menge als Bild-Menge.

Es ist in *SetLX* auch möglich, das *Auswahl-Prinzip* zu verwenden. Dazu können wir Iteratoren mit einer Bedingung verknüpfen. Die allgemeine Syntax dafür ist:

$$\{expr : x_1 \text{ in } S_1, \dots, x_n \text{ in } S_n \mid cond\}.$$

Hierbei haben die Ausdrücke $expr$ und S_i die selbe Bedeutung wie oben und $cond$ ist ein Ausdruck, der von den Variablen x_1, \dots, x_n abhängt und dessen Auswertung entweder **true** oder **false** ergibt. Die mathematische Interpretation der obigen Menge ist dann

$$\{expr \mid x_1 \in S_1 \wedge \dots \wedge x_n \in S_n \wedge cond\}.$$

Wir geben ein konkretes Beispiel: Nach der Zuweisung

```
primes := { p : p in {2..100} | { x : x in {1..p} | p % x == 0 } == {1, p} };
```

enthält **primes** die Menge aller Primzahlen, die kleiner oder gleich 100 sind. Die der obigen Berechnung zugrunde liegende Idee besteht darin, dass eine Zahl genau dann eine Primzahl ist, wenn Sie nur durch 1 und sich selbst teilbar ist. Um festzustellen, ob eine Zahl p durch eine andere Zahl x teilbar ist, können wir in *SetLX* den Operator **%** verwenden: Der Ausdruck $p \% x$ berechnet den Rest, der übrig bleibt, wenn Sie die Zahl p durch x teilen. Eine Zahl p ist also genau dann durch eine andere Zahl x teilbar, wenn der Rest 0 ist, wenn also $p \% x = 0$ gilt. Damit liefert

$$\{ t \text{ in } \{1..p\} \mid p \% t == 0 \}$$

genau die Menge aller Teiler von p und p ist eine Primzahl, wenn diese Menge nur aus den beiden Zahlen 1 und p besteht. Das Programm aus der Datei **primes.stl**, das in Abbildung 2.6 auf Seite 12 gezeigt wird, benutzt diese Methode zur Berechnung der Primzahlen.

```

1  teiler := procedure(p) {
2      return { t in {1..p} | p % t == 0 };
3  };
4  n      := 100;
5  primes := { p in {2..n} | teiler(p) == {1, p} };
6  print(primes);

```

Abbildung 2.6: Alternatives Programm zur Berechnung der Primzahlen.

Zunächst haben wir in den Zeilen 1 bis 3 die Funktion **teiler(p)** definiert, die als Ergebnis

die Menge der Teiler der Zahl p berechnet. Dabei haben wir in Zeile 2 eine etwas andere Syntax benutzt, als oben angegeben. Wir haben dabei die folgende Abkürzungs-Möglichkeit der Sprache *SetIX* verwendet: Eine Menge der Form

$$\{x : x \text{ in } S \mid \text{cond}\} \quad \text{kann kürzer als} \quad \{x \text{ in } S \mid \text{cond}\}$$

geschrieben werden.

2.3 Paare, Relationen und Funktionen

Das Paar $\langle x, y \rangle$ wird in *SetIX* in der Form $[x, y]$ dargestellt, die spitzen Klammern werden also durch eckige Klammern ersetzt. Wir hatten im letzten Kapitel gesehen, dass wir eine Menge von Paaren, die sowohl links-total als auch rechts-eindeutig ist, auch als Funktion auffassen können. Ist R eine solche Menge und $x \in \text{dom}(R)$, so bezeichnet in *SetIX* der Ausdruck $R(x)$ das eindeutig bestimmte Element y , für das $\langle x, y \rangle \in R$ gilt. Das Programm `function.stl` in Abbildung 2.7 auf Seite 13 zeigt dies konkret. Zusätzlich zeigt das Programm noch, dass in *SetIX* bei einer binären Relation $\text{dom}(R)$ als `domain(R)` und $\text{rng}(R)$ als `range(R)` geschrieben wird. Außerdem sehen wir in Zeile 2, dass wir den Wert einer funktionalen Relation durch eine Zuweisung ändern können.

```

1  q := { [n, n**2] : n in {1..10} };
2  q(5) := 7;
3  print( "q(3)   = ", q(3)       );
4  print( "q(5)   = ", q(5)       );
5  print( "dom(q) = ", domain(q) );
6  print( "rng(q) = ", range(q)  );
7  print( "q = ", q );

```

Abbildung 2.7: Rechnen mit binären Relationen.

Das Programm berechnet in Zeile 1 die binäre Relation q so, dass q die Funktion $x \mapsto x * x$ auf der Menge $\{n \in \mathbb{N} \mid 1 \leq n \wedge n \leq 10\}$ repräsentiert.

An dieser Stelle eine Warnung: In *SetIX* müssen alle Variablen mit einem kleinen Buchstaben beginnen! Normalerweise hätte ich die Relation `q` mit einem großen `Q` bezeichnen wollen, aber das geht nicht, denn alle Namen, die mit einem großen Buchstaben beginnen, sind reserviert.

In Zeile 2 wird die Relation an der Stelle $x = 5$ so abgeändert, dass nun $q(5)$ den Wert 7 hat. Anschließend werden noch $\text{dom}(Q)$ und $\text{rng}(Q)$ berechnet. Das Programm liefert die folgende Ausgabe:

```

q(3)   = 9
q(5)   = 7
dom(q) = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
rng(q) = {1, 4, 7, 9, 16, 36, 49, 64, 81, 100}
q = {[1, 1], [2, 4], [3, 9], [4, 16], [5, 7], [6, 36], [7, 49], [8, 64],
     [9, 81], [10, 100]}

```

Es ist naheliegend zu fragen, was bei der Auswertung eines Ausdrucks der Form $R(x)$ passiert, wenn die Menge $\{y \mid \langle x, y \rangle \in R\}$ entweder leer ist oder aber aus mehr als einem Element besteht. Das Programm `buggy-function.stl` in Abbildung 2.8 auf Seite 14 beantwortet diese Frage auf experimentellem Wege.

Falls die Menge $\{y \mid \langle x, y \rangle \in r\}$ entweder leer ist oder mehr als eine Element enthält, so ist der Ausdruck $r(x)$ undefiniert. Ein solcher Ausdruck wird in *SetIX* durch den String "om" dargestellt. Der Versuch, einen undefinierten Wert in eine Menge M einzufügen, ändert diese Menge nicht,

```

1  r := { [1, 1], [1, 4], [3, 3] };
2  print( "r(1) = ", r(1) );
3  print( "r(2) = ", r(2) );
4  print( "{ r(1), r(2) } = ", { r(1), r(2) } );
5  print( "r{1} = ", r{1} );
6  print( "r{2} = ", r{2} );

```

Abbildung 2.8: Rechnen mit nicht-funktionalen binären Relationen.

es gibt aber auch keine Fehlermeldung. Deswegen wird in Zeile 4 für die Menge $\{ r(1), r(2) \}$ einfach die leere Menge ausgegeben.

Will man das Auftreten von undefinierten Werten beim Auswerten einer binären Relation r vermeiden, so gibt es in *SetIX* die Möglichkeit, $r\{x\}$ statt $r(x)$ zu schreiben, die runden Klammern werden also durch geschweifte Klammern ersetzt. Der Ausdruck $r\{x\}$ ist für eine binäre Relation r wie folgt definiert:

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

Daher liefert das Programm aus Abbildung 2.8 die folgende Ausgabe:

```

r(1) = om
r(2) = om
{ r(1), r(2) } = {}
r{1} = {1, 4}
r{2} = {}

```

2.4 Allgemeine Tupel

Auch beliebige n -Tupel lassen sich in *SetIX* darstellen. Diese können ganz analog zu Mengen definiert werden. Das geht denkbar einfach: Es müssen nur alle geschweiften Klammern der Form “{” und “}” durch die entsprechenden eckigen Klammern “[” und “]” ersetzt werden. Dann können Tupel durch explizite Aufzählung, arithmetische Aufzählung, Iteration und das Auswahlprinzip in der selben Weise wie Mengen gebildet werden. Das Programm in Abbildung 2.9 zeigt ein Beispiel. Dieses Programm berechnet die Primzahlen nach dem selben Verfahren wie das Programm in Abbildung 2.6 auf Seite 12, benutzt aber Listen sowohl zur Darstellung der Menge der Primzahlen als auch zur Darstellung der Menge der Teiler.

```

1  teiler := procedure(p) {
2      return [ t in {1..p} | p % t == 0 ];
3  };
4
5  n := 100;
6  primes := [ p in [2 .. n] | teiler(p) == {1, p} ];
7  print(primes);

```

Abbildung 2.9: Berechnung der Primzahlen mit Tupeln.

2.5 Spezielle Funktionen und Operatoren auf Mengen

Das Programm in Abbildung 2.10 auf Seite 15 zeigt ein einfaches Verfahren um eine Liste von Zahlen zu sortieren. Der Ausdruck

```
max(s)
```

berechnet das größte Element der Liste s . Damit läuft die Variable n in dem Iterator

```
n in [1 .. max(s)]
```

dann von 1 bis zur größten in s auftretenden Zahl. Aufgrund der Bedingung “ n in s ” wird die Zahl n genau dann in die resultierende Liste eingefügt, wenn n ein Element der Liste s ist. Da der Iterator “ n in [1 .. max(s)]” die Zahlen der Reihe nach aufzählt, ist das Ergebnis eine sortierte Liste, die genau die Zahlen enthält, die Elemente von s sind.

Offensichtlich ist der in der Prozedur `sort()` implementierte Algorithmus nicht sehr effizient. Wir werden später noch effizientere Algorithmen diskutieren.

```
1  sort := procedure(s) {
2      return [ n in [1 .. max(s)] | n in s ];
3  };
4  s := [ 13, 5, 7, 2, 4 ];
5  print( "sort( ", s, " ) = ", sort(s) );
```

Abbildung 2.10: Sortieren einer Menge.

Analog zu der Funktion `max()` gibt es noch die Funktion `min()` die das Minimum einer Menge oder Liste berechnet.

Weiterhin können die Operatoren “+” und “*” auf Mengen angewendet werden. Der Operator “+” berechnet die Summe aller Elemente einer Menge, während der Operator “*” das Produkt der Elemente berechnet. Ist die zu Grunde liegende Menge oder Liste leer, so gibt der Operator “+” als Ergebnis 0 zurück, während der Operator “*” als Ergebnis eine 1 liefert.

Als nächstes besprechen wir die Funktion “`from`”, mit dem wir ein (nicht näher spezifiziertes) Element aus einer Menge auswählen können. Die Syntax ist:

```
x := from(s);
```

Hierbei ist s eine Menge und x eine Variable. Wird diese Anweisung ausgeführt, so wird ein nicht näher spezifiziertes Element aus der Menge s entfernt. Dieses Element wird darüber hinaus der Variablen x zugewiesen. Falls s leer ist, so erhält x den undefinierten Wert “om” und s bleibt unverändert. Das Programm `from.stl` in Abbildung 2.11 auf Seite 16 nutzt diese Anweisung um eine Menge elementweise auszugeben. Jedes Element wird dabei in einer eigenen Zeile ausgedruckt.

Neben der Funktion “`from`” gibt es noch die Funktion “`arb`”, die ein beliebiges Element aus einer Menge auswählt, die Menge selbst aber unverändert läßt. Nach den Befehlen

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

erhalten wir die folgende Ausgabe:

```
x = 13
s = {2, 3, 5, 7, 13}
```

Weiterhin steht für Listen der Operator “+” zur Verfügung, mit dem zwei Listen aneinander gehängt werden können. Außerdem gibt es noch den unären Operator “#”, der für Mengen und

```

1  printSet := procedure(s) {
2      if (s == {}) {
3          return;
4      }
5      x := from(s);
6      print(x);
7      printSet(s);
8  };
9  s := { 13, 5, 7, 2, 4 };
10 printSet(s);

```

Abbildung 2.11: Menge elementweise ausdrucken.

Listen die Anzahl der Elemente berechnet. Schließlich kann man Elemente von Listen mit der Schreibweise

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

indizieren. In diesem Fall muss t eine Liste sein, die mindestens die Länge n hat. Die obige Anweisung weist der Variablen x dann den Wert des n -ten Elementes der Liste t zu. Die obige Zuweisung läßt sich auch umdrehen: Mit

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

wird die Liste t so abgeändert, dass das n -te Element danach den Wert x hat. Im Gegensatz zu der Sprache C werden in *SetIX* Listen mit 1 beginnend indiziert, falls wir die beiden Befehle

```

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

```

ausführen, hat x also anschließend den Wert 1.

Das Programm `simple-tuple.stlx` in Abbildung 2.12 auf Seite 17 demonstriert die eben vorgestellten Operatoren. Zusätzlich sehen wir in Zeile 19, dass *SetIX* simultane Zuweisungen unterstützt. Das Programm produziert die folgende Ausgabe:

```

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

```

2.5.1 Anwendung: *Sortieren durch Auswahl*

Als praktische Anwendung zeigen wir eine Implementierung des Algorithmus *Sortieren durch Auswahl*. Dieser Algorithmus, dessen Aufgabe es ist, eine gegebene Liste L zu sortieren, kann wie folgt beschrieben werden:

1. Falls L leer ist, so ist auch $\text{sort}(L)$ leer:

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

2. Sonst berechnen wir das Minimum m von L :

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

```

1  a := [ 1, 2, 3 ];
2  b := [ 2, 3, 4, 5, 6 ];
3  c := { 5, 6, 7 };
4  // Aneinanderhängen von Tupeln mit +
5  print(a, " + ", b, " = ", a + b);
6  // Berechnung der Anzahl der Elemente einer Menge
7  print("# ", c, " = ", # c);
8  // Berechnung der Länge eines Tupels
9  print("# ", a, " = ", # a);
10 // Auswahl des dritten Elements von b
11 print(b, "(3) = ", b(3) );
12 // Überschreiben des 10. Elements von b
13 b(10) := 42;
14 print("b = ", b);
15 // Auswahl einer Teilliste
16 d := b(2..4);
17 print( "d = ", d);
18 x := 1; y := 2;
19 [ x, y ] := [ y, x ];
20 print("x = ", x, ", y = ", y);

```

Abbildung 2.12: Weitere Operatoren auf Tupeln und Mengen.

Dann entfernen wir m aus der Liste L und sortieren die Restliste rekursiv:

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

Abbildung 2.13 auf Seite 17 zeigt die Umsetzung dieser Idee in *SetLX*.

```

1  minSort := procedure(l) {
2      if (l == []) {
3          return [];
4      }
5      m := min(l);
6      return [ m ] + minSort( [ x in l | x != m ] );
7  };
8
9  l := [ 13, 5, 13, 7, 2, 4 ];
10 print( "sort( ", l, " ) = ", minSort(l) );

```

Abbildung 2.13: Implementierung des Algorithmus *Sortieren durch Auswahl*.

2.6 Kontroll-Strukturen

Die Sprache *SetIX* stellt alle Kontroll-Strukturen zur Verfügung, die in modernen Sprachen üblich sind. Wir haben “if”-Abfragen bereits mehrfach gesehen. In der allgemeinsten Form hat eine Fallunterscheidung die in Abbildung 2.14 auf Seite 18 gezeigte Struktur.

```
1   if (test0) {  
2       body0  
3   } else if (test1) {  
4       body1  
5       :  
6   } else if (testn) {  
7       bodyn  
8   } else {  
9       bodyn+1  
10  }
```

Abbildung 2.14: Struktur der allgemeinen Fallunterscheidung.

Hierbei steht $test_i$ für einen Test, der “true” oder “false” liefert. Liefert der Test “true”, so wird der zugehörigen Anweisungen in $body_i$ ausgeführt, andernfalls wird der nächste Test $test_{i+1}$ versucht. Schlagen alle Tests fehl, so wird $body_{n+1}$ ausgeführt.

Die Tests selber können dabei die binären Operatoren

“==”, “!=”, “>”, “<”, “>=”, “<=”, “in”,

verwenden. Dabei steht “==” für den Vergleich auf Gleichheit, “!=” für den Vergleich auf Ungleichheit. Für Zahlen führen die Operatoren “>”, “<”, “>=” und “<=” die selben Größenvergleiche durch wie in der Sprache *C*. Für Mengen überprüfen diese Operatoren analog, ob die entsprechenden Teilmengen-Beziehung erfüllt ist. Der Operator “in” überprüft, ob das erste Argument ein Element der als zweites Argument gegebene Menge ist: Der Test

$x \text{ in } S$

hat genau dann den Wert **true**, wenn $x \in S$ gilt. Aus den einfachen Tests, die mit Hilfe der oben vorgestellten Operatoren definiert werden können, können mit Hilfe der Junktoren “&&” (logisches *und*), “||” (logisches *oder*) und “not” (logisches *nicht*) komplexere Tests aufgebaut werden. Dabei bindet der Operator “||” am schwächsten und der Operator “!” bindet am stärksten. Ein Ausdruck der Form

`!a == b && b < c || x >= y`

wird also als

`((!(a == b)) && b < c) || x >= y`

geklammert.

SetIX unterstützt auch die Verwendung von Quantoren. Die Syntax für die Verwendung des Allquantors ist

`forall (x in s | cond)`

Hier ist s eine Menge und $cond$ ist ein Ausdruck, der von der Variablen x abhängt und einen Wahrheitswert als Ergebnis zurück liefert. Die Auswertung des oben angegebenen Allquantors liefert genau dann **true** wenn die Auswertung von $cond$ für alle Elemente der Menge s den Wert **true** ergibt. Abbildung 2.15 auf Seite 19 zeigt eine Verwendung des Allquantors zur Berechnung

von Primzahlen. Die Bedingung

```
forall (x in divisors(p) | x in {1, p})
```

trifft auf genau die Zahlen p zu, für die gilt, dass alle Teiler Elemente der Menge $\{1, p\}$ sind. Dies sind genau die Primzahlen.

```
1  isPrime := procedure(p) {
2      return forall (x in divisors(p) | x in {1, p});
3  };
4  divisors := procedure(p) {
5      return { t in {1..p} | p % t == 0 };
6  };
7  n := 100;
8  primes := [ p in [2..n] | isPrime(p) ];
9  print( primes );
```

Abbildung 2.15: Berechnung der Primzahlen mit Hilfe eines Allquantors

Neben dem Allquantor gibt es noch den Existenzquantor. Die Syntax ist:

```
exists (x in s | cond)
```

Hierbei ist wieder eine s eine Menge und $cond$ ist ein Ausdruck, der zu **true** oder **false** ausgewertet werden kann. Falls es wenigstens ein x gibt, so dass die Auswertung von $cond$ true ergibt, liefert die Auswertung des Existenzquantor ebenfalls den Wert **true**. Zusätzlich wird in diesem Fall der Variablen x ein Wert zugewiesen, für den die Bedingung $cond$ erfüllt ist. Falls die Auswertung des Existenzquantors den Wert **false** ergibt, ändert sich der Wert von x nicht.

Case-Blöcke

Als Alternative zur Fallunterscheidung mit Hilfe von **if-then-else**-Konstrukten gibt es noch den **case**-Block. Ein solcher Block hat die in Abbildung 2.16 auf Seite 19 gezeigte Struktur. Bei der Abarbeitung werden der Reihe nach die Tests $test_1, \dots, test_n$ ausgewertet. Für den ersten Test $test_i$, dessen Auswertung den Wert **true** ergibt, wird der zugehörige Block $body_i$ ausgeführt. Nur dann, wenn alle Tests $test_1, \dots, test_n$ scheitern, wird der Block $body_{n+1}$ hinter dem Schlüsselwort **otherwise** ausgeführt. Den selben Effekt könnte man natürlich auch mit einer **if-elseif-...-elseif-else-end if** Konstruktion erreichen, nur ist die Verwendung eines **case**-Blocks oft übersichtlicher.

```
1  switch {
2      case test1 : body1
3      :
4      case testn : bodyn
5      default : bodyn+1
6  }
```

Abbildung 2.16: Struktur eines Case-Blocks

Abbildung 2.17 zeigt eine (zugegebenermaßen triviale) Anwendung eines **case**-Blocks, bei der es darum geht, in Abhängigkeit von der letzte Ziffer einer Zahl eine Meldung auszugeben. Bei der Behandlung der Aussagenlogik werden wir noch realistischere Anwendungs-Beispiele für den **case**-Block kennenlernen.

```

1  print("Zahl eingeben:");
2  n := read();
3  m := n % 10;
4  switch {
5      case m == 0 : print("letzte Ziffer ist 0");
6      case m == 1 : print("letzte Ziffer ist 1");
7      case m == 2 : print("letzte Ziffer ist 2");
8      case m == 3 : print("letzte Ziffer ist 3");
9      case m == 4 : print("letzte Ziffer ist 4");
10     case m == 5 : print("letzte Ziffer ist 5");
11     case m == 6 : print("letzte Ziffer ist 6");
12     case m == 7 : print("letzte Ziffer ist 7");
13     case m == 8 : print("letzte Ziffer ist 8");
14     case m == 9 : print("letzte Ziffer ist 9");
15     default      : print("impossible");
16 }

```

Abbildung 2.17: Anwendung eines `case`-Blocks

In der Sprache C gibt es eine analoge Konstruktion. In C ist es so, dass nach einem Block, der nicht durch einen `break`-Befehl abgeschlossen wird, auch alle folgenden Blocks ausgeführt werden. Dies ist in *SetIX* anders: Dort wird immer genau ein Block ausgeführt.

2.6.1 Schleifen

Es gibt in *SetIX* Kopf-gesteuerte Schleifen (`while`-Schleifen) und Schleifen, die über die Elemente einer Menge iterieren (`for`-Schleifen), sowie Schleifen, bei denen die Abbruch-Bedingung Teil des Rumpfs ist (`loop`-Schleifen). Wir diskutieren diese Schleifenformen jetzt im Detail.

`while`-Schleifen

Die allgemeine Syntax der `while`-Schleife ist in Abbildung 2.18 auf Seite 20 gezeigt. Hierbei ist *test* ein Ausdruck, der zu Beginn ausgewertet wird und der “`true`” oder “`false`” ergeben muss. Ergibt die Auswertung “`false`”, so ist die Auswertung der `while`-Schleife bereits beendet. Ergibt die Auswertung allerdings “`true`”, so wird anschließend *body* ausgewertet. Danach beginnt die Auswertung der Schleife dann wieder von vorne, d.h. es wird wieder *test* ausgewertet und danach wird abhängig von dem Ergebnis dieser wieder *body* ausgewertet. Das ganze passiert so lange, bis irgendwann einmal die Auswertung von *test* den Wert “`false`” ergibt.

```

while (test) {
    body
};

```

Abbildung 2.18: Struktur der `while`-Schleife

Abbildung 2.19 auf Seite 21 zeigt eine Berechnung von Primzahlen mit Hilfe einer `while`-Schleife. Hier ist die Idee, dass eine Zahl genau dann Primzahl ist, wenn es keine kleinere Primzahl gibt, die diese Zahl teilt.

```

1  n := 100;
2  primes := {};
3  p := 2;
4  while (p <= n) {
5      if (forall (t in primes | p % t != 0)) {
6          print(p);
7          primes := primes + { p };
8      }
9      p := p + 1;
10 }

```

Abbildung 2.19: Iterative Berechnung der Primzahlen.

for-Schleifen

Die allgemeine Syntax der **for**-Schleife ist in Abbildung 2.20 auf Seite 21 gezeigt. Hierbei ist s eine Menge, und x der Name einer Variablen. Diese Variable wird nacheinander mit allen Werten aus der Menge s belegt und anschließend wird mit dem jeweiligen Wert von x der Schleifenrumpf *body* ausgeführt. Anstelle einer Menge kann s auch eine Liste sein.

```

for (x in s) {
    body
};

```

Abbildung 2.20: Struktur der **for**-Schleife.

Abbildung 2.21 auf Seite 22 zeigt eine Berechnung von Primzahlen mit Hilfe einer **for**-Schleife. Der dabei verwendete Algorithmus ist als *Sieb des Eratosthenes* bekannt. Das funktioniert wie folgt: Sollen alle Primzahlen kleiner oder gleich n berechnet werden, so wird zunächst ein Tupel der Länge n gebildet, dessen i -tes Element den Wert i hat. Das passiert in Zeile 3. Anschließend werden alle Zahlen, die Vielfache von 2, 3, 4, \dots sind, aus der Menge der Primzahlen entfernt, indem die Zahl, die an dem entsprechenden Index in der Liste **primes** steht, auf 0 gesetzt wird. Dazu sind zwei Schleifen erforderlich: Die äußere **for**-Schleife iteriert i über alle Werte von 2 bis n . Die innere **while**-Schleife iteriert dann für gegebenes i über alle Werte $i \cdot j$, für die das Produkt $i \cdot j \leq n$ ist. Schließlich werden in der letzten **for**-Schleife in den Zeilen 14 bis 18 alle die Indizes i ausgedruckt, für die **primes**(i) nicht auf 0 gesetzt worden ist, denn das sind genau die Primzahlen.

Der Algorithmus aus Abbildung 2.21 kann durch die folgende Beobachtungen noch verbessert werden:

1. Es reicht aus, wenn j mit i initialisiert wird, denn alle kleineren Vielfachen wurden bereits vorher auf 0 gesetzt.
2. Falls in der äußeren Schleife die Zahl i keine Primzahl ist, so bringt es nichts mehr, die innere **while**-Schleife in den Zeilen 9 bis 11 zu durchlaufen, denn alle Indizes, für die dort **primes**($i \cdot j$) auf 0 gesetzt wird, sind schon bei dem vorherigen Durchlauf der äußeren Schleife, bei der **primes**(i) auf 0 gesetzt wurde, zu 0 gesetzt worden. Abbildung 2.22 auf Seite 22 zeigt den resultierenden Algorithmus. Um den Durchlauf der inneren while Schleife in dem Fall, dass **primes**(i) = 0 ist, zu überspringen, haben wir den Befehl “**continue**” benutzt. Der Aufruf von “**continue**” bricht die Abarbeitung des Schleifen-Rumpfs für den aktuellen Wert von i ab, weist der Variablen i den nächsten Wert aus $[1..n]$ zu und fährt dann mit der Abarbeitung der Schleife in Zeile 4 fort. Der Befehl “**continue**” verhält sich also genauso, wie der Befehl “**continue**” in der Sprache C.

```
1  n := 100;
2  primes := [1 .. n];
3  for (i in [2 .. n]) {
4      j := 2;
5      while (i * j <= n) {
6          primes(i * j) := 0;
7          j := j + 1;
8      }
9  }
10 for (i in [2 .. n]) {
11     if (primes(i) > 0) {
12         print(i);
13     }
14 }
```

Abbildung 2.21: Berechnung der Primzahlen nach Eratosthenes.

```
1  n := 10000;
2  primes := [1 .. n];
3  for (i in [2 .. n/2]) {
4      if (primes(i) == 0) {
5          continue;
6      }
7      j := i;
8      while (i * j <= n) {
9          primes(i * j) := 0;
10         j := j + 1;
11     }
12 }
13 for (i in [2 .. n]) {
14     if (primes(i) > 0) {
15         print(i);
16     }
17 }
```

Abbildung 2.22: Effizientere Berechnung der Primzahlen nach Eratosthenes.

2.6.2 Fixpunkt-Algorithmen

Angenommen, wir wollen in der Menge \mathbb{R} der reellen Zahlen die Gleichung

$$x = \cos(x)$$

lösen. Ein naives Verfahren, das hier zum Ziel führt, basiert auf der Beobachtung, dass die Folge $(x_n)_n$, die durch

$$x_0 := 0 \text{ und } x_{n+1} := \cos(x_n) \text{ für alle } n \in \mathbb{N}$$

definiert ist, gegen eine Lösung der obigen Gleichung konvergiert. Damit führt der in Abbildung 2.23 auf Seite 23 angegebene Algorithmus zum Ziel.

```
1  x := 0.0;
2  while (true) {
3      old_x := x;
4      x := cos(x);
5      print(x);
6      if (abs(x - old_x) < 1.0e-13) {
7          print("x = ", x);
8          break;
9      }
10 }
```

Abbildung 2.23: Lösung der Gleichung $x = \cos(x)$ durch Iteration.

Bei dieser Implementierung wird die Schleife in dem Moment abgebrochen, wenn die Werte von `x` und `old_x` nahe genug beieinander liegen. Dieser Test kann aber am Anfang der Schleife noch gar nicht durchgeführt werden, weil da die Variable `old_x` noch gar keinen Wert hat. Daher brauchen wir hier das Kommando “`break`”. Dieses bricht die Schleife ab. In der Sprache *C* gibt es das Kommando “`break`” ebenfalls.

Ganz nebenbei zeigt das obige Beispiel auch, dass Sie in *SetlX* nicht nur mit ganzen, sondern auch mit reellen Zahlen rechnen können. Eine Zahlen-Konstante, die den Punkt “.” enthält, wird automatisch als reelle Zahl erkannt und auch so abgespeichert. In *SetlX* stehen unter anderem die folgenden reellen Funktionen zur Verfügung:

1. Der Ausdruck `sin(x)` berechnet den Sinus von x . Außerdem stehen die trigonometrischen Funktionen `cos(x)` und `tan(x)` zur Verfügung. Die Umkehr-Funktionen der trigonometrischen Funktionen sind `asin(x)`, `acos(x)` und `atan(x)`.

Der Sinus Hyperbolicus von x wird durch `sinh(x)` berechnet. Entsprechend berechnet `cosh(x)` den Kosinus Hyperbolicus und `tanh(x)` den Tangens Hyperbolicus.

2. Der Ausdruck `exp(x)` berechnet die Potenz zur Basis e , es gilt

$$\exp(x) = e^x.$$

3. Der Ausdruck `log(x)` berechnet den natürlichen Logarithmus von x . Der Logarithmus zur Basis 10 von x wird durch `log10(x)` berechnet.
4. Der Ausdruck `abs(x)` berechnet den Absolut-Betrag von x , während `signum(x)` das Vorzeichen von x berechnet.

5. Der Ausdruck `sqrt(x)` berechnet die Quadrat-Wurzel von x , es gilt

$$\text{sqrt}(x) = \sqrt{x}.$$

6. Der Ausdruck `cbirt(x)` berechnet die dritte Wurzel von x , es gilt

$$\text{cbirt}(x) = \sqrt[3]{x}.$$

7. Der Ausdruck `ceil(x)` berechnet die kleinste ganze Zahl, die größer oder gleich x ist, es gilt

$$\text{ceil}(x) = \min(\{z \in \mathbb{Z} \mid z \geq x\}).$$

8. Der Ausdruck `floor(x)` berechnet die größte ganze Zahl, die kleiner oder gleich x ist, es gilt

$$\text{floor}(x) = \max(\{z \in \mathbb{Z} \mid z \leq x\}).$$

9. Der Ausdruck `round(x)` rundet x zu einer ganzen Zahl.

2.6.3 Verschiedenes

Der Interpreter bietet die Möglichkeit, komplexe Programme zu laden. Der Befehl

```
load(file);
```

lädt das Programm, das sich in der Datei *file* befindet und führt die in dem Programm vorhandenen Befehle aus. Führen wir beispielsweise den Befehl

```
load("primes-forall.stlx");
```

im Interpreter aus und enthält die Datei “primes-forall.stlx” das in Abbildung 2.15 auf Seite 19 gezeigte Programm, so können wir anschließend mit den in dieser Datei definierten Variablen arbeiten. Beispielsweise liefert der Befehl

```
print(isPrime);
```

die Ausgabe:

```
procedure (p) { return forall (x in divisors(p) | x in {1, p}); }
```

Zeichenketten, auch bekannt als *Strings*, werden in *SetLX* in doppelte Hochkommata gesetzt. Der Operator “+” kann dazu benutzt werden, zwei Strings aneinander zu hängen, der Ausdruck

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

liefert also das Ergebnis

```
"abcuvw".
```

Zusätzlich kann eine natürliche Zahl n mit einem String s über den Multiplikations-Operator “*” verknüpft werden. Der Ausdruck

```
n * s;
```

liefert als Ergebnis die n -malige Verkettung von s . Beispielsweise ist das Ergebnis von

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

der String

```
"abcabcabc".
```

2.7 Fallstudie: Berechnung von Wahrscheinlichkeiten

Wir wollen in diesem kurzen Abschnitt zeigen, wie sich Wahrscheinlichkeiten für die Poker-Variante *Texas Hold'em* berechnen lassen. Bei dieser Poker-Variante gibt es insgesamt 52 Karten. Jeder dieser Karten hat eine Farbe, die ein Element der Menge

$$\text{suits} = \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$$

ist und außerdem einen Wert hat, der ein Element der Menge

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

ist. Die einzelnen Karten können dann mathematisch als Paare dargestellt werden. Die gesamte Menge von Karten wäre dann beispielsweise wie folgt gegeben:

$$deck = \{\langle v, s \rangle \mid v \in values \wedge s \in suits\}.$$

Jeder Spieler bekommt zunächst zwei Karten. Diese beiden Karten werden als *Preflop* oder *Hole* bezeichnet. Nach einer Bietphase werden anschließend drei weitere Karten, der sogenannte *Flop*, auf den Tisch gelegt. Wir nehmen an, dass ein Spieler zunächst die Karten $\{\langle 3, \clubsuit \rangle, \langle 3, \spadesuit \rangle\}$ bekommen hat und nun wissen möchte, wie groß die Wahrscheinlichkeit dafür ist, dass beim Flop eine weitere 3 auf den Tisch gelegt wird. Um diese Wahrscheinlichkeit zu berechnen, muss der Spieler die Anzahl aller Flops, bei denen eine weitere 3 erscheinen kann, durch die gesamte Anzahl der Flops teilen. Das in Abbildung 2.24 gezeigte Programm führt diese Berechnung durch.

```

1  suits := { "c", "h", "d", "s" };
2  values := { "2", "3", "4", "5", "6", "7", "8", "9",
3             "T", "J", "Q", "K", "A" };
4  deck := { [ v, s ] : v in values, s in suits };
5  hole := { [ "3", "c" ], [ "3", "s" ] };
6  rest := deck - hole;
7  flops := { { k1, k2, k3 } : k1 in rest, k2 in rest, k3 in rest
8                  | #{ k1, k2, k3 } == 3 };
9  print(#flops);
10 trips := { f in flops | [ "3", "d" ] in f || [ "3", "h" ] in f };
11 print(1.0 * #trips / #flops);

```

Abbildung 2.24: Berechnung von Wahrscheinlichkeiten im Poker

1. In Zeile 1 definieren wir die Menge `suits` der Farben einer Karte.
2. In Zeile 2 und 3 definieren wir die Menge `values` der möglichen Werte einer Karte.
3. In Zeile 4 stellen wir die Menge aller Karten, die wir mit `deck` bezeichnen, als Menge von Paaren dar, wobei die erste Komponente der Paare den Wert und die zweite Komponente die Farbe der Karte angibt.
4. Die in Zeile 5 definierte Menge `hole` stellt die Menge der Karten des Spielers dar.
5. Die verbleibenden Karten bezeichnen wir in Zeile 6 als `rest`.
6. In Zeile 7 und 8 berechnen wir die Menge aller möglichen Flops. Da die Reihenfolge der Karten im Flop keine Rolle spielt, verwenden wir zur Darstellung der einzelnen Flops eine Menge. Dabei müssen wir aber darauf achten, dass der Flop auch wirklich aus drei verschiedenen Karten besteht. Daher haben wir bei der Auswahl der Menge die Bedingung

$$\# \{ k1, k2, k3 \} = 3$$

zu beachten.

7. Die Teilmenge der Flops, in denen mindestens eine weitere 3 auftritt, wird in Zeile 10 berechnet.
8. Schließlich berechnet sich die Wahrscheinlichkeit für eine 3 im Flop als das Verhältnis der Anzahl der günstigen Fälle zu der Anzahl der möglichen Fälle. Wir müssen also nur die Anzahl der Elemente der entsprechenden Mengen ins Verhältnis setzen.

Hier gibt es aber noch eine Klippe, die umschifft werden muss: Da die Variablen `#Trips` und `#Flops` mit ganzen Zahlen belegt sind, würde die Division `#Trips / #Flops` als ganzzahlige Division ausgeführt und das Ergebnis wäre 0! Daher ist es erforderlich, zunächst eine der beiden Zahlen in eine Fließkomma-Zahl zu konvertieren, damit der Quotienten als Fließkomma-Zahl berechnet wird. Dies können wir beispielsweise dadurch erreichen, dass wir den Dividenten `#Trips` mit der Fließkomma-Zahl 1.0 multiplizieren.

Das Problem, dass bei ganzen Zahlen der Operator “/” an Stelle einer Fließkomma-Division eine ganzzahlige durchführt gibt es auch in vielen anderen Programmier-Sprachen.

Lassen wir das Programm laufen, so sehen wir, dass die Wahrscheinlichkeit, ein Pocket-Paar im Flop auf Trips zu verbessern, bei etwa 11,8% liegt.

Bemerkung: Die Berechnung von Wahrscheinlichkeiten ist in der oben dargestellten Weise nur dann möglich, wenn die im Laufe der Berechnung auftretenden Mengen so klein sind, dass sie sich explizit im Rechner darstellen lassen. Wenn diese Voraussetzung nicht mehr erfüllt ist, können die gesuchten Wahrscheinlichkeiten mit Hilfe der im zweiten Semester vorgestellten *Monte-Carlo-Methode* berechnet werden.

2.8 Fallstudie: Berechnung von Pfaden

Wir wollen dieses Kapitel mit einer praktisch relevanten Anwendung der Sprache *SetIX* abschließen. Dazu betrachten wir das Problem, Pfade in einem *Graphen* zu bestimmen. Abstrakt gesehen beinhaltet ein Graph die Information, zwischen welchen Punkten es direkte Verbindungen gibt. Zur Vereinfachung wollen wir zunächst annehmen, dass die einzelnen Punkte durch Zahlen identifiziert werden. Dann können wir eine direkte Verbindung zwischen zwei Punkten durch ein Paar von Zahlen darstellen. Den Graphen selber stellen wir als eine Menge solcher Paaren dar. Wir betrachten ein Beispiel. Sei R wie folgt definiert:

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

In diesem Graphen haben wir die Punkte 1, 2, 3, 4 und 5. Eine Darstellung dieses Graphen finden Sie in Abbildung 2.25. Beachten Sie, dass die Verbindungen in diesem Graphen *Einbahn-Straßen* sind: Wir haben zwar eine Verbindung von 1 nach 2, aber keine Verbindung von 2 nach 1.

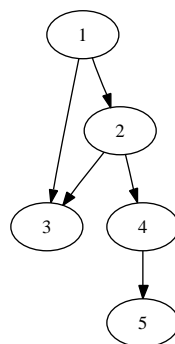


Abbildung 2.25: Ein einfacher Graph.

In dem Graphen sind nur die unmittelbaren Verbindungen zwischen zwei Punkten verzeichnet. Es gibt aber unter Umständen auch noch andere Verbindungen. Beispielsweise gibt es eine unmittelbare Verbindung von 1 nach 3. Es gibt aber auch noch einen Pfad von 1 nach 3, der über den Punkt 2 geht. Unser Ziel in diesem Abschnitt ist es einen Algorithmus zu entwickeln, der überprüft, ob zwischen zwei Punkten eine Verbindung existiert und gegebenenfalls berechnet. Dazu entwickeln wir zunächst einen Algorithmus, der nur überprüft, ob es eine Verbindung zwischen zwei Punkten gibt und erweitern diesen Algorithmus dann später so, dass die Verbindung auch berechnet wird.

2.8.1 Berechnung des transitiven Abschlusses einer Relation

Als erstes bemerken wir, dass ein Graph R nichts anderes ist als eine binäre Relation. Um feststellen zu können, ob es zwischen zwei Punkten eine Verbindung gibt, müssen wir den transitiven Abschluß R^+ der Relation R bilden. Wir erinnern daran, dass wir in der Mathematik-Vorlesung gesehen haben, dass R^+ wie folgt berechnet werden kann:

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

Auf den ersten Blick betrachtet sieht diese Formel so aus, als ob wir unendlich lange rechnen müßten. Aber versuchen wir einmal, diese Formel anschaulich zu verstehen. Zunächst steht da R . Das sind die Verbindungen, die unmittelbar gegeben sind. Als nächstes steht dort R^2 und das ist $R \circ R$. Es gilt aber

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

In R^2 sind also alle die Pfade enthalten, die aus zwei direkten Verbindungen zusammengesetzt sind. Allgemein läßt sich durch Induktion sehen, dass R^n alle die Pfade enthält, die aus n direkten Verbindungen zusammengesetzt sind. Nun ist die Zahl der Punkte, die wir haben, endlich. Sagen wir mal, dass es k Punkte sind. Dann macht es keinen Sinn solche Pfade zu betrachten, die aus mehr als $k - 1$ direkten Verbindungen zusammengesetzt sind, denn wir wollen ja nicht im Kreis herum laufen. Damit kann dann aber die Formel zur Berechnung des transitiven Abschlusses vereinfacht werden:

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

Diese Formel könnten wir tatsächlich so benutzen. Es ist aber noch effizienter, einen Fixpunkt-Algorithmus zu verwenden. Dazu zeigen wir zunächst, dass der transitive Abschluß R^+ die folgende Fixpunkt-Gleichung erfüllt:

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

Wir erinnern hier daran, dass wir vereinbart haben, dass der Operator \circ stärker bindet als der Operator \cup , so dass der Ausdruck $R \cup R \circ R^+$ als $R \cup (R \circ R^+)$ zu lesen ist. Die Fixpunkt-Gleichung 2.1 läßt sich algebraisch beweisen. Es gilt

$$\begin{aligned} & R \cup R \circ R^+ \\ &= R \cup R \circ \bigcup_{i=1}^{\infty} R^i \\ &= R \cup R \circ (R^1 \cup R^2 \cup R^3 \cup \dots) \\ &= R \cup (R \circ R^1 \cup R \circ R^2 \cup R \circ R^3 \cup \dots) \quad \text{Distributiv-Gesetz} \\ &= R \cup (R^2 \cup R^3 \cup R^4 \cup \dots) \quad \text{Potenz-Gesetz} \\ &= R^1 \cup (R^2 \cup R^3 \cup R^4 \cup \dots) \\ &= \bigcup_{i=1}^{\infty} R^i \\ &= R^+ \end{aligned}$$

Die Gleichung 2.1 kann benutzt werden um den transitiven Abschluß iterativ zu berechnen. Wir definieren eine Folge $(T_n)_{n \in \mathbb{N}}$ durch Induktion folgt:

$$\text{I.A. } n = 1: \quad T_1 := R$$

$$\text{I.S. } n \mapsto n + 1: \quad T_{n+1} := R \cup R \circ T_n.$$

Die Relationen T_n lassen sich auf die Relation R zurückführen:

1. $T_1 = R$.
2. $T_2 = R \cup R \circ T_1 = R \cup R \circ R = R^1 \cup R^2$.
3. $T_3 = R \cup R \circ T_2$
 $= R \cup R \circ (R^1 \cup R^2)$
 $= R^1 \cup R^2 \cup R^3$.

Allgemein können wir durch vollständige Induktion über $n \in \mathbb{N}$ beweisen, dass

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

gilt. Der Induktions-Anfang folgt unmittelbar aus der Definition von T_1 . Um den Induktions-Schritt durchzuführen, betrachten wir

$$\begin{aligned}
T_{n+1} &= R \cup R \circ T_n && \text{gilt nach Definition} \\
&= R \cup R \circ \left(\bigcup_{i=1}^n R^i \right) && \text{gilt nach Induktions-Voraussetzung} \\
&= R \cup R^2 \cup \dots \cup R^{n+1} && \text{Distributiv-Gesetz} \\
&= R^1 \cup \dots \cup R^{n+1} \\
&= \bigcup_{i=1}^{n+1} R^i && \square
\end{aligned}$$

Die Folge $(T_n)_{n \in \mathbb{N}}$ hat eine weitere nützliche Eigenschaft: Sie ist *monoton steigend*. Allgemein nennen wir eine Folge von Mengen $(X_n)_{n \in \mathbb{N}}$ *monoton steigend*, wenn

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

gilt, wenn also die Mengen X_n mit wachsendem Index n immer größer werden. Die Monotonie der Folge $(T_n)_{n \in \mathbb{N}}$ folgt aus der gerade bewiesenen Eigenschaft $T_n = \bigcup_{i=1}^n R^i$, denn es gilt

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

und die letzte Formel ist offenbar wahr. Ist nun die Relation R endlich, so ist natürlich auch R^+ eine endliche Menge. Da die Folge T_n aber in dieser Menge liegt, denn es gilt ja

$$T_n = \bigcup_{i=1}^n R^i \subseteq \bigcup_{i=1}^{\infty} R^i = R^+ \quad \text{für alle } n \in \mathbb{N},$$

können die Mengen T_n nicht beliebig groß werden. Aufgrund der Monotonie der Folge $(T_n)_{n \in \mathbb{N}}$ muss es daher einen Index k geben, ab dem die Mengen T_n alle gleich sind:

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

Berücksichtigen wir die Gleichung $T_n = \bigcup_{i=1}^n R^i$, so haben wir

$$T_n = \bigcup_{i=1}^n R^i = \bigcup_{i=1}^k R^i = T_k \quad \text{für alle } n \geq k.$$

Daraus folgt dann aber, dass

$$T_n = \bigcup_{i=1}^n R^i = \bigcup_{i=1}^{\infty} R^i = R^+ \quad \text{für alle } n \geq k$$

gilt. Der Algorithmus zur Berechnung von R^+ sieht nun so aus, dass wir die Iteration

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

solange durchführen bis $T_{n+1} = T_n$ gilt, denn dann gilt auch $T_n = R^+$.

```

1  closure := procedure(r) {
2      t := r;
3      while (true) {
4          oldT := t;
5          t := r + product(r, t);
6          if (t == oldT) {
7              return t;
8          }
9      }
10 };
11 product := procedure(r1, r2) {
12     // WARNING:
13     //   return { [x,z] : [x,y] in r1, [y,z] in r2 };
14     // DOES NOT WORK!
15     return { [x,z] : [x,y1] in r1, [y2,z] in r2 | y1 == y2 };
16 };
17 r := { [1,2], [2,3], [1,3], [2,4], [4,5] };
18 print( "r = ", r );
19 print( "computing transitive closure of r" );
20 t := closure(r);
21 print( "r+ = ", t );

```

Abbildung 2.26: Berechnung des transitiven Abschlusses.

Das Programm in Abbildung 2.26 auf Seite 29 zeigt eine Implementierung dieses Gedankens. Lassen wir dieses Programm laufen, so erhalten wir als Ausgabe:

```

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]}

```

Der transitive Abschluß R^+ der Relation R läßt sich jetzt anschaulich interpretieren: Er enthält alle Paare $\langle x, y \rangle$, für die es einen *Pfad* von x nach y gibt. Ein Pfad von x nach y ist dabei eine Liste der Form

$$[x_1, x_2, \dots, x_n],$$

für die $x = x_1$ und $y = x_n$ gilt und für die außerdem

$$\langle x_i, x_{i+1} \rangle \in R \quad \text{für alle } i = 1, \dots, n-1 \text{ gilt.}$$

Die Funktion $product(r_1, r_2)$ berechnet das relationale Produkt $r_1 \circ r_2$ nach der Formel

$$r_1 \circ r_2 = \{ \langle x, z \rangle \mid \exists y : \langle x, y \rangle \in r_1 \wedge \langle y, z \rangle \in r_2 \}.$$

Die Implementierung dieser Prozedur zeigt die allgemeine Form der Mengen-Definition durch Iteratoren in *SetIX*. Allgemein können wir eine Menge durch den Ausdruck

$$\{ \text{expr} : [x_1^{(1)}, \dots, x_{n(1)}^{(1)}] \text{ in } s_1, \dots, [x_1^{(k)}, \dots, x_{n(k)}^{(k)}] \text{ in } s_k \mid \text{cond} \}$$

definieren. Dabei muss s_i für alle $i = 1, \dots, k$ eine Menge von Listen der Länge $n(i)$ sein. Bei der Auswertung dieses Ausdrucks werden für die Variablen $x_1^{(i)}, \dots, x_{n(i)}^{(i)}$ die Werte eingesetzt, die

die entsprechenden Komponenten der Listen haben, die in der Menge s_i auftreten. Beispielsweise würde die Auswertung von

```
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 };
```

für M die Menge

```
{ [1, 2, 3, "a", "b"], [5, 6, 7, "c", "d"],
  [1, 2, 3, "c", "d"], [5, 6, 7, "a", "b"] }
```

berechnen.

Bei einer Mengen-Definition in der obigen Form gibt es eine **wichtige Einschränkung** zu beachten: Die Variablen $x_j^{(i)}$ müssen alle verschieden sein. Daher kann die Prozedur *product* auch nicht wie folgt programmiert werden:

```
1      product := procedure(r1, r2) {
2          return { [x,z] : [x,y] in r1, [y,z] in r2 };
3      };
```

Bei dieser Implementierung wird an zwei verschiedenen Stellen die selbe Variable y verwendet. Die Lösung des Problems besteht darin, statt einer Variablen y zwei verschiedene Variablen y1 und y2 zu verwenden. Um zu erzwingen, dass der Wert von y1 mit dem Wert von y2 übereinstimmt, wird dann die Bedingung “y1 == y2” hinzugenommen.

2.8.2 Berechnung der Pfade

Als nächstes wollen wir das Programm zur Berechnung des transitiven Abschlusses so erweitern, dass wir nicht nur feststellen können, dass es einen Pfad zwischen zwei Punkten gibt, sondern dass wir diesen auch berechnen können. Die Idee ist, dass wir statt des relationalen Produkts, das für zwei Relationen definiert ist, ein sogenanntes *Pfad-Produkt*, das auf Mengen von Pfaden definiert ist, berechnen. Vorab führen wir für Pfade, die wir ja durch Listen repräsentieren, drei Begriffe ein.

1. Die Funktion $first(p)$ liefert den ersten Punkt der Liste p :

$$first([x_1, \dots, x_m]) = x_1.$$

2. Die Funktion $last(p)$ liefert den letzten Punkt der Liste p :

$$last([x_1, \dots, x_m]) = x_m.$$

3. Sind $p = [x_1, \dots, x_m]$ und $q = [y_1, \dots, y_n]$ zwei Pfade mit $first(q) = last(p)$, dann definieren wir die Summe von p und q als

$$p \oplus q := [x_1, \dots, x_m, y_2, \dots, y_n].$$

Sind nun P_1 und P_2 Mengen von Pfaden, so definieren wir das *Pfad-Produkt* von P_1 und P_2 als

$$P_1 \bullet P_2 := \{ p_1 \oplus p_2 \mid p_1 \in P_1 \wedge p_2 \in P_2 \wedge last(p_1) = first(p_2) \}.$$

Damit können wir das Programm in Abbildung 2.26 so abändern, dass alle möglichen Verbindungen zwischen zwei Punkten berechnet werden. Abbildung 2.27 zeigt das resultierende Programm. Leider funktioniert das Programm dann nicht mehr, wenn der Graph Zyklen enthält. Abbildung 2.28 zeigt einen Graphen, der einen Zyklus enthält. In diesem Graphen gibt es unendlich viele Pfade, die von dem Punkt 1 zu dem Punkt 2 führen:

```

1  closure := procedure(r) {
2      p := r;
3      while (true) {
4          oldP := p;
5          p := r + pathProduct(r, p);
6          if (p == oldP) {
7              return p;
8          }
9      }
10 };
11 pathProduct := procedure(p, q) {
12     return { add(x, y) : x in p, y in q | x(#x) == y(1) };
13 };
14 add := procedure(p, q) {
15     return p + q(2..);
16 };
17 r := { [1,2], [2,3], [1,3], [2,4], [4,5] };
18 print( "r = ", r );
19 print( "computing all pathes" );
20 p := closure(r);
21 print( "p = ", p );

```

Abbildung 2.27: Berechnung aller Verbindungen.

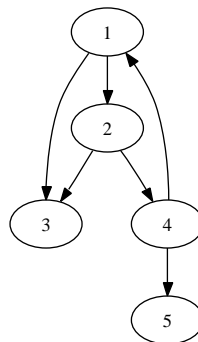


Abbildung 2.28: Ein zyklischer Graph.

$[1, 2], [1, 2, 4, 1, 2], [1, 2, 4, 1, 2, 4, 1, 2], [1, 2, 4, 1, 2, 4, 1, 2, 4, 1, 4], \dots$

Offenbar sind die Pfade unwichtig, die einen Punkt mehrfach enthalten und die daher zyklisch sind. Solche Pfade sollten wir bei der Berechnung des Pfad-Produktes eliminieren.

```

1  pathProduct := procedure(p, q) {
2      return { add(x,y) : x in p, y in q | x(#x) == y(1) && !cyclic(add(x,y)) };
3  };
4  cyclic := procedure(p) {
5      return #{ x : x in p } < #p;
6  };

```

Abbildung 2.29: Berechnung aller Verbindungen in zyklischen Graphen

Abbildung 2.29 zeigt, wie wir das Programm ändern müssen, damit es auch für zyklische Graphen funktioniert.

1. In Zeile 2 berücksichtigen wir nur die Pfade $p \oplus q$, die nicht zyklisch sind.
2. In Zeile 5 überprüfen wir, ob ein Pfad zyklisch ist. Ein Pfad ist genau dann zyklisch wenn er einen Punkt mehrfach enthält. Wenn wir also den Pfad in eine Menge umwandeln und der Pfad ist zyklisch, so enthält die Menge hinterher weniger Elemente als der Pfad, denn in einer Menge kommt ja jedes Element nur einmal vor.

In den meisten Fällen wird gar nicht daran interessiert, alle möglichen Verbindungen zwischen allen Punkten zu berechnen, das wäre nämlich viel zu aufwendig, sondern wir wollen nur zwischen zwei gegebenen Punkten eine Verbindung finden. Abbildung 2.30 zeigt die Implementierung einer Prozedur `reachable(x, y, r)`, die überprüft, ob es in dem Graphen r eine Verbindung von x nach y gibt und die diese Verbindung berechnet.

1. In Zeile 2 initialisieren wir p so, dass zunächst nur der Pfad der Länge 0, der mit dem Punkt x startet, in p liegt.
2. In Zeile 6 selektieren wir die Pfade aus p , die zum Ziel y führen.
3. Wenn wir dann in Zeile 7 feststellen, dass wir einen solchen Pfad berechnet haben, geben wir einen dieser Pfade in Zeile 8 zurück.
4. Falls es nicht gelingt einen solchen Pfad zu berechnen und wir keine neuen Pfade mehr finden können, verlassen wir die Prozedur in Zeile 12 mit dem Befehl `return`. Da wir bei diesem `return`-Befehl keinen Wert zurückgeben, ist der Rückgabewert der Prozedur in diesem Fall automatisch Ω .

```

1  reachable := procedure(x, y, r) {
2      p := { [x] };
3      while (true) {
4          oldP := p;
5          p := p + pathProduct(p, r);
6          found := { l in p | l(#l) == y };
7          if (found != {}) {
8              return arb(found);
9          }
10         if (p == oldP) {
11             return;
12         }
13     }
14 };

```

Abbildung 2.30: Berechnung aller Verbindungen zwischen zwei Punkten

2.8.3 Der Bauer mit dem Wolf, der Ziege und dem Kohl

Wir präsentieren nun eine betriebswissenschaftliche Anwendung des oben entwickelten Algorithmus und betrachten folgendes Problem.

Ein Bauer will mit einem Wolf, einer Ziege und einem Kohl über einen Fluß übersetzen, um diese als Waren auf dem Markt zu verkaufen. Das Boot ist aber so klein, dass er nicht mehr als zwei Waren gleichzeitig mitnehmen kann. Wenn der Bauer den Wolf mit der Ziege allein läßt, dann frißt der Wolf die Ziege und wenn er die Ziege mit dem Kohl allein läßt, dann frißt die Ziege den Kohl.

Wir wollen einen Fahrplan entwickeln, mit dem der Bauer alle seine Waren unbeschadet zum Markt bringen kann. Dazu modellieren wir das Rätsel als Erreichbarkeits-Problem in einem Graphen. Die Punkte des Graphen beschreiben dabei die einzelnen Situationen, die auftreten können. Wir definieren eine Menge

$$\mathbf{all} := \{\text{"Bauer"}, \text{"Wolf"}, \text{"Ziege"}, \text{"Kohl"}\}.$$

Die einzelnen Punkte sind dann Paare von Mengen, haben also die Form

$$\langle s_1, s_2 \rangle \quad \text{mit } s_1, s_2 \subseteq \mathbf{all}.$$

Dabei gibt die Menge s_1 an, was am linken Ufer ist und s_2 gibt an, was am rechten Ufer ist. Die Menge aller Punkte können wir dann definieren als

$$p := \{ \langle s_1, s_2 \rangle \in 2^{\mathbf{all}} \times 2^{\mathbf{all}} \mid s_1 \cup s_2 = \mathbf{all} \wedge s_1 \cap s_2 = \{\} \}.$$

Die Bedingung $s_1 \cup s_2 = \mathbf{all}$ stellt dabei sicher, dass nichts verloren geht: Jedes der Elemente aus \mathbf{all} ist entweder am linken oder am rechten Ufer. Die Bedingung $s_1 \cap s_2 = \{\}$ verbietet die Bilokalisation von Objekten, sie stellt also sicher, dass kein Element aus der Menge \mathbf{all} gleichzeitig am linken und am rechten Ufer ist.

Als nächstes definieren wir den Graphen r , also die möglichen Verbindungen zwischen Punkten. Dazu definieren wir eine Prozedur $problem(s)$. Hierbei ist s eine Menge von Objekten, die an einem Ufer sind. Die Prozedur $problem(s)$ liefert genau dann **true**, wenn es bei der Menge s ein Problem gibt, weil entweder die Ziege mit dem Kohl allein ist, oder aber der Wolf die Ziege frißt.

```
problem := procedure(s) {
  return "goat" in s && "cabbage" in s || "wolf" in s && "goat" in s;
};
```

Damit können wir eine Relation r_1 wie folgt definieren:

$$r_1 := \left\{ \langle \langle s_1, s_2 \rangle, \langle s_1 \setminus b, s_2 \cup b \rangle \rangle \mid \right. \\ \left. \langle s_1, s_2 \rangle \in P \wedge b \subseteq s_1 \wedge \text{"Bauer"} \in b \wedge \text{card}(b) \leq 2 \wedge \neg problem(s_1 \setminus b) \right\}.$$

Diese Menge beschreibt alle die Fahrten, bei denen der Bauer vom linken Ufer zum rechten Ufer fährt und bei denen zusätzlich sichergestellt ist, dass am linken Ufer nach der Überfahrt kein Problem auftritt. Die einzelnen Terme werden wie folgt interpretiert:

1. $\langle s_1, s_2 \rangle$ ist der Zustand vor der Überfahrt des Bootes, s_1 gibt also die Objekte am linken Ufer an, s_2 sind die Objekte am rechten Ufer.
2. b ist der Inhalt des Bootes, daher beschreibt $\langle s_1 \setminus b, s_2 \cup b \rangle$ den Zustand nach der Überfahrt des Bootes: Links sind nun nur noch die Objekte aus $s_1 \setminus b$, dafür sind rechts dann die Objekte $s_2 \cup b$.

Die Bedingungen lassen sich wie folgt interpretieren.

3. $b \subseteq s_1$: Es können natürlich nur solche Objekte ins Boot genommen werden, die vorher am linken Ufer waren.

4. "Bauer" $\in b$: Der Bauer muß auf jeden Fall ins Boot, denn weder der Wolf noch die Ziege können rudern.
5. $\text{card}(b) \leq 2$: Die Menge der Objekte im Boot darf nicht mehr als zwei Elemente haben, denn im Boot ist nur für zwei Platz.
6. $\neg \text{problem}(s_1 \setminus b)$: Am linken Ufer soll es nach der Überfahrt kein Problem geben, denn der Bauer ist ja hinterher am rechten Ufer.

In analoger Weise definieren wir nun eine Relation r_2 die die Überfahrten vom rechten Ufer zum linken Ufer beschreibt:

$$r_2 := \left\{ \langle \langle s_1, s_2 \rangle, \langle s_1 \cup b, s_2 \setminus b \rangle \rangle \mid \langle s_1, s_2 \rangle \in P \wedge b \subseteq s_2 \wedge \text{"Bauer"} \in b \wedge \text{card}(b) \leq 2 \wedge \neg \text{problem}(s_2 \setminus b) \right\}.$$

Die gesamte Relation r definieren wir nun als

$$r := r_1 \cup r_2.$$

Als nächstes müssen wir den Start-Zustand modellieren. Am Anfang sind alle am linken Ufer, also wird der Start-Zustand durch das Paar

$$\langle \{ \text{"Bauer"}, \text{"Wolf"}, \text{"Ziege"}, \text{"Kohl"} \}, \{ \} \rangle.$$

Beim Ziel ist es genau umgekehrt, dann sollen alle auf der rechten Seite des Ufers sein:

$$\langle \{ \}, \{ \text{"Bauer"}, \text{"Wolf"}, \text{"Ziege"}, \text{"Kohl"} \} \rangle.$$

Damit haben wir das Problem in der Mengenlehre modelliert und können die im letzten Abschnitt entwickelte Prozedur **reachable** benutzen, um das Problem zu lösen. Abbildung 2.31 zeigt das Programm. Die von diesem Programm berechnete Lösung finden Sie in Abbildung 2.32.

```

1  all := { "farmer", "wolf", "goat", "cabbage" };
2  p   := pow(all);
3  r1  := { [ s, s - b ] : s in p, b in pow(s)
4          | "farmer" in b && #b <= 2 && !problem(s - b)
5          };
6  r2  := { [ s, s + b ] : s in p, b in pow(all - s)
7          | "farmer" in b && #b <= 2 && !problem(all - (s + b))
8          };
9  r    := r1 + r2;

```

Abbildung 2.31: Wie kommt der Bauer ans andere Ufer?

2.8.4 Ausblick

Wir können aus Zeitgründen nur einen Teil der Funktionalität von *SetIX* diskutieren. Hinzu kommt, dass die Sprache *SetIX* sich noch im Entwicklungs-Stadium befindet: Bisher ist nur ein Teil des definierten Sprachumfangs implementiert.

Noch eine Bemerkung zu den in diesem Kapitel vorgestellten Algorithmen: Sie sollten sich keineswegs der Illusion hingeben zu glauben, dass diese Algorithmen effizient sind. Sie dienen nur dazu, die Begriffsbildungen aus der Mengenlehre konkret werden zu lassen. Die Entwicklung effizienter Algorithmen ist Gegenstand des zweiten Semesters.

1	{ "Kohl", "Ziege", "Wolf", "Bauer" }	{ }
2	>> { "Ziege", "Bauer" } >>	
3	{ "Kohl", "Wolf" }	{ "Ziege", "Bauer" }
4	<< { "Bauer" } <<<<	
5	{ "Kohl", "Wolf", "Bauer" }	{ "Ziege" }
6	>> { "Wolf", "Bauer" } >>>	
7	{ "Kohl" }	{ "Ziege", "Wolf", "Bauer" }
8	<< { "Ziege", "Bauer" } <<	
9	{ "Kohl", "Ziege", "Bauer" }	{ "Wolf" }
10	>> { "Kohl", "Bauer" } >>>	
11	{ "Ziege" }	{ "Kohl", "Wolf", "Bauer" }
12	<< { "Bauer" } <<<<	
13	{ "Ziege", "Bauer" }	{ "Kohl", "Wolf" }
14	>> { "Ziege", "Bauer" } >>	
15	{ }	{ "Kohl", "Ziege", "Wolf", "Bauer" }

Abbildung 2.32: Ein Fahrplan für den Bauern

Kapitel 3

Aussagenlogik

3.1 Motivation

Die Aussagenlogik beschäftigt sich mit der Verknüpfung einfacher Aussagen durch *Junktoren*. Dabei sind Junktoren Worte wie “und”, “oder”, “nicht”, “wenn \dots , dann”, und “genau dann, wenn”. Einfache Aussagen sind dabei Sätze, die

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

Beispiele für einfache Aussagen sind

1. “Die Sonne scheint.”
2. “Es regnet.”
3. “Am Himmel ist ein Regenbogen.”

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

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

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

Um die Struktur komplexerer Aussagen übersichtlich werden zu lassen, führen wir in der Aussagenlogik zunächst sogenannte *Aussage-Variablen* ein. Diese stehen für atomare Aussagen. Zusätzlich führen wir für die Junktoren “nicht”, “und”, “oder”, “wenn, \dots dann”, und “genau dann, wenn” die folgenden Abkürzungen ein:

1. $\neg a$ für *nicht a*
2. $a \wedge b$ für *a und b*
3. $a \vee b$ für *a oder b*

4. $a \rightarrow b$ für wenn a , dann b
5. $a \leftrightarrow b$ für a genau dann, wenn b

Aussagenlogische Formeln werden aus Aussage-Variablen mit Hilfe von Junktoren aufgebaut. Bestimmte aussagenlogische Formeln sind offenbar immer wahr, egal welche Wahrheitswerte wir für die einzelnen Teilaussagen einsetzen, beispielsweise ist die Formel

$$p \vee \neg p$$

immer wahr. Solche Aussagen bezeichnen wir als *Tautologien*. Andere aussagenlogische Formeln sind nie wahr, beispielsweise ist die Formel

$$p \wedge \neg p$$

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

3.2 Anwendungen der Aussagenlogik

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

1. Analyse und Design digitaler Schaltungen.

Komplexe digitale Schaltungen bestehen heute aus mehreren Millionen logischen Gattern¹. Ein Gatter ist dabei, aus logischer Sicht betrachtet, ein Baustein, der einen der logischen Junktoren wie „und“, „oder“, „nicht“, etc. auf elektronischer Ebene repräsentiert.

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

Eine ganz konkrete Anwendung ist der Schaltungs-Vergleich. Hier werden zwei digitale Schaltungen als aussagenlogische Formeln dargestellt. Anschließend wird versucht, mit aussagenlogischen Mitteln die Äquivalenz dieser Formeln zu zeigen. Software-Werkzeuge, die für die Verifikation digitaler Schaltungen eingesetzt werden, kosten heutzutage über 100 000 \$². Dies zeigt die wirtschaftliche Bedeutung der Aussagenlogik.

2. Erstellung von Einsatzplänen (*crew scheduling*).

International tätige Fluggesellschaften müssen bei der Einteilung Ihrer Crews einerseits gesetzlich vorgesehene Ruhezeiten einhalten, wollen aber ihr Personal möglichst effizient einsetzen. Das führt zu Problemen, die sich mit Hilfe aussagenlogischer Formeln beschreiben und lösen lassen.

3. Erstellung von Verschußplänen für die Weichen und Signale von Bahnhöfen.

Bei einem größeren Bahnhof gibt es einige hundert Weichen und Signale, die ständig neu eingestellt werden müssen, um sogenannte *Fahrstraßen* für die Züge zu realisieren. Verschiedene Fahrstraßen sollen sich natürlich nicht kreuzen. Die einzelnen Fahrstraßen werden durch sogenannte *Verschußpläne* beschrieben. Die Korrektheit solcher Verschußpläne kann durch aussagenlogische Formeln ausgedrückt werden.

¹Die Version des Pentium 4 Prozessors mit dem Northwood Kernel enthält etwa 55 Millionen Transistoren.

²Die Firma Magma bietet beispielsweise den *Equivalence-Checker Quartz Formal* zum Preis von 150 000 \$ pro Lizenz an. Eine solche Lizenz ist dann drei Jahre lang gültig.

4. Eine Reihe kombinatorischer Puzzles lassen sich als aussagenlogische Formeln kodieren und können dann mit Hilfe aussagenlogischer Methoden lösen. Als ein Beispiel werden wir in der Vorlesung das 8-Damen-Problem behandeln. Dabei geht es um die Frage, ob 8 Damen so auf einem Schachbrett angeordnet werden können, dass keine der Damen eine andere Dame bedroht.

3.3 Formale Definition der aussagenlogischen Formeln

Wir behandeln zunächst die *Syntax* der Aussagenlogik und besprechen anschließend die *Semantik*. Die *Syntax* gibt an, wie Formeln geschrieben werden. Die *Semantik* befasst sich mit der Bedeutung der Formeln. Nachdem wir die Semantik der aussagenlogischen Formeln definiert haben, zeigen wir, wie sich diese Semantik in SETL implementieren läßt.

3.3.1 Syntax der aussagenlogischen Formeln

Wir betrachten eine Menge \mathcal{P} von *Aussage-Variablen* als gegeben. Aussagenlogische Formeln sind dann Wörter, die aus dem Alphabet

$$\mathcal{A} := \mathcal{P} \cup \{\top, \perp, \neg, \vee, \wedge, \rightarrow, \leftrightarrow, (,)\}$$

gebildet werden. Wir definieren die Menge der aussagenlogischen Formeln \mathcal{F} durch eine induktive Definition:

1. $\top \in \mathcal{F}$ und $\perp \in \mathcal{F}$.

Hier steht \top für die Formel, die immer wahr ist, während \perp für die Formel steht, die immer falsch ist. Die Formel \top trägt auch den Namen *Verum*, für \perp sagen wir auch *Falsum*.

2. Ist $p \in \mathcal{P}$, so gilt auch $p \in \mathcal{F}$.

Jede aussagenlogische Variable ist also eine aussagenlogische Formel.

3. Ist $f \in \mathcal{F}$, so gilt auch $\neg f \in \mathcal{F}$.

4. Sind $f_1, f_2 \in \mathcal{F}$, so gilt auch

- (a) $(f_1 \vee f_2) \in \mathcal{F}$ (gelesen: f_1 oder f_2),
- (b) $(f_1 \wedge f_2) \in \mathcal{F}$ (gelesen: f_1 und f_2),
- (c) $(f_1 \rightarrow f_2) \in \mathcal{F}$ (gelesen: wenn f_1 , dann f_2),
- (d) $(f_1 \leftrightarrow f_2) \in \mathcal{F}$ (gelesen: f_1 genau dann, wenn f_2).

Die Menge \mathcal{F} ist nun die kleinste Teilmenge der aus dem Alphabet \mathcal{A} gebildeten Wörter, die den oben aufgestellten Forderungen genügt.

Beispiel: Gilt $\mathcal{P} = \{p, q, r\}$, so haben wir beispielsweise:

1. $p \in \mathcal{F}$,
2. $(p \wedge q) \in \mathcal{F}$,
3. $((\neg p \rightarrow q) \vee (q \rightarrow \neg p)) \in \mathcal{F}$.

□

Um Klammern zu sparen, vereinbaren wir:

1. Äußere Klammern werden weggelassen, wir schreiben also beispielsweise

$$p \wedge q \quad \text{statt} \quad (p \wedge q).$$

2. Die Junktoren \vee und \wedge werden implizit links geklammert, d.h. wir schreiben

$$p \wedge q \wedge r \quad \text{statt} \quad (p \wedge q) \wedge r.$$

Operatoren, die implizit nach links geklammert werden, nennen wir *links-assoziativ*.

3. Der Junktor \rightarrow wird implizit rechts geklammert, d.h. wir schreiben

$$p \rightarrow q \rightarrow r \quad \text{statt} \quad p \rightarrow (q \rightarrow r).$$

Operatoren, die implizit nach rechts geklammert werden, nennen wir *rechts-assoziativ*.

4. Die Junktoren \vee und \wedge binden stärker als \rightarrow , wir schreiben also

$$p \wedge q \rightarrow r \quad \text{statt} \quad (p \wedge q) \rightarrow r$$

5. Der Junktor \rightarrow bindet stärker als \leftrightarrow , wir schreiben also

$$p \rightarrow q \leftrightarrow r \quad \text{statt} \quad (p \rightarrow q) \leftrightarrow r.$$

3.3.2 Semantik der aussagenlogischen Formeln

Um aussagenlogischen Formeln einen Wahrheitswert zuordnen zu können, definieren wir zunächst die Menge \mathbb{B} der Wahrheitswerte:

$$\mathbb{B} := \{\text{true}, \text{false}\}.$$

Damit können wir nun den Begriff einer *aussagenlogischen Interpretation* festlegen.

Definition 1 (Aussagenlogische Interpretation) Eine *aussagenlogische Interpretation* ist eine Funktion

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

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

Eine aussagenlogische Interpretation wird oft auch als *Belegung* der Aussage-Variablen mit Wahrheits-Werten bezeichnet.

Eine aussagenlogische Interpretation \mathcal{I} interpretiert die Aussage-Variablen. Um nicht nur Variablen sondern auch aussagenlogische Formel interpretieren zu können, benötigen wir eine Interpretation der Junktoren “ \neg ”, “ \wedge ”, “ \vee ”, “ \rightarrow ” und “ \leftrightarrow ”. Zu diesem Zweck definieren wir auf der Menge \mathbb{B} Funktionen \neg , \wedge , \vee , \rightarrow und \leftrightarrow mit deren Hilfe wir die aussagenlogischen Junktoren interpretieren können:

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. $\leftrightarrow : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

Wir haben in der Mengenlehre gesehen, dass Funktionen als spezielle Relationen aufgefaßt werden können. Die Funktion \neg dreht die Wahrheits-Werte um und kann daher wie folgt geschrieben werden:

$$\neg = \{ \langle \text{true}, \text{false} \rangle, \langle \text{false}, \text{true} \rangle \}.$$

Wir werden diese Funktionen auch tatsächlich so in *SetIX* darstellen, aber für die Tafel ist dieses Schreibweise zu umständlich. Dort ist es einfacher, die Funktionen durch eine Tabelle zu definieren. Diese Tabelle ist auf Seite 40 abgebildet.

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

Tabelle 3.1: Interpretation der Junktoren.

Nun können wir den Wert, den eine aussagenlogische Formel f unter einer aussagenlogischen Interpretation \mathcal{I} annimmt, durch Induktion nach dem Aufbau der Formel f definieren. Wir werden diesen Wert mit $\widehat{\mathcal{I}}(f)$ bezeichnen. Wir setzen:

1. $\widehat{\mathcal{I}}(\perp) := \text{false}$.
2. $\widehat{\mathcal{I}}(\top) := \text{true}$.
3. $\widehat{\mathcal{I}}(p) := \mathcal{I}(p)$ für alle $p \in \mathcal{P}$.
4. $\widehat{\mathcal{I}}(\neg f) := \neg(\widehat{\mathcal{I}}(f))$ für alle $f \in \mathcal{F}$.
5. $\widehat{\mathcal{I}}(f \wedge g) := \wedge(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \vee(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \rightarrow g) := \oplus(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \ominus(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.

Um die Schreibweise nicht übermäßig kompliziert werden zu lassen, unterscheiden wir in Zukunft nicht mehr zwischen $\widehat{\mathcal{I}}$ und \mathcal{I} , wir werden das Hütchen über dem \mathcal{I} also weglassen.

Beispiel: Wir zeigen, wie sich der Wahrheits-Wert der Formel

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

für die aussagenlogische Interpretation \mathcal{I} , die durch $\mathcal{I}(p) = \text{true}$ und $\mathcal{I}(q) = \text{false}$ definiert ist, berechnen läßt:

$$\begin{aligned}
\mathcal{I}\big((p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q\big) &= \oplus\big(\mathcal{I}\big((p \rightarrow q) \rightarrow (\neg p \rightarrow q)\big), \mathcal{I}(q)\big) \\
&= \oplus\big(\oplus(\mathcal{I}(p), \mathcal{I}(q)), \mathcal{I}(\neg p \rightarrow q)\big) \\
&= \oplus\big(\oplus(\text{true}, \text{false}), \mathcal{I}(\neg p \rightarrow q)\big) \\
&= \oplus\big(\text{false}, \mathcal{I}(\neg p \rightarrow q)\big) \\
&= \text{true}
\end{aligned}$$

Beachten Sie, dass wir bei der Berechnung gerade so viele Teile der Formel ausgewertet haben, wie notwendig waren um den Wert der Formel zu bestimmen. Trotzdem ist die eben durchgeführte Rechnung für die Praxis zu umständlich. Stattdessen wird der Wert einer Formel direkt mit Hilfe der Tabelle 3.1 auf Seite 40 berechnet. Wir zeigen exemplarisch, wie wir den Wahrheits-Wert der Formel

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

für beliebige Belegungen \mathcal{I} über diese Tabelle berechnen können. Um nun die Wahrheitswerte dieser Formel unter einer gegebenen Belegung der Aussage-Variablen bestimmen zu können, bauen wir

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

Tabelle 3.2: Berechnung Der Wahrheitswerte von $(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$.

eine Tabelle auf, die für jede in der Formel auftretende Teilformel eine Spalte enthält. Tabelle 3.2 auf Seite 41 zeigt die entstehende Tabelle.

Betrachten wir die letzte Spalte der Tabelle so sehen wir, dass dort immer der Wert **true** auftritt. Also liefert die Auswertung der Formel $(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$ für jede aussagenlogische Belegung \mathcal{I} den Wert **true**. Formeln, die immer wahr sind, haben in der Aussagenlogik eine besondere Bedeutung und werden als *Tautologien* bezeichnet.

Wir erläutern die Aufstellung dieser Tabelle anhand der zweiten Zeile. In dieser Zeile sind zunächst die aussagenlogischen Variablen p auf **true** und q auf **false** gesetzt. Bezeichnen wir die aussagenlogische Interpretation mit \mathcal{I} , so gilt also

$$\mathcal{I}(p) = \text{true} \text{ und } \mathcal{I}(q) = \text{false}.$$

Damit erhalten wir folgende Rechnung:

1. $\mathcal{I}(\neg p) = \ominus(\mathcal{I}(p)) = \ominus(\text{true}) = \text{false}$
2. $\mathcal{I}(p \rightarrow q) = \ominus(\mathcal{I}(p), \mathcal{I}(q)) = \ominus(\text{true}, \text{false}) = \text{false}$
3. $\mathcal{I}(\neg p \rightarrow q) = \ominus(\mathcal{I}(\neg p), \mathcal{I}(q)) = \ominus(\text{false}, \text{false}) = \text{true}$
4. $\mathcal{I}((\neg p \rightarrow q) \rightarrow q) = \ominus(\mathcal{I}(\neg p \rightarrow q), \mathcal{I}(q)) = \ominus(\text{true}, \text{false}) = \text{false}$
5. $\mathcal{I}((p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q) = \ominus(\mathcal{I}(p \rightarrow q), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)) = \ominus(\text{false}, \text{false}) = \text{true}$

Für komplexe Formeln ist die Auswertung von Hand viel zu mühsam und fehleranfällig um praktikabel zu sein. Wir zeigen deshalb im nächsten Abschnitt, wie sich dieser Prozeß automatisieren läßt.

3.3.3 Extensionale und intensionale Interpretationen der Aussagenlogik

Die Interpretation des aussagenlogischen Junktoren ist rein *extensional*: Wenn wir den Wahrheitswert der Formel

$$\mathcal{I}(f \rightarrow g)$$

berechnen wollen, so müssen wir die Details der Teilformeln f und g nicht kennen, es reicht, wenn wir die Werte $\mathcal{I}(f)$ und $\mathcal{I}(g)$ kennen. Das ist problematisch, den in der Umgangssprache hat der Junktor “*wenn ... , dann*” auch eine *kausale* Bedeutung. Mit der extensionalen Implikation wird der Satz

“Wenn $3 \cdot 3 = 8$, dann schneit es.”

als wahr interpretiert. Dass ist problematisch, weil wir diesen Satz in der Umgangssprache als sinnlos erkennen. Insofern ist die extensionale Interpretation des sprachlichen Junktors “*wenn ... , dann*” nur eine Approximation der umgangssprachlichen Interpretation, die sich für Mathematik und Informatik aber als ausreichend erwiesen hat.

3.3.4 Implementierung in *SetIX*

Um die bisher eingeführten Begriffe nicht zu abstrakt werden zu lassen, entwickeln wir in *SetIX* ein Programm, mit dessen Hilfe sich Formeln auswerten lassen. Jedesmal, wenn wir ein Programm zur Berechnung irgendwelcher Wert entwickeln wollen, müssen wir uns als erstes fragen, wie wir die Argumente der zu implementierenden Funktion und die Ergebnisse dieser Funktion in der verwendeten Programmier-Sprache darstellen können. In diesem Fall müssen wir uns also überlegen, wie wir eine aussagenlogische Formel in *SetIX* repräsentieren können, denn Ergebnisswerte **true** und **false** stehen ja als Wahrheitswerte unmittelbar zur Verfügung. Zusammengesetzte Datenstrukturen können in *SetIX* am einfachsten als Listen dargestellt werden und das ist auch der Weg, den wir für die aussagenlogischen Formeln, die aus anderen Formeln zusammengesetzt sind, beschreiten werden. Wir definieren die Repräsentation von aussagenlogischen Formeln formal dadurch, dass wir eine Funktion

$$rep : \mathcal{F} \rightarrow SetIX$$

definieren, die einer aussagenlogischen Formel f eine SETL2-Datenstruktur $rep(f)$ zuordnet.

1. \top wird repräsentiert durch die Zahl 1

$$rep(\top) := 1$$

2. \perp wird repräsentiert durch die Zahl 0.

$$rep(\perp) := 0$$

3. Eine aussagenlogische Variable $p \in \mathcal{P}$ repräsentieren wir durch einen String, der den Namen der Variablen angibt. Falls die Aussage-Variablen von vorne herein Strings sind, dann gilt also

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

4. Ist f eine aussagenlogische Formel, so repräsentieren wir $\neg f$ als zwei-elementige Liste, die als erstes Element den String “-” enthält:

$$rep(\neg f) := ["-", rep(f)].$$

5. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \vee f_2$ als drei-elementige Liste, die als zweites Element den String “+” enthält:

$$rep(f \vee g) := [rep(f), "+", rep(g)].$$

6. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \wedge f_2$ als drei-elementige Liste, die als zweites Element den String “*” enthält:

$$rep(f \wedge g) := [rep(f), "*", rep(g)].$$

7. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \rightarrow f_2$ als drei-elementige Liste, die als zweites Element den String “->” enthält:

$$rep(f \rightarrow g) := [rep(f), "->", rep(g)].$$

8. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \leftrightarrow f_2$ als drei-elementige Liste, die als zweites Element den String “<->” enthält:

$$rep(f \leftrightarrow g) := [rep(f), "<->", rep(g)].$$

Bei der Wahl der Repräsentation, mit der wir eine Formel in SETL2 repräsentieren, sind wir weitgehend frei. Wir hätten oben sicher auch eine andere Repräsentation verwenden können. Beispielsweise stand in einer früheren Version dieses Skriptes der Junktor immer an der ersten Stelle der Liste. Eine gute Repräsentation sollte einerseits möglichst intuitiv sein, andererseits ist es auch wichtig, dass die Repräsentation für die zu entwickelnden Algorithmen adäquat ist. Im wesentlichen heißt dies, dass es möglichst einfach sein sollte, auf die Komponenten einer Formel zuzugreifen.

Als nächstes geben wir an, wie wir eine aussagenlogische Interpretation in *SetIX* darstellen. Eine aussagenlogische Interpretation ist eine Funktion

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

von der Menge der Aussage-Variablen \mathcal{P} in die Menge der Wahrheitswerte \mathbb{B} . Ist eine Formel f gegeben, so ist klar, dass bei der Interpretation \mathcal{I} nur die Aussage-Variablen p eine Rolle spielen, die auch in der Formel f auftreten. Wir können daher die Interpretation \mathcal{I} durch eine funktionale Relation darstellen, also durch eine Menge von Paaren $[p, b]$, für die p eine Aussage-Variable ist und $b \in \mathbb{B}$:

$$\mathcal{I} \subseteq \mathcal{P} \times \mathbb{B}.$$

Damit können wir jetzt eine einfache Prozedur schreiben, dass den Wahrheitswert einer aussagenlogischen Formel f unter einer gegebenen aussagenlogischen Interpretation \mathcal{I} berechnet. Ein solche Prozedur ist in Figur 3.1 auf Seite 43 gezeigt.

```

1  eval := procedure(f, i) {
2      switch {
3          case f == 1:      return true;
4          case f == 0:      return false;
5          case isString(f): return i(f);
6          case f(1) == "-": return !eval(f(2), i);
7          case f(2) == "*": return eval(f(1), i) && eval(f(3), i);
8          case f(2) == "+": return eval(f(1), i) || eval(f(3), i);
9          case f(2) == "->": return !eval(f(1), i) || eval(f(3), i);
10         case f(2) == "<->": return eval(f(1), i) == eval(f(3), i);
11         default: print("eval($f$): syntax error ");
12     }
13 };

```

Abbildung 3.1: Auswertung einer aussagenlogischen Formel.

Wir diskutieren jetzt die Implementierung der Funktion `eval()` Zeile für Zeile:

1. Falls das Argument f den Wert 1 hat, so repräsentiert f die Formel \top . Also ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation i immer **true**.
2. Falls das Argument f den Wert 0 hat, so repräsentiert f die Formel \perp . Also ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation i immer **false**.
3. In Zeile 5 betrachten wir den Fall, dass das Argument f eine aussagenlogische Variable repräsentiert. Dies erkennen wir mit Hilfe der Bibliotheks-Funktion `isString()`, die genau dann **true** zurück gibt, wenn ihr Argument ein String ist. In diesem Fall müssen wir die Belegung i , die ja eine Funktion von den aussagenlogischen Variablen in die Wahrheitswerte ist, auf die Variable f anwenden.
4. In Zeile 6 betrachten wir den Fall, dass f die Form $["-", g]$ hat und folglich die Formel $\neg g$ repräsentiert. In diesem Fall werten wir erst g unter der Belegung i aus und negieren dann das Ergebnis.
5. In Zeile 7 betrachten wir den Fall, dass f die Form $[g_1, "*", g_2]$ hat und folglich die Formel $g_1 \wedge g_2$ repräsentiert. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung i aus und verknüpfen das Ergebnis mit dem Operator "**&&**".

6. In Zeile 8 betrachten wir den Fall, dass f die Form $[g_1, "+", g_2]$ hat und folglich die Formel $g_1 \vee g_2$ repräsentiert. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung i aus und verknüpfen das Ergebnis mit dem Operator " $||$ ".
7. In Zeile 9 betrachten wir den Fall, dass f die Form $[g_1, "->", g_2]$ hat und folglich die Formel $g_1 \rightarrow g_2$ repräsentiert. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung i aus und benutzen die folgende aussagenlogische Äquivalenz:
$$(p \rightarrow q) \leftrightarrow \neg p \vee q.$$
8. In Zeile 10 führen wir die Auswertung einer Formel $g_1 \leftrightarrow g_2$ zurück auf die Gleichheit.
9. Wenn keiner der vorhergehenden Fälle greift, liegt ein Syntax-Fehler vor, auf den wir in Zeile 11 hinweisen.

3.3.5 Eine Anwendung

Wir betrachten eine spielerische Anwendung der Aussagenlogik. Inspektor Watson wird zu einem Juweliergeschäft gerufen, in das eingebrochen worden ist. In der unmittelbaren Umgebung werden drei Verdächtige Anton, Bruno und Claus festgenommen. Die Auswertung der Akten ergibt folgendes:

1. Einer der drei Verdächtigen muß die Tat begangen haben:

$$f_1 := a \vee b \vee c.$$

2. Wenn Anton schuldig ist, so hat er genau einen Komplizen.

Diese Aussage zerlegen wir zunächst in zwei Teilaussagen:

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

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

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

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

3. Wenn Bruno unschuldig ist, dann ist auch Claus unschuldig:

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

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

Es ist nicht leicht zu sehen, wie diese Aussage sich aussagenlogisch formulieren läßt. Wir behelfen uns mit einem Trick und überlegen uns, wann die obige Aussage falsch ist. Wir sehen, die Aussage ist dann falsch, wenn Claus nicht schuldig ist und wenn gleichzeitig Anton und Bruno schuldig sind. Damit lautet die Formalisierung der obigen Aussage:

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

5. Wenn Claus unschuldig ist, ist Anton schuldig.

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

Wir haben nun eine Menge $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$ von Formeln. Wir fragen uns nun, für welche Belegungen \mathcal{I} alle Formeln aus F wahr werden. Wenn es genau eine Belegungen gibt, für die dies der Fall ist, dann liefert uns die Belegung den oder die Täter. Eine Belegung entspricht dabei 1-zu-1 der Menge der Täter. Hätten wir beispielsweise

$$\mathcal{I} = \{\langle a, \text{false} \rangle, \langle b, \text{false} \rangle, \langle c, \text{true} \rangle\}.$$

In diesem Fall wäre Claus der alleinige Täter. Diese Belegung löst unser Problem offenbar nicht,

denn Sie widerspricht der dritten Aussage: Da Bruno unschuldig wäre, wäre dann auch Claus unschuldig. Da es zu zeitraubend ist, alle Belegungen von Hand auszuprobieren, schreiben wir besser ein Programm, das für uns die notwendige Berechnung durchführt. Abbildung 3.2 zeigt ein solches Programm.

```

1  // This procedure turns a subset m of a into a propositional valuation i,
2  // such that i(x) is true iff x is an element of m.
3  createValuation := procedure(m, a) {
4      return { [ x, x in m ] : x in a };
5  };
6  // Austin, Brian, or Colin is guilty.
7  f1 := [ [ "a", "+", "b" ], "+", "c" ];
8  // If Austin is guilty, he has exactly one accomplice.
9  f2 := [ "a", "->", [ "b", "+", "c" ] ]; // at least one accomplice
10 f3 := [ "a", "->", [ "-", [ "b", "*", "c" ] ] ]; // at most one accomplice
11 // If Brian is innocent, then Colin is innocent, too.
12 f4 := [ [ "-", "b" ], "->", [ "-", "c" ] ];
13 // If exactly two are guilty, then Colin is one of them.
14 f5 := [ "-", [ [ "a", "*", "b" ], "*", [ "-", "c" ] ] ];
15 // If Colin is innocent, then Austin is guilty.
16 f6 := [ [ "-", "c" ], "->", "a" ];
17 fs := { f1, f2, f3, f4, f5, f6 };
18 a := { "a", "b", "c" };
19 p := pow(a);
20 print("p = ", p);
21 // b is the set of all propositional valuations.
22 b := { createValuation(m, a) : m in p };
23 s := { i in b | forall (f in fs | eval(f, i)) };
24 print("Set of all valuations satisfying all facts: ", s);
25 if (#s == 1) {
26     i := arb(s);
27     taeter := { x in a | i(x) };
28     print("Set of culprits: ", taeter);
29 }

```

Abbildung 3.2: Programm zur Aufklärung des Einbruchs.

Wir diskutieren diese Programm nun Zeile für Zeile.

1. In den Zeilen 6 – 16 definieren wir die Formeln f_1, \dots, f_6 . Wir müssen hier die Formeln in die *SetIX*-Repräsentation bringen.
2. Als nächstes müssen wir uns überlegen, wie wir alle Belegungen aufzählen können. Wir hatten oben schon beobachtet, dass die Belegungen 1-zu-1 zu den möglichen Mengen der Täter korrespondieren. Die Mengen der möglichen Täter sind aber alle Teilmengen der Menge

$$\{\text{"a"}, \text{"b"}, \text{"c"}\}.$$

Wir berechnen daher in Zeile 18 zunächst die Menge aller dieser Teilmengen.

3. Wir brauchen jetzt eine Möglichkeit, eine Teilmenge in eine Belegung umzuformen. In den Zeilen 3 – 5 haben wir eine Prozedur implementiert, die genau dies leistet. Um zu verstehen, wie diese Funktion arbeitet, betrachten wir ein Beispiel und nehmen an, dass wir aus der Menge

$$m = \{ \text{"a"}, \text{"c"} \}$$

eine Belegung \mathcal{I} erstellen sollen. Wir erhalten dann

$$\mathcal{I} = \{ \langle \text{"a"}, \text{true} \rangle, \langle \text{"b"}, \text{false} \rangle, \langle \text{"c"}, \text{true} \rangle \}.$$

Das allgemeine Prinzip ist offenbar, dass für eine aussagenlogische Variable x das Paar $\langle x, \text{true} \rangle$ genau dann in der Belegung \mathcal{I} enthalten ist, wenn $x \in m$ ist, andernfalls ist das Paar $\langle x, \text{false} \rangle$ in \mathcal{I} . Damit könnten wir die Menge aller Belegungen, die genau die Elemente aus m wahr machen, wie folgt schreiben:

$$\{ [x, \text{true}] : x \in m \} + \{ [x, \text{false}] : x \in A \mid \neg(x \in m) \}$$

Es geht aber einfacher, denn wir können beide Fälle zusammenfassen, indem wir fordern, dass das Paar $\langle x, x \in m \rangle$ ein Element der Belegung \mathcal{I} ist. Genau das steht in Zeile 4.

4. In Zeile 22 sammeln wir in der Menge b alle möglichen Belegungen auf.
5. In Zeile 23 berechnen wir die Menge s aller der Belegungen i , für die alle Formeln aus der Menge fs wahr werden.
6. Falls es genau eine Belegung gibt, die alle Formeln wahr macht, dann haben wir das Problem lösen können. In diesem Fall extrahieren wir in Zeile 26 diese Belegungen aus der Menge s und geben anschließend die Menge der Täter aus.

Lassen wir das Programm laufen, so erhalten wir als Ausgabe

Menge der Täter: {"b", "c"}

Damit liefern unsere ursprünglichen Formeln ausreichende Information um die Täter zu überführen: Bruno und Claus sind schuldig.

3.4 Tautologien

Die Tabelle in Abbildung 3.2 zeigt, dass die Formel

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

für jede aussagenlogische Interpretation wahr ist, denn in der letzten Spalte dieser Tabelle steht immer der Wert **true**. Formeln mit dieser Eigenschaft bezeichnen wir als *Tautologie*.

Definition 2 (Tautologie) Ist f eine aussagenlogische Formel und gilt

$$\mathcal{I}(f) = \mathbf{true} \quad \text{für jede aussagenlogische Interpretation } \mathcal{I},$$

dann ist f eine *Tautologie*. In diesem Fall schreiben wir

$$\models f.$$

□

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

Beispiele:

1. $\models p \vee \neg p$
2. $\models p \rightarrow p$
3. $\models p \wedge q \rightarrow p$
4. $\models p \rightarrow p \vee q$
5. $\models (p \rightarrow \perp) \leftrightarrow \neg p$
6. $\models p \wedge q \leftrightarrow q \wedge p$

Wir können die Tatsache, dass es sich bei diesen Formeln um Tautologien handelt, durch eine Tabelle nachweisen, die analog zu der auf Seite 41 gezeigten Tabelle 3.2 aufgebaut ist. Dieses Verfahren ist zwar konzeptuell sehr einfach, allerdings zu ineffizient, wenn die Anzahl der aussagenlogischen Variablen groß ist. Ziel dieses Kapitels ist daher die Entwicklung besserer Verfahren.

Die letzten beiden Beispiele in der obigen Aufzählung geben Anlaß zu einer neuen Definition.

Definition 3 (Äquivalent) Zwei Formeln f und g heißen *äquivalent* g.d.w. gilt

$$\models f \leftrightarrow g$$

□

Beispiele: Es gelten die folgenden Äquivalenzen:

$\models \neg \perp \leftrightarrow \top$	$\models \neg \top \leftrightarrow \perp$	
$\models p \vee \neg p \leftrightarrow \top$	$\models p \wedge \neg p \leftrightarrow \perp$	Tertium-non-Datur
$\models p \vee \perp \leftrightarrow p$	$\models p \wedge \top \leftrightarrow p$	Neutrales Element
$\models p \vee \top \leftrightarrow \top$	$\models p \wedge \perp \leftrightarrow \perp$	
$\models p \wedge p \leftrightarrow p$	$\models p \vee p \leftrightarrow p$	Idempotenz
$\models p \wedge q \leftrightarrow q \wedge p$	$\models p \vee q \leftrightarrow q \vee p$	Kommutativität
$\models (p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$	$\models (p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$	Assoziativität
$\models \neg \neg p \leftrightarrow p$		Elimination von $\neg \neg$
$\models p \wedge (p \vee q) \leftrightarrow p$	$\models p \vee (p \wedge q) \leftrightarrow p$	Absorption
$\models p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	$\models p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$	Distributivität
$\models \neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$	$\models \neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$	DeMorgan'sche Regeln
$\models (p \rightarrow q) \leftrightarrow \neg p \vee q$		Elimination von \rightarrow
$\models (p \leftrightarrow q) \leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p)$		Elimination von \leftrightarrow

Wir können diese Äquivalenzen nachweisen, indem wir in einer Tabelle sämtliche Belegungen durchprobieren. Eine solche Tabelle heißt auch *Wahrheits-Tafel*. Wir demonstrieren dieses Verfahren anhand der ersten DeMorgan'schen Regel. Wir erkennen, dass in Abbildung 3.3 in den

p	q	$\neg p$	$\neg q$	$p \wedge q$	$\neg(p \wedge q)$	$\neg p \vee \neg q$
true	true	false	false	true	false	false
true	false	false	true	false	true	true
false	true	true	false	false	true	true
false	false	true	true	false	true	true

Tabelle 3.3: Nachweis der ersten DeMorgan'schen Regel.

letzten beiden Spalten in jeder Zeile die selben Werte stehen. Daher sind die Formeln, die zu diesen Spalten gehören, äquivalent.

3.4.1 Testen der Allgemeingültigkeit in *SetIX*

Die manuelle Überprüfung der Frage, ob eine gegebene Formel f eine Tautologie ist, läuft auf die Erstellung umfangreicher Wahrheitstafeln heraus. Solche Wahrheitstafeln von Hand zu erstellen ist viel zu zeitaufwendig. Wir wollen daher nun ein *SetIX*-Programm entwickeln, mit dessen Hilfe wir die obige Frage automatisch lösen können. Die Grundidee ist, dass wir die zu untersuchende Formel für alle möglichen Belegungen auswerten und überprüfen, dass sich bei der Auswertung jedesmal der Wert **true** ergibt. Dazu müssen wir zunächst einen Weg finden, alle möglichen Belegungen einer Formel zu berechnen. Wir haben früher schon gesehen, dass Belegungen \mathcal{I} zu Teilmengen M der Menge der aussagenlogischen Variablen \mathcal{P} korrespondieren, denn für jedes $M \subseteq \mathcal{P}$ können wir eine aussagenlogische Belegung $\mathcal{I}(M)$ wie folgt definieren:

$$\mathcal{I}(M)(p) := \begin{cases} \text{true} & \text{falls } p \in M; \\ \text{false} & \text{falls } p \notin M. \end{cases}$$

Um die aussagenlogische Belegung \mathcal{I} in *SetIX* darstellen zu können, fassen wir die Belegung \mathcal{I} als links-totale und rechts-eindeutige Relation $\mathcal{I} \subseteq \mathcal{P} \times \mathbb{B}$ auf. Dann haben wir

$$\mathcal{I} = \{ \langle p, \text{true} \rangle \mid p \in M \} \cup \{ \langle p, \text{false} \rangle \mid p \notin M \}.$$

Dies läßt sich noch zu

$$\mathcal{I} = \{ \langle p, p \in M \rangle \mid p \in \mathcal{P} \}$$

vereinfachen. Mit dieser Idee können wir nun eine Prozedur implementieren, die für eine gegebene aussagenlogische Formel f testet, ob f eine Tautologie ist.

```

1  tautology := procedure(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))) {
6          return {};
7      } else {
8          return arb({ i in a | !eval(f, i) });
9      }
10 };
11
12 collectVars := procedure(f) {
13     switch {
14         case f == 1 :
15             return {};
16         case f == 0 :
17             return {};
18         case isString(f) :
19             return { f };
20         case f(1) == "-" :
21             return collectVars( f(2) );
22         case f(2) in { "*", "+", "->", "<->" } :
23             return collectVars( f(1) ) + collectVars( f(3) );
24         default:
25             print("malformed formula: ", f);
26     }
27 };

```

Abbildung 3.3: Überprüfung der Allgemeingültigkeit einer aussagenlogischen Formel.

Die in Abbildung 3.3 auf Seite 49 gezeigte Prozedur `tautology(f)` testet, ob die als Argument übergebene aussagenlogische Formel f allgemeingültig ist. Die Prozedur verwendet die Prozedur `eval` aus dem in Abbildung 3.1 auf Seite 43 gezeigten Programm. Wir diskutieren die Definition der Funktion *tautology* nun Zeile für Zeile:

1. In Zeile 2 sammeln wir alle aussagenlogischen Variablen auf, die in der zu überprüfenden Formel auftreten. Die dazu benötigte Prozedur `collectVars` ist in den Zeilen 12 – 27 gezeigt. Diese Prozedur ist durch Induktion über den Aufbau einer Formel definiert und liefert als Ergebnis die Menge aller Aussage-Variablen, die in der aussagenlogischen Formel f auftreten.

Es ist klar, dass bei der Berechnung von $\mathcal{I}(f)$ für eine Formel f und eine aussagenlogische Interpretation \mathcal{I} nur die Werte von $\mathcal{I}(p)$ eine Rolle spielen, für die die Variable p in f auftritt. Zur Analyse von f können wir uns also auf aussagenlogische Interpretationen der Form

$$\mathcal{I} : \mathcal{P} \rightarrow \mathbb{B} \quad \text{mit} \quad \mathcal{P} = \text{collectVars}(f)$$

beschränken.

2. In Zeile 4 berechnen wir die Menge aller aussagenlogischen Interpretationen über der Menge \mathcal{P} der aussagenlogischen Variablen. Wir berechnen für eine Menge m von aussagenlogischen

Variablen die Interpretation $\mathcal{I}(m)$ wie oben diskutiert mit Hilfe der Formel

$$\mathcal{I}(m) := \{\langle x, x \in m \rangle \mid x \in V\}.$$

Betrachten wir zur Verdeutlichung als Beispiel die Formel

$$\neg(p \wedge q) \leftrightarrow \neg p \vee \neg q.$$

Die Menge \mathcal{P} der aussagenlogischen Variablen, die in dieser Formel auftreten, ist

$$\mathcal{P} = \{p, q\}.$$

Die Potenz-Menge der Menge \mathcal{P} ist

$$2^{\mathcal{P}} = \{\{\}, \{p\}, \{q\}, \{p, q\}\}.$$

Wir bezeichnen die vier Elemente dieser Menge mit m_1, m_2, m_3, m_4 :

$$m_1 := \{\}, m_2 := \{p\}, m_3 := \{q\}, m_4 := \{p, q\}.$$

Aus jeder dieser Mengen m_i gewinnen wir nun eine aussagenlogische Interpretation $\mathcal{I}(m_i)$:

$$\mathcal{I}(m_1) := \{\langle x, x \in \{\} \rangle \mid x \in \{p, q\}\} = \{\langle p, p \in \{\} \rangle, \langle q, q \in \{\} \rangle\} = \{\langle p, \text{false} \rangle, \langle q, \text{false} \rangle\}.$$

$$\mathcal{I}(m_2) := \{\langle x, x \in \{p\} \rangle \mid x \in \{p, q\}\} = \{\langle p, p \in \{p\} \rangle, \langle q, q \in \{p\} \rangle\} = \{\langle p, \text{true} \rangle, \langle q, \text{false} \rangle\}.$$

$$\mathcal{I}(m_3) := \{\langle x, x \in \{q\} \rangle \mid x \in \{p, q\}\} = \{\langle p, p \in \{q\} \rangle, \langle q, q \in \{q\} \rangle\} = \{\langle p, \text{false} \rangle, \langle q, \text{true} \rangle\}.$$

$$\mathcal{I}(m_4) := \{\langle x, x \in \{p, q\} \rangle \mid x \in \{p, q\}\} = \{\langle p, p \in \{p, q\} \rangle, \langle q, q \in \{p, q\} \rangle\} = \{\langle p, \text{true} \rangle, \langle q, \text{true} \rangle\}.$$

Damit haben wir alle möglichen Interpretationen der Variablen p und q .

3. In Zeile 5 testen wir, ob die Formel f für alle möglichen Interpretationen i aus der Menge a aller Interpretationen wahr ist. Ist dies der Fall, so geben wir die leere Menge als Ergebnis zurück.

Falls es allerdings eine Belegungen i in der Menge a gibt, für die die Auswertung von f den Wert *false* liefert, so bilden wir in Zeile 8 die Menge aller solcher Interpretationen und wählen mit Hilfe der Funktion *arb* eine beliebige Interpretation aus dieser Menge aus, die wir dann als Gegenbeispiel zurück geben.

3.4.2 Nachweis der Allgemeingültigkeit durch Äquivalenz-Umformungen

Wollen wir nachweisen, dass eine Formel eine Tautologie ist, können wir uns prinzipiell immer einer Wahrheits-Tafel bedienen. Aber diese Methode hat einen Haken: Kommen in der Formel n verschiedene Aussage-Variablen vor, so hat die Tabelle 2^n Zeilen. Beispielsweise hat die Tabelle zum Nachweis eines der Distributiv-Gesetze bereits 8 Zeilen, da hier 3 verschiedene Variablen auftreten. Eine andere Möglichkeit nachzuweisen, dass eine Formel eine Tautologie ist, ergibt sich dadurch, dass wir die Formel mit Hilfe der oben aufgeführten Äquivalenzen *vereinfachen*. Wenn es gelingt, eine Formel F unter Verwendung dieser Äquivalenzen zu \top zu vereinfachen, dann ist gezeigt, dass F eine Tautologie ist. Wir demonstrieren das Verfahren zunächst an einem Beispiel. Mit Hilfe einer Wahrheits-Tafel hatten wir schon gezeigt, dass die Formel

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

eine Tautologie ist. Wir zeigen nun, wie wir diesen Tatbestand auch durch eine Kette von Äquivalenz-Umformungen einsehen können:

	$(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$	(Elimination von \rightarrow)
\Leftrightarrow	$(\neg p \vee q) \rightarrow (\neg p \rightarrow q) \rightarrow q$	(Elimination von \rightarrow)
\Leftrightarrow	$(\neg p \vee q) \rightarrow (\neg \neg p \vee q) \rightarrow q$	(Elimination der Doppelnegation)
\Leftrightarrow	$(\neg p \vee q) \rightarrow (p \vee q) \rightarrow q$	(Elimination von \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 der Doppelnegation)
\Leftrightarrow	$(p \wedge \neg q) \vee ((p \vee q) \rightarrow q)$	(Elimination von \rightarrow)
\Leftrightarrow	$(p \wedge \neg q) \vee (\neg(p \vee q) \vee q)$	(DeMorgan)
\Leftrightarrow	$(p \wedge \neg q) \vee ((\neg p \wedge \neg q) \vee q)$	(Distributivität)
\Leftrightarrow	$(p \wedge \neg q) \vee ((\neg p \vee q) \wedge (\neg q \vee q))$	(Tertium-non-Datur)
\Leftrightarrow	$(p \wedge \neg q) \vee ((\neg p \vee q) \wedge \top)$	(Neutrales Element)
\Leftrightarrow	$(p \wedge \neg q) \vee (\neg p \vee q)$	(Distributivität)
\Leftrightarrow	$(p \vee (\neg p \vee q)) \wedge (\neg q \vee (\neg p \vee q))$	(Assoziativität)
\Leftrightarrow	$((p \vee \neg p) \vee q) \wedge (\neg q \vee (\neg p \vee q))$	(Tertium-non-Datur)
\Leftrightarrow	$(\top \vee q) \wedge (\neg q \vee (\neg p \vee q))$	(Neutrales Element)
\Leftrightarrow	$\top \wedge (\neg q \vee (\neg p \vee q))$	(Neutrales Element)
\Leftrightarrow	$\neg q \vee (\neg p \vee q)$	(Assoziativität)
\Leftrightarrow	$(\neg q \vee \neg p) \vee q$	(Kommutativität)
\Leftrightarrow	$(\neg p \vee \neg q) \vee q$	(Assoziativität)
\Leftrightarrow	$\neg p \vee (\neg q \vee q)$	(Tertium-non-Datur)
\Leftrightarrow	$\neg p \vee \top$	(Neutrales Element)
\Leftrightarrow	\top	

Die Umformungen in dem obigen Beweis sind nach einem bestimmten System durchgeführt worden. Um dieses System präzise formulieren zu können, brauchen wir noch einige Definitionen.

Definition 4 (Literal) Eine aussagenlogische Formel f heißt *Literal* g.d.w. einer der folgenden Fälle vorliegt:

1. $f = \top$ oder $f = \perp$.
2. $f = p$, wobei p eine aussagenlogische Variable ist.
In diesem Fall sprechen wir von einem *positiven* Literal.
3. $f = \neg p$, wobei p eine aussagenlogische Variable ist.
In diesem Fall sprechen wir von einem *negativen* Literal.

Die Menge aller Literale bezeichnen wir mit \mathcal{L} . □

Später werden wir noch den Begriff des *Komplements* eines Literals benötigen. Ist l ein Literal, so wird das Komplement von l mit \overline{l} bezeichnet. Das Komplement wird durch Fall-Unterscheidung definiert:

1. $\overline{\top} = \perp$ und $\overline{\perp} = \top$.
2. $\overline{p} := \neg p$, falls $p \in \mathcal{P}$.
3. $\overline{\neg p} := p$, falls $p \in \mathcal{P}$.

Definition 5 (Klausel) Eine aussagenlogische Formel k ist eine *Klausel* wenn k die Form

$$k = l_1 \vee \dots \vee l_r$$

hat, wobei l_i für alle $i = 1, \dots, r$ ein Literal ist. Eine Klausel ist also eine Disjunktion von Literalen. Die Menge aller Klauseln bezeichnen wir mit \mathcal{K} . □

Oft werden Klauseln auch einfach als *Mengen* von Literalen betrachtet. Durch diese Sichtweise abstrahieren wir von der Reihenfolge und der Anzahl des Auftretens der Literale in der Disjunktion. Dies ist möglich aufgrund der Assoziativität, Kommutativität und Idempotenz des Junktors “ \vee ”. Für die Klausel $l_1 \vee \dots \vee l_r$ schreiben wir also in Zukunft auch

$$\{l_1, \dots, l_r\}.$$

Das folgende Beispiel illustriert die Nützlichkeit der Mengen-Schreibweise von Klauseln. Wir betrachten die beiden Klauseln

$$p \vee q \vee \neg r \vee p \quad \text{und} \quad \neg r \vee q \vee \neg r \vee p.$$

Die beiden Klauseln sind zwar äquivalent, aber die Formeln sind verschieden. Überführen wir die beiden Klauseln in Mengen-Schreibweise, so erhalten wir

$$\{p, q, \neg r\} \quad \text{und} \quad \{\neg r, q, p\}.$$

In einer Menge kommt jedes Element höchstens einmal vor und die Reihenfolge, in der die Elemente auftreten, spielt auch keine Rolle. Daher sind die beiden obigen Mengen gleich! Durch die Tatsache, dass Mengen von der Reihenfolge und der Anzahl der Elemente abstrahieren, implementiert die Mengen-Schreibweise die Assoziativität, Kommutativität und Idempotenz der Disjunktion. Übertragen wir die aussagenlogische Äquivalenz

$$l_1 \vee \dots \vee l_r \vee \perp \leftrightarrow l_1 \vee \dots \vee l_r$$

in Mengen-Schreibweise, so erhalten wir

$$\{l_1, \dots, l_r, \perp\} \leftrightarrow \{l_1, \dots, l_r\}.$$

Dies zeigt, dass wir das Element \perp in einer Klausel getrost weglassen können. Betrachten wir die letzten Äquivalenz für den Fall, dass $r = 0$ ist, so haben wir

$$\{\perp\} \leftrightarrow \{\}.$$

Damit sehen wir, dass die leere Menge von Literalen als \perp zu interpretieren ist.

Definition 6 Eine Klausel k ist *trivial*, wenn einer der beiden folgenden Fälle vorliegt:

1. $\top \in k$.
2. Es existiert $p \in \mathcal{P}$ mit $p \in k$ und $\neg p \in k$.

In diesem Fall bezeichnen wir p und $\neg p$ als *komplementäre Literale*. □

Satz 7 Eine Klausel ist genau dann eine Tautologie, wenn sie trivial ist.

Beweis: Wir nehmen zunächst an, dass die Klausel k trivial ist. Falls nun $\top \in k$ ist, dann gilt wegen der Gültigkeit der Äquivalenz $f \vee \top \leftrightarrow \top$ offenbar $k \leftrightarrow \top$. Ist p eine Aussage-Variable, so dass sowohl $p \in k$ als auch $\neg p \in k$ gilt, dann folgt aufgrund der Äquivalenz $p \vee \neg p \leftrightarrow \top$ sofort $k \leftrightarrow \top$.

Wir nehmen nun an, dass die Klausel k eine Tautologie ist. Wir führen den Beweis indirekt und nehmen an, dass k nicht trivial ist. Damit gilt $\top \notin k$ und k kann auch keine komplementären Literale enthalten. Damit hat k dann die Form

$$k = \{\neg p_1, \dots, \neg p_m, q_1, \dots, q_n\} \quad \text{mit } p_i \neq q_j \text{ für alle } i \in \{1, \dots, m\} \text{ und } j \in \{1, \dots, n\}.$$

Dann könnten wir eine Interpretation \mathcal{I} wie folgt definieren:

1. $\mathcal{I}(p_i) = \mathbf{true}$ für alle $i = 1, \dots, m$ und
2. $\mathcal{I}(q_j) = \mathbf{false}$ für alle $j = 1, \dots, n$,

Mit dieser Interpretation würde offenbar $\mathcal{I}(k) = \mathbf{false}$ gelten und damit könnte k keine Tautologie sein. Also ist die Annahme, dass k nicht trivial ist, falsch. □

Definition 8 (Konjunktive Normalform) Eine Formel f ist in *konjunktiver Normalform* (kurz KNF) genau dann, wenn f eine Konjunktion von Klauseln ist, wenn also gilt

$$f = k_1 \wedge \cdots \wedge k_n,$$

wobei die k_i für alle $i = 1, \dots, n$ Klauseln sind. □

Aus der Definition der KNF folgt sofort:

Korollar 9 Ist $f = k_1 \wedge \cdots \wedge k_n$ in konjunktiver Normalform, so gilt

$$\models f \quad \text{genau dann, wenn} \quad \models k_i \quad \text{für alle } i = 1, \dots, n. \quad \square$$

Damit können wir für eine Formel $f = k_1 \wedge \cdots \wedge k_n$ in konjunktiver Normalform leicht entscheiden, ob f eine Tautologie ist, denn f ist genau dann eine Tautologie, wenn alle Klauseln k_i trivial sind.

Da für die Konjunktion genau wie für die Disjunktion Assoziativ-Gesetz, Kommutativ-Gesetz und Idempotenz-Gesetz gilt, ist es zweckmäßig, auch für Formeln in konjunktiver Normalform eine Mengen-Schreibweise einzuführen. Ist also die Formel

$$f = k_1 \wedge \cdots \wedge k_n$$

in konjunktiver Normalform, so repräsentieren wir diese Formel durch die Menge ihrer Klauseln und schreiben

$$f = \{k_1, \dots, k_n\}.$$

Wir geben ein Beispiel: Sind p, q und r Aussage-Variablen, so ist die Formel

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

in konjunktiver Normalform. In Mengen-Schreibweise wird daraus

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

Wir stellen nun ein Verfahren vor, mit dem sich jede Formel in KNF transformieren läßt. Nach dem oben Gesagten können wir dann leicht entscheiden, ob f eine Tautologie ist.

1. Eliminiere alle Vorkommen des Junktors “ \leftrightarrow ” mit Hilfe der Äquivalenz

$$(f \leftrightarrow g) \leftrightarrow (f \rightarrow g) \wedge (g \rightarrow f)$$

2. Eliminiere alle Vorkommen des Junktors “ \rightarrow ” mit Hilfe der Äquivalenz

$$(f \rightarrow g) \leftrightarrow \neg f \vee g$$

3. Schiebe die Negationszeichen soweit es geht nach innen. Verwende dazu die folgenden Äquivalenzen:

$$(a) \quad \neg \perp \leftrightarrow \top$$

$$(b) \quad \neg \top \leftrightarrow \perp$$

$$(c) \quad \neg \neg f \leftrightarrow f$$

$$(d) \quad \neg(f \wedge g) \leftrightarrow \neg f \vee \neg g$$

$$(e) \quad \neg(f \vee g) \leftrightarrow \neg f \wedge \neg g$$

In dem Ergebnis, das wir nach diesem Schritt erhalten, stehen die Negationszeichen nur noch unmittelbar vor den aussagenlogischen Variablen. Formeln mit dieser Eigenschaft bezeichnen wir auch als Formeln in *Negations-Normalform*.

4. Stehen in der Formel jetzt “ \vee ”-Junktoren über “ \wedge ”-Junktoren, so können wir durch *Ausmultiplizieren*, sprich Verwendung der Distributiv-Gesetze

$$f \vee (g \wedge h) \leftrightarrow (f \vee g) \wedge (f \vee h) \quad \text{und} \quad (f \wedge g) \vee h \leftrightarrow (f \vee h) \wedge (g \vee h)$$

diese Junktoren nach innen schieben.

5. In einem letzten Schritt überführen wir die Formel nun in Mengen-Schreibweise, indem wir zunächst die Disjunktionen aller Literale als Mengen zusammenfassen und anschließend alle so entstandenen Klauseln wieder in einer Menge zusammen fassen.

Hier sollten wir noch bemerken, dass die Formel beim Ausmultiplizieren stark anwachsen kann. Das liegt daran, dass die Formel f auf der rechten Seite der Äquivalenz $f \vee (g \wedge h) \leftrightarrow (f \vee g) \wedge (f \vee h)$ zweimal auftritt, während sie links nur einmal vorkommt.

Wir demonstrieren das Verfahren am Beispiel der Formel

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

1. Da die Formel den Junktor “ \leftrightarrow ” nicht enthält, ist im ersten Schritt nichts zu tun.
2. Die Elimination von “ \rightarrow ” liefert

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

3. Die Umrechnung auf Negations-Normalform liefert

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

4. Durch “Ausmultiplizieren” erhalten wir

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

5. Die Überführung in die Mengen-Schreibweise ergibt zunächst als Klauseln die beiden Mengen

$$\{p, p, \neg q\} \quad \text{und} \quad \{\neg q, p, \neg q\}.$$

Da die Reihenfolge der Elemente einer Menge aber unwichtig ist und außerdem eine Menge jedes Element nur einmal enthält, stellen wir fest, dass diese beiden Klauseln gleich sind. Fassen wir jetzt die Klauseln noch in einer Menge zusammen, so erhalten wir

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

Beachten Sie, dass sich die Formel durch die Überführung in Mengen-Schreibweise noch einmal deutlich vereinfacht hat.

Damit ist die Formel in KNF überführt.

3.4.3 Berechnung der konjunktiven Normalform in *SetIX*

Wir geben nun eine Reihe von Prozeduren an, mit deren Hilfe sich eine gegebene Formel f in konjunktive Normalform überführen läßt. Wir beginnen mit einer Prozedur

$$\text{elimGdw} : \mathcal{F} \rightarrow \mathcal{F}$$

die die Aufgabe hat, eine vorgegebene aussagenlogische Formel f in eine äquivalente Formel umzuformen, die den Junktor “ \leftrightarrow ” nicht mehr enthält. Die Funktion $\text{elimGdw}(f)$ wird durch Induktion über den Aufbau der aussagenlogischen Formel f definiert. Dazu stellen wir zunächst rekursive Gleichungen auf, die das Verhalten der Funktion $\text{elimGdw}()$ beschreiben:

1. Hat f die Form $f = \top$ oder $f = \perp$, oder wenn f eine Aussage-Variable p ist, so ist nichts zu tun:
 - (a) $\text{elimGdw}(\top) = \top$.
 - (b) $\text{elimGdw}(\perp) = \perp$.
 - (c) $\text{elimGdw}(p) = p$ für alle $p \in \mathcal{P}$.

2. Hat f die Form $f = \neg g$, so eliminieren wir den Junktor “ \neg ” aus der Formel g :

$$\text{elimGdw}(\neg g) = \neg \text{elimGdw}(g).$$

3. Im Falle $f = g_1 \wedge g_2$ eliminieren wir den Junktor “ \wedge ” aus den Formeln g_1 und g_2 :

$$\text{elimGdw}(g_1 \wedge g_2) = \text{elimGdw}(g_1) \wedge \text{elimGdw}(g_2).$$

4. Im Falle $f = g_1 \vee g_2$ eliminieren wir den Junktor “ \vee ” aus den Formeln g_1 und g_2 :

$$\text{elimGdw}(g_1 \vee g_2) = \text{elimGdw}(g_1) \vee \text{elimGdw}(g_2).$$

5. Im Falle $f = g_1 \rightarrow g_2$ eliminieren wir den Junktor “ \rightarrow ” aus den Formeln g_1 und g_2 :

$$\text{elimGdw}(g_1 \rightarrow g_2) = \text{elimGdw}(g_1) \rightarrow \text{elimGdw}(g_2).$$

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

$$(g_1 \leftrightarrow g_2) \leftrightarrow ((g_1 \rightarrow g_2) \wedge (g_2 \rightarrow g_1)).$$

Das führt auf die Gleichung:

$$\text{elimGdw}(g_1 \leftrightarrow g_2) = \text{elimGdw}((g_1 \rightarrow g_2) \wedge (g_2 \rightarrow g_1)).$$

Der Aufruf von `elimGdw` auf der rechten Seite der Gleichung ist notwendig, denn der Junktor “ \leftrightarrow ” kann ja noch in g_1 und g_2 auftreten.

Abbildung 3.4 auf Seite 55 zeigt die Implementierung der Prozedur `elimGdw`.

```

1  elimGdw := procedure(f) {
2      switch {
3          case f == 1:
4              return 1;
5          case f == 0:
6              return 0;
7          case isString(f):
8              return f;
9          case f(1) == "-":
10             return [ "-", elimGdw(f(2)) ];
11          case f(2) == "*":
12             return [ elimGdw(f(1)), "*", elimGdw(f(3)) ];
13          case f(2) == "+":
14             return [ elimGdw(f(1)), "+", elimGdw(f(3)) ];
15          case f(2) == "->":
16             return [ elimGdw(f(1)), "->", elimGdw(f(3)) ];
17          case f(2) == "<->":
18             return elimGdw([[f(1), "->", f(3)], "*", [f(3), "->", f(1) ]]);
19          default:
20             printErr("Fehler in elimGdw( $$$ )" );
21      }
22  };

```

Abbildung 3.4: Elimination von \leftrightarrow .

Als nächstes betrachten wir die Prozedur zur Elimination des Junktors “ \rightarrow ”. Abbildung 3.5 auf Seite 56 zeigt die Implementierung. Die der Implementierung zu Grunde liegende Idee ist die selbe wie bei der Elimination des Junktors “ \leftrightarrow ”. Der einzige Unterschied besteht darin, dass wir jetzt die Äquivalenz

$$(g_1 \rightarrow g_2) \leftrightarrow (\neg g_1 \vee g_2)$$

benutzen. Außerdem können wir schon voraussetzen, dass der Junktoren “ \leftrightarrow ” bereits vorher eliminiert wurde. Dadurch entfällt ein Fall.

```

1  elimFolgt := procedure(f) {
2      switch {
3          case f == 1:
4              return 1;
5          case f == 0:
6              return 0;
7          case isString(f):
8              return f;
9          case f(1) == "-":
10             return [ "-", elimFolgt(f(2)) ];
11          case f(2) == "*":
12             return [ elimFolgt(f(1)), "*", elimFolgt(f(3)) ];
13          case f(2) == "+":
14             return [ elimFolgt(f(1)), "+", elimFolgt(f(3)) ];
15          case f(2) == "->":
16             return elimFolgt([ "-", f(1) ], "+", f(3));
17          default:
18             print("Fehler in elimFolgt( $f$ )" );
19      }
20 };

```

Abbildung 3.5: Elimination von \rightarrow .

Als nächstes zeigen wir die Routinen zur Berechnung der Negations-Normalform. Abbildung 3.6 auf Seite 58 zeigt die Implementierung. Hier erfolgt die Implementierung durch die beiden Prozeduren **nnf** und **neg**, die sich wechselseitig aufrufen. Dabei berechnet **neg**(f) die Negations-Normalform von $\neg f$, während **nnf**(f) die Negations-Normalform von f berechnet, es gilt also

$$\text{neg}(f) = \text{nnf}(\neg f).$$

Die eigentliche Arbeit wird dabei in der Funktion **neg** erledigt, denn dort kommen die beiden DeMorgan'schen Gesetze

$$\neg(f \wedge g) \leftrightarrow (\neg f \vee \neg g) \quad \text{und} \quad \neg(f \vee g) \leftrightarrow (\neg f \wedge \neg g)$$

zur Anwendung. Wir beschreiben die Umformung in Negations-Normalform durch die folgenden Gleichungen:

1. $\text{nnf}(\top) = \top$,
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. $\text{nnf}(f_1 \vee f_2) = \text{nnf}(f_1) \vee \text{nnf}(f_2)$.

Die Hilfsprozedur **neg**, die die Negations-Normalform von $\neg f$ berechnet, spezifizieren wir ebenfalls durch rekursive Gleichungen:

1. $\text{neg}(\top) = \text{nnf}(\neg \top) = \perp$,

2. $\text{neg}(\perp) = \text{nnf}(\neg\perp) = \top$,
3. $\text{neg}(p) = \text{nnf}(\neg p) = \neg p$ für alle Aussage-Variablen 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.
$$\begin{aligned}
& \text{neg}(f_1 \vee f_2) \\
&= \text{nnf}(\neg(f_1 \vee f_2)) \\
&= \text{nnf}(\neg f_1 \wedge \neg f_2) \\
&= \text{nnf}(\neg f_1) \wedge \text{nnf}(\neg f_2) \\
&= \text{neg}(f_1) \wedge \text{neg}(f_2).
\end{aligned}$$

Als letztes stellen wir die Prozeduren vor, mit denen die Formeln, die bereits in Negations-Normalform sind, ausmultipliziert und dadurch in konjunktive Normalform gebracht werden. Gleichzeitig werden die zu normalisierende Formel dabei in die Mengen-Schreibweise transformiert, d.h. die Formeln werden als Mengen von Mengen von Literalen dargestellt. Dabei interpretieren wir eine Menge von Literalen als Disjunktion der Literale und eine Menge von Klauseln interpretieren wir als Konjunktion der Klauseln. Abbildung 3.7 auf Seite 59 zeigt die Implementierung.

1. Zunächst überlegen wir uns, wie wir \top in der Mengen-Schreibweise darstellen können. Da \top das neutrale Element der Konjunktion ist, haben wir die folgende Äquivalenz:

$$k_1 \wedge \dots \wedge k_n \wedge \top \leftrightarrow k_1 \wedge \dots \wedge k_n.$$

Sind nun k_1, \dots, k_n Klauseln, so hat die obige Äquivalenz in Mengen-Schreibweise die folgende Form:

$$\{k_1, \dots, k_n, \top\} \leftrightarrow \{k_1, \dots, k_n\}$$

Wir vereinbaren, dass diese Äquivalenz auch für $n = 0$ gelten soll. Dann haben wir

$$\{\top\} \leftrightarrow \{\},$$

in der der Mengen-Schreibweise interpretieren wir die leere Menge $\{\}$ von Klauseln als \top .

Die obigen Überlegungen erklären die Zeile 3 der Prozedur **knf**.

2. Als nächstes überlegen wir uns, wie wir \perp in der Mengen-Schreibweise darstellen können: Da \perp das neutrale Element der Disjunktion ist, haben wir die folgende Äquivalenz:

$$L_1 \vee \dots \vee L_n \vee \perp \leftrightarrow L_1 \vee \dots \vee L_n.$$

Sind nun L_1, \dots, L_n Literale, so hat die obige Äquivalenz in Mengen-Schreibweise die folgende Form:

$$\{L_1, \dots, L_n, \perp\} \leftrightarrow \{L_1, \dots, L_n\}$$

Wir vereinbaren, dass diese Äquivalenz auch für $n = 0$ gelten soll. Dann haben wir

$$\{\perp\} \leftrightarrow \{\},$$

wir interpretieren also in der der Mengen-Schreibweise eine leere Menge $\{\}$ von Literalen als \perp . Diese leere Menge repräsentiert eine Klausel. Um die Formel \perp in KNF darzustellen, erhalten wir den Ausdruck $\{\{\}\}$, denn eine Formel in KNF ist ja eine Menge von Klauseln.

Die obigen Überlegungen erklären die Zeile 4 der Prozedur **knf**.

```

1  nnf := procedure(f) {
2      switch {
3          case f == 1:
4              return 1;
5          case f == 0:
6              return 0;
7          case isString(f):
8              return f;
9          case f(1) == "-":
10             return neg( f(2) );
11          case f(2) == "*":
12             return [ nnf(f(1)), "*", nnf(f(3)) ];
13          case f(2) == "+":
14             return [ nnf(f(1)), "+", nnf(f(3)) ];
15          default:
16             print("Fehler in nnf( $$ )" );
17      }
18 };
19
20 neg := procedure(f) {
21     switch {
22         case f == 1:
23             return 0;
24         case f == 0:
25             return 1;
26         case isString(f):
27             return [ "-", f ];
28         case f(1) == "-":
29             return nnf( f(2) );
30         case f(2) == "*":
31             return [ neg(f(1)), "+", neg(f(3)) ];
32         case f(2) == "+":
33             return [ neg(f(1)), "*", neg(f(3)) ];
34         default:
35             print("Fehler in neg( $$ )" );
36     }
37 };

```

Abbildung 3.6: Berechnung der Negations-Normalform.

3. Falls die Formel f , die wir in KNF transformieren wollen, eine Aussage-Variable ist, so transformieren wir f zunächst in eine Klausel. Das liefert $\{f\}$. Da eine KNF eine Menge von Klauseln ist, ist die KNF dann $\{\{f\}\}$. Dieses Ergebnis geben wir in Zeile 5 zurück.

4. Falls die Formel f , die wir in KNF transformieren wollen, die Form

$$f = \neg g$$

hat, so muss g eine Aussage-Variable sein, denn f ist ja in Negations-Normalform. Damit können wir f in eine Klausel transformieren, indem wir $\{\neg g\}$, also $\{f\}$ schreiben. Da eine KNF eine Menge von Klauseln ist, ist dann $\{\{f\}\}$ das Ergebnis, das wir in Zeile 6 zurück geben.

5. Falls $f = f_1 \wedge f_2$ ist, transformieren wir zunächst f_1 und f_2 in KNF. Dabei erhalten wir

Dabei sind die h_i und die k_j Klauseln. Um nun die KNF von $f_1 \wedge f_2$ zu bilden, reicht es aus, die Vereinigung dieser beiden Mengen zu bilden, wir haben also

Das liefert Zeile 7 der Implementierung.

- $$\text{knf}(f_1) = \{h_1, \dots, h_m\} \quad \text{und} \quad \text{knf}(f_2) = \{k_1, \dots, k_n\}.$$

$$\begin{aligned}
& f_1 \vee f_2 \\
\Leftrightarrow & (h_1 \wedge \cdots \wedge h_m) \vee (k_1 \wedge \cdots \wedge k_n) \\
\Leftrightarrow & (h_1 \vee k_1) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_1) \quad \wedge \\
& \qquad \vdots \qquad \qquad \qquad \qquad \qquad \qquad \vdots \\
& (h_1 \vee k_n) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_n) \\
\Leftrightarrow & \{h_i \vee k_j : i \in \{1, \dots, m\}, j \in \{1, \dots, n\}\}
\end{aligned}$$
$$\mathbf{k}\mathbf{n}\mathbf{f}(f_1 \vee f_2) = \{h \cup k \mid h \in \mathbf{k}\mathbf{n}\mathbf{f}(f_1) \wedge k \in \mathbf{k}\mathbf{n}\mathbf{f}(f_2)\}.$$

```

1 knf := procedure(f) {
2   switch {
3     case f == 1:
4       return {};
5     case f == 0:
6       return { {} };
7     case isString(f):
8       return { { f } };
9     case f(1) == "-":
10      return { { f } };
11     case f(2) == "*":
12      return knf(f(1)) + knf(f(3));
13     case f(2) == "+":
14      return { k1 + k2 : k1 in knf(f(1)), k2 in knf(f(3)) };
15     default:
16      print("Fehler in knf( $f$ )" );
17   }
18 };

```

Zum Abschluß zeigen wir in Abbildung 3.8 auf Seite 60 wie die einzelnen Funktionen zusammenspielen.

```

1  normalize := procedure(f) {
2      n1 := elimGdw(f);
3      n2 := elimFolgt(n1);
4      n3 := nnf(n2);
5      n4 := knf(n3);
6      return n4;
7  };

```

Abbildung 3.8: Normalisierung einer Formel

3.5 Der Herleitungs-Begriff

Ist $\{f_1, \dots, f_n\}$ eine Menge von Formeln, und g eine weitere Formel, so können wir uns fragen, ob die Formel g aus f_1, \dots, f_n *folgt*, ob also

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

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

$$f_1 \wedge \dots \wedge f_n \rightarrow g$$

äquivalent ist. Diese Formel ist nun genau dann eine Tautologie, wenn jede der Klauseln k_1, \dots, k_n trivial ist.

Das oben dargestellte Verfahren ist aber sehr aufwendig. Wir zeigen dies an Hand eines Beispiels und wenden das Verfahren an, um zu entscheiden, ob $p \rightarrow r$ aus den beiden Formeln $p \rightarrow q$ und $q \rightarrow r$ folgt. Wir bilden also die konjunktive Normalform der Formel

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

und erhalten

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

Zwar können wir jetzt sehen, dass die Formel h eine Tautologie ist, aber angesichts der Tatsache, dass wir mit bloßem Auge sehen, dass $p \rightarrow r$ aus den Formeln $p \rightarrow q$ und $q \rightarrow r$ folgt, ist die Rechnung doch sehr mühsam.

Wir stellen daher nun ein weiteres Verfahren vor, mit dessen Hilfe wir entscheiden können, ob eine Formel aus einer gegebenen Menge von Formeln folgt. Die Idee bei diesem Verfahren ist es, die Formel f mit Hilfe von *Schluss-Regeln* aus den gegebenen Formeln f_1, \dots, f_n herzuleiten. Das Konzept einer Schluss-Regel wird in der nun folgenden Definition festgelegt.

Definition 10 (Schluss-Regel) Eine *Schluss-Regel* ist eine Paar $\langle \{f_1, \dots, f_n\}, k \rangle$. Dabei ist $\{f_1, \dots, f_n\}$ eine Menge von Formeln und k ist eine einzelne Formel. Die Formeln f_1, \dots, f_n bezeichnen wir als *Prämissen*, die Formel k heißt die *Konklusion* der Schluss-Regel. Ist das Paar $\langle \{f_1, \dots, f_n\}, k \rangle$ eine Schluss-Regel, so schreiben wir dies als:

$$\frac{f_1 \quad \dots \quad f_n}{k}.$$

Wir lesen diese Schluss-Regel wie folgt: “Aus f_1, \dots, f_n kann auf k geschlossen werden.” \square

Beispiele für Schluss-Regeln:

<i>Modus Ponens:</i>	<i>Modus Tollens:</i>	<i>Modus Tollendo Tollens:</i>
$\frac{p \quad p \rightarrow q}{q}$	$\frac{\neg q \quad p \rightarrow q}{\neg p}$	$\frac{\neg p \quad p \rightarrow q}{\neg q}$

Die Definition der Schluss-Regel schränkt zunächst die Formeln, die als Prämissen bzw. Konklusion verwendet werden können, nicht weiter ein. Es ist aber sicher nicht sinnvoll, beliebige Schluss-Regeln zuzulassen. Wollen wir Schluss-Regeln in Beweisen verwenden, so sollten die Schluss-Regeln in dem in der folgenden Definition erklärten Sinne *korrekt* sein.

Definition 11 (Korrekte Schluss-Regel) Eine Schluss-Regel

$$\frac{f_1 \quad \cdots \quad f_n}{k}$$

ist genau dann *korrekt*, wenn $\models f_1 \wedge \cdots \wedge f_n \rightarrow k$ gilt. \square

Mit dieser Definition sehen wir, dass die oben als “*Modus Ponens*” und “*Modus Ponendo Tollens*” bezeichneten Schluss-Regeln korrekt sind, während die als “*Modus Tollendo Tollens*” bezeichnete Schluss-Regel nicht korrekt ist.

Im folgenden gehen wir davon aus, dass alle Formeln Klauseln sind. Einerseits ist dies keine echte Einschränkung, denn wir können ja jede Formel in eine äquivalente Menge von Klauseln umrechnen. Andererseits haben viele in der Praxis auftretende aussagenlogische Probleme die Gestalt von Klauseln. Daher stellen wir jetzt eine Schluss-Regel vor, in der sowohl die Prämissen als auch die Konklusion Klauseln sind.

Definition 12 (Schnitt-Regel) Ist p eine aussagenlogische Variable und sind k_1 und k_2 Mengen von Literalen, die wir als Klauseln interpretieren, so bezeichnen wir die folgende Schluss-Regel als die *Schnitt-Regel*:

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

\square

Die Schnitt-Regel ist sehr allgemein. Setzen wir in der obigen Definition für $k_1 = \{\}$ und $k_2 = \{q\}$ ein, so erhalten wir die folgende Regel als Spezialfall:

$$\frac{\{\} \cup \{p\} \quad \{\neg p\} \cup \{q\}}{\{\} \cup \{q\}}$$

Interpretieren wir nun die Mengen als Disjunktionen, so haben wir:

$$\frac{p \quad \neg p \vee q}{q}$$

Wenn wir jetzt noch berücksichtigen, dass die Formel $\neg p \vee q$ äquivalent ist zu der Formel $p \rightarrow q$, dann ist das nichts anderes als der *Modus Ponens*. Die Regel *Modus Tollens* ist ebenfalls ein Spezialfall der Schnitt-Regel. Wir erhalten diese Regel, wenn wir in der Schnitt-Regel $k_1 = \{\neg q\}$ und $k_2 = \{\}$ setzen.

Satz 13 Die Schnitt-Regel ist korrekt.

Beweis: Wir müssen zeigen, dass

$$\models (k_1 \vee p) \wedge (\neg p \vee k_2) \rightarrow k_1 \vee k_2$$

gilt. Dazu überführen wir die obige Formel in konjunktive Normalform:

$$\begin{aligned}
& (k_1 \vee p) \wedge (\neg p \vee k_2) \rightarrow k_1 \vee k_2 \\
\leftrightarrow & \neg((k_1 \vee p) \wedge (\neg p \vee k_2)) \vee k_1 \vee k_2 \\
\leftrightarrow & \neg(k_1 \vee p) \vee \neg(\neg p \vee k_2) \vee k_1 \vee k_2 \\
\leftrightarrow & (\neg k_1 \wedge \neg p) \vee (p \wedge \neg k_2) \vee k_1 \vee k_2 \\
\leftrightarrow & (\neg k_1 \vee p \vee k_1 \vee k_2) \wedge (\neg k_1 \vee \neg k_2 \vee k_1 \vee k_2) \wedge (\neg p \vee p \vee k_1 \vee k_2) \wedge (\neg p \vee \neg k_2 \vee k_1 \vee k_2) \\
\leftrightarrow & \top \wedge \top \wedge \top \wedge \top \\
\leftrightarrow & \top
\end{aligned}$$

□

Definition 14 (\vdash) Es sei M eine Menge von Klauseln und f sei eine einzelne Klausel. Die Formeln aus M bezeichnen wir als unsere Annahmen. Unser Ziel ist es, mit diesen Annahmen die Formel f zu beweisen. Dazu definieren wir induktiv die Relation

$$M \vdash f.$$

Wir lesen “ $M \vdash f$ ” als “ M leitet f her”. Die induktive Definition ist wie folgt:

1. Aus einer Menge M von Annahmen kann jede der Annahmen hergeleitet werden:

$$\text{Falls } f \in M \text{ ist, dann gilt } M \vdash f.$$

2. Sind $k_1 \cup \{p\}$ und $\{\neg p\} \cup k_2$ Klauseln, die aus M hergeleitet werden können, so kann mit der Schnitt-Regel auch die Klausel $k_1 \cup k_2$ aus M hergeleitet werden:

$$\text{Falls sowohl } M \vdash k_1 \cup \{p\} \text{ als auch } M \vdash \{\neg p\} \cup k_2 \text{ gilt, dann gilt auch } M \vdash k_1 \cup k_2.$$

Dies ist die Schnitt-Regel. □

Beispiel: Um den Beweis-Begriff zu veranschaulichen geben wir ein Beispiel und zeigen

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

Gleichzeitig zeigen wir an Hand des Beispiels, wie wir Beweise zu Papier bringen:

1. Aus $\{ \neg p, q \}$ und $\{ \neg q, \neg p \}$ folgt mit der Schnitt-Regel $\{ \neg p, \neg p \}$. Wegen $\{ \neg p, \neg p \} = \{ \neg p \}$ schreiben wir dies als

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

Dieses Beispiel zeigt, dass die Klausel $k_1 \cup k_2$ durchaus auch weniger Elemente enthalten kann als die Summe $\#k_1 + \#k_2$. Dieser Fall tritt genau dann ein, wenn es Literale gibt, die sowohl in k_1 als auch in k_2 vorkommen.

2. $\{ \neg q, \neg p \}, \{ p, \neg q \} \vdash \{ \neg q \}.$
3. $\{ p, q \}, \{ \neg q \} \vdash \{ p \}.$
4. $\{ \neg p \}, \{ p \} \vdash \{ \}.$

Als weiteres Beispiel zeigen wir nun, dass $p \rightarrow r$ aus $p \rightarrow q$ und $q \rightarrow r$ folgt. Dazu überführen wir zunächst alle Formeln in Klauseln:

$$\text{knf}(p \rightarrow q) = \{ \{ \neg p, q \} \}, \quad \text{knf}(q \rightarrow r) = \{ \{ \neg q, r \} \}, \quad \text{knf}(p \rightarrow r) = \{ \{ \neg p, r \} \}.$$

Wir haben also $M = \{ \{ \neg p, q \}, \{ \neg q, r \} \}$ und müssen zeigen, dass

$$M \vdash \{ \neg p, r \}$$

folgt. Der Beweis besteht aus einer einzigen Zeile:

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

3.5.1 Eigenschaften des Herleitungs-Begriffs

Die Relation \vdash hat zwei wichtige Eigenschaften:

Satz 15 (Korrektheit) Ist $\{k_1, \dots, k_n\}$ eine Menge von Klauseln und k eine einzelne Klausel, so haben wir:

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

Beweis: Der Beweis verläuft durch eine Induktion nach der Definition der Relation \vdash .

1. Fall: Es gilt $\{k_1, \dots, k_n\} \vdash k$ weil $k \in \{k_1, \dots, k_n\}$ ist. Dann gibt es also ein $i \in \{1, \dots, n\}$, so dass $k = k_i$ ist. In diesem Fall müssen wir

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

zeigen, was ebenfalls offensichtlich ist.

2. Fall: Es gilt $\{k_1, \dots, k_n\} \vdash k$ weil es eine aussagenlogische Variable p und Klauseln g und h gibt, so dass

$$\{k_1, \dots, k_n\} \vdash g \cup \{p\} \quad \text{und} \quad \{k_1, \dots, k_n\} \vdash g \cup \{\neg p\}$$

gilt und daraus haben wir mit der Schnitt-Regel auf

$$\{k_1, \dots, k_n\} \vdash g \cup h$$

geschlossen mit $k = g \cup h$.

Nach Induktions-Voraussetzung haben wir dann

$$\models k_1 \wedge \dots \wedge k_n \rightarrow g \vee p \quad \text{und} \quad \models k_1 \wedge \dots \wedge k_n \rightarrow h \vee \neg p.$$

Wegen

$$\models (g \vee p) \wedge (h \vee \neg p) \rightarrow g \vee h \quad \text{und} \quad k = g \cup h$$

folgt daraus die Behauptung. □

Die Umkehrung dieses Satzes gilt leider nur in abgeschwächter Form und zwar dann, wenn k die leere Klausel ist, also im Fall $k = \perp$.

Satz 16 (Widerlegungs-Vollständigkeit) Ist $\{k_1, \dots, k_n\}$ eine Menge von Klauseln, so haben wir:

$$\text{Wenn } \models k_1 \wedge \dots \wedge k_n \rightarrow \perp \text{ gilt, dann gilt auch } \{k_1, \dots, k_n\} \vdash \{\}.$$

Diesen Satz im Rahmen der Vorlesung zu beweisen würde zuviel Zeit kosten. Ein Beweis findet sich beispielsweise in dem Buch von Schöning [Sch87].

3.6 Das Verfahren von Davis und Putnam

In der Praxis stellt sich oft die Aufgabe, für eine gegebene Menge von Klauseln K eine Belegung \mathcal{I} der Variablen zu berechnen, so dass

$$\text{eval}(k, \mathcal{I}) = \text{true} \quad \text{für alle } k \in K$$

gilt. In diesem Fall sagen wir auch, dass die Belegung \mathcal{I} eine *Lösung* der Klausel-Menge K ist. Wir werden in diesem Abschnitt ein Verfahren vorstellen, mit dem eine solche Aufgabe bearbeitet werden kann. Dieses Verfahren geht auf Davis und Putnam [DLL62] zurück. Verfeinerungen dieses Verfahrens werden beispielsweise eingesetzt, um die Korrektheit digitaler elektronischer Schaltungen nachzuweisen.

Um das Verfahren zu motivieren überlegen wir zunächst, bei welcher Form der Klausel-Menge K unmittelbar klar ist, ob es eine Belegung gibt, die K löst und wie diese Belegung aussieht. Betrachten wir dazu ein Beispiel:

$$K_1 = \{ \{p\}, \{\neg q\}, \{r\}, \{\neg s\}, \{\neg t\} \}$$

Die Klausel-Menge K_1 entspricht der aussagenlogischen Formel

$$p \wedge \neg q \wedge r \wedge \neg s \wedge \neg t.$$

Daher ist K_1 lösbar und die Belegung

$$\mathcal{I} = \{ \langle p, \text{true} \rangle, \langle q, \text{false} \rangle, \langle r, \text{true} \rangle, \langle s, \text{false} \rangle, \langle t, \text{false} \rangle \}$$

ist eine Lösung. Betrachten wir eine weiteres Beispiel:

$$K_2 = \{ \{\}, \{p\}, \{\neg q\}, \{r\} \}$$

Diese Klausel-Menge entspricht der Formel

$$\perp \wedge p \wedge \neg q \wedge r.$$

Offensichtlich ist K_2 unlösbar. Als letztes Beispiel betrachten wir

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

Diese Klausel-Menge kodiert die Formel

$$p \wedge \neg q \wedge \neg p$$

Offenbar ist K_3 ebenfalls unlösbar, denn eine Lösung \mathcal{I} müßte p gleichzeitig wahr und falsch machen. Wir nehmen die an den letzten drei Beispielen gemachten Beobachtungen zum Anlaß für zwei Definitionen.

Definition 17 (Unit-Klausel) Eine Klausel k heißt *Unit-Klausel*, wenn k nur aus einem Literal besteht. Es gilt dann

$$k = \{p\} \quad \text{oder} \quad k = \{\neg p\}$$

für eine Aussage-Variable p .

Definition 18 (Triviale Klausel-Mengen) Eine Klausel-Menge K heißt *trivial* wenn einer der beiden folgenden Fälle vorliegt.

1. K enthält die leere Klausel: $\{\} \in K$.
In diesem Fall ist K offensichtlich unlösbar.
2. K enthält nur Unit-Klausel mit verschiedenen Aussage-Variablen. Bezeichnen wir die Menge der aussagenlogischen Variablen mit \mathcal{P} , so schreibt sich diese Bedingung als

$$\forall k \in K : \text{card}(k) = 1 \quad \text{und} \quad \forall p \in \mathcal{P} : \neg(\{p\} \in K \wedge \{\neg p\} \in K).$$

Dann ist

$$\mathcal{I} = \{ \langle p, \mathbf{true} \rangle \mid \{p\} \in K \} \cup \{ \langle p, \mathbf{false} \rangle \mid \{\neg p\} \in K \}$$

eine Lösung von K .

Wie können wir nun eine Menge von Klauseln so vereinfachen, dass die Menge schließlich nur noch aus Unit-Klauseln besteht? Es gibt drei Möglichkeiten, Klauselmengen zu vereinfachen:

1. Schnitt-Regel
2. Subsumption
3. Fallunterscheidung

Wir betrachten diese Möglichkeiten jetzt der Reihe nach.

3.6.1 Vereinfachung mit der Schnitt-Regel

Eine typische Anwendung der Schnitt-Regel hat die Form:

$$\frac{k_1 \cup \{p\} \quad \{\neg p\} \cup k_2}{k_1 \cup k_2}$$

Die hierbei erzeugte Klausel $k_1 \cup k_2$ wird in der Regel mehr Literale enthalten als die Prämissen $k_1 \cup \{p\}$ und $\{\neg p\} \cup k_2$. Enthält die Klausel $k_1 \cup \{p\}$ insgesamt $m + 1$ Literale und enthält die Klausel $\{\neg p\} \cup k_2$ insgesamt $n + 1$ Literale, so kann die Konklusion $k_1 \cup k_2$ insgesamt $m + n$ Literale enthalten. Natürlich können es auch weniger Literale sein, und zwar dann, wenn es Literale gibt, die sowohl in k_1 als auch in k_2 auftreten. Im allgemeinen ist $m + n$ größer als $m + 1$ und als $n + 1$. Die Klauseln wachsen nur dann sicher nicht, wenn entweder $n = 0$ oder $m = 0$ ist. Dieser Fall liegt vor, wenn einer der beiden Klauseln nur aus einem Literal besteht und folglich eine *Unit-Klausel* ist. Da es unser Ziel ist, die Klausel-Mengen zu vereinfachen, lassen wir nur solche Anwendungen der Schnitt-Regel zu, bei denen eine der Klauseln eine Unit-Klausel ist. Solche Schnitte bezeichnen wir als *Unit-Schnitte*. Um alle mit einer gegebenen Unit-Klausel $\{l\}$ möglichen Schnitte durchführen zu können, definieren wir eine Funktion

$$\mathit{unitCut} : 2^{\mathcal{K}} \times \mathcal{L} \rightarrow 2^{\mathcal{K}}$$

so, dass für eine Klausel-Menge K und eine Unit-Klausel $\{l\}$ die Funktion $\mathit{unitCut}(K, l)$ die Klausel-Menge K soweit wie möglich mit Unit-Schnitten mit der Klausel $\{l\}$ vereinfacht:

$$\mathit{unitCut}(K, l) = \left\{ k \setminus \{\overline{l}\} \mid k \in K \right\}$$

3.6.2 Vereinfachung durch Subsumption

Das Prinzip der Subsumption demonstrieren wir zunächst an einem Beispiel. Wir betrachten

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

Offenbar impliziert die Klausel $\{p\}$ die Klausel $\{p, q, \neg r\}$, immer wenn $\{p\}$ erfüllt ist, ist automatisch auch $\{p, q, \neg r\}$ erfüllt, denn es gilt

$$\models p \rightarrow q \vee p \vee \neg r.$$

Allgemein sagen wir, dass eine Klausel k von einer Unit-Klausel u *subsumiert* wird, wenn

$$u \subseteq k$$

gilt. Ist K eine Klausel-Menge mit $k \in K$ und $u \in K$ und wird k durch u subsumiert, so können

wir K durch Unit-Subsumption zu $K - \{k\}$ vereinfachen, indem wir die Klausel k aus K löschen. Allgemein definieren wir eine Funktion

$$\text{subsume} : 2^{\mathcal{K}} \times \mathcal{L} \rightarrow 2^{\mathcal{K}}$$

die eine gegebene Klauselmenge K , die die Unit-Klausel $\{l\}$ enthält, mittels Subsumption dadurch vereinfacht, dass alle durch $\{l\}$ subsumierten Klauseln aus K gelöscht werden. Die Unit-Klausel $\{l\}$ behalten wir natürlich. Daher definieren wir:

$$\text{subsume}(K, l) := (K \setminus \{k \in K \mid l \in k\}) \cup \{\{l\}\} = \{k \in K \mid l \notin k\} \cup \{\{l\}\}.$$

In der obigen Definition muss $\{l\}$ in das Ergebnis eingefügt werden, weil die Menge $\{k \in K \mid l \notin k\}$ die Unit-Klausel $\{l\}$ nicht enthält.

3.6.3 Vereinfachung durch Fallunterscheidung

Ein Kalkül, der nur mit Unit-Schnitten und Subsumption arbeitet, ist nicht widerlegungs-vollständig. Wir brauchen daher eine weitere Möglichkeit, Klausel-Mengen zu vereinfachen. Eine solche Möglichkeit bietet das Prinzip der *Fallunterscheidung*. Dieses Prinzip basiert auf dem folgenden Satz.

Satz 19 Ist K eine Menge von Klauseln und ist p eine aussagenlogische Variable, so ist K genau dann erfüllbar, wenn $K \cup \{\{p\}\}$ oder $K \cup \{\{\neg p\}\}$ erfüllbar ist.

Beweis: Ist K erfüllbar durch eine Belegung \mathcal{I} , so gibt es für $\mathcal{I}(p)$ zwei Möglichkeiten: Falls $\mathcal{I}(p) = \text{true}$ ist, ist damit auch die Menge $K \cup \{\{p\}\}$ erfüllbar, andernfalls ist $K \cup \{\{\neg p\}\}$ erfüllbar.

Da K sowohl eine Teilmenge von $K \cup \{\{p\}\}$ als auch von $K \cup \{\{\neg p\}\}$ ist, ist klar, dass K erfüllbar ist, wenn eine dieser Mengen erfüllbar sind. \square

Wir können nun eine Menge K von Klauseln dadurch vereinfachen, dass wir eine aussagenlogische Variable p wählen, die in K vorkommt. Anschließend bilden wir die Mengen

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

und untersuchen rekursiv ob K_1 erfüllbar ist. Falls wir eine Lösung für K_1 finden, ist dies auch eine Lösung für die ursprüngliche Klausel-Menge K und wir haben unser Ziel erreicht. Andernfalls untersuchen wir rekursiv ob K_2 erfüllbar ist. Falls wir nun eine Lösung finden, ist dies auch eine Lösung von K und wenn wir für K_2 keine Lösung finden, dann hat auch K keine Lösung. Die rekursive Untersuchung von K_1 bzw. K_2 ist leichter, weil ja wir dort mit den Unit-Klausel $\{p\}$ bzw. $\{\neg p\}$ zunächst Unit-Subsumption und anschließend Unit-Schnitte durchführen können.

3.6.4 Der Algorithmus

Wir können jetzt den Algorithmus von Davis und Putnam skizzieren. Gegeben sei eine Menge K von Klauseln. Gesucht ist dann eine Lösung von K . Wir suchen also eine Belegung \mathcal{I} , so dass gilt:

$$\mathcal{I}(k) = \text{true} \quad \text{für alle } k \in K.$$

Das Verfahren von Davis und Putnam besteht nun aus den folgenden Schritten.

1. Führe alle Unit-Schnitte aus, die mit Klauseln aus K möglich sind und führe zusätzlich alle Unit-Subsumptionen aus.
2. Falls K nun trivial ist, sind wir fertig.
3. Andernfalls wählen wir eine aussagenlogische Variable p , die in K auftritt.

(a) Jetzt versuchen wir rekursiv die Klausel-Menge

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

zu lösen. Falls diese gelingt, haben wir eine Lösung von K .

(b) Andernfalls versuchen wir die Klausel-Menge

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

zu lösen. Wenn auch dies fehlschlägt, ist K unlösbar, andernfalls haben wir eine Lösung von K .

Für die Implementierung ist es zweckmäßig, die beiden oben definierten Funktionen $unitCut()$ und $subsume()$ zusammen zu fassen. Wir definieren eine Funktion

$$reduce : 2^{\mathcal{K}} \times \mathcal{L} \rightarrow 2^{\mathcal{K}}$$

wie folgt:

$$reduce(K, l) = \{k \setminus \{\bar{l}\} \mid k \in K \wedge \bar{l} \in k\} \cup \{k \in K \mid \bar{l} \notin k \wedge l \notin k\} \cup \{\{l\}\}.$$

Die Menge enthält also einerseits die Ergebnisse von Schnitten mit der Unit-Klausel $\{l\}$ und andererseits nur noch die Klauseln k , die mit l nichts zu tun haben weil weder $l \in k$ noch $\bar{l} \in k$ gilt. Außerdem fügen wir auch noch die Unit-Klausel $\{l\}$ hinzu. Dadurch erreichen wir, dass die beiden Mengen K und $reduce(K, l)$ logisch äquivalent sind, wenn wir diese Mengen als Formeln in konjunktiver Normalform interpretieren.

3.6.5 Ein Beispiel

Zur Veranschaulichung demonstrieren wir das Verfahren von Davis und Putnam an einem Beispiel. Die Menge K sei wie folgt definiert:

$$K := \left\{ \{p, q, s\}, \{\neg p, r, \neg t\}, \{r, s\}, \{\neg r, q, \neg p\}, \right. \\ \left. \{\neg s, p\}, \{\neg p, \neg q, s, \neg r\}, \{p, \neg q, s\}, \{\neg r, \neg s\}, \{\neg p, \neg s\} \right\}.$$

Wir zeigen nun mit dem Verfahren von Davis und Putnam, dass K nicht lösbar ist. Da die Menge K keine Unit-Klauseln enthält, ist im ersten Schritt nichts zu tun. Da K nicht trivial ist, sind wir noch nicht fertig. Also gehen wir jetzt zu Schritt 3 und wählen eine aussagenlogische Variable, die in K auftritt. An dieser Stelle ist es sinnvoll eine Variable zu wählen, die in möglichst vielen Klauseln von K auftritt. Wir wählen daher die aussagenlogische Variable p .

1. Zunächst bilden wir jetzt die Menge

$$K_0 := K \cup \{\{p\}\}$$

und versuchen, diese Menge zu lösen. Dazu bilden wir jetzt

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

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

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

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

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

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

$$K_4 := reduce(K_3, q) = \left\{ \{r\}, \{q\}, \{\}, \{\neg s\}, \{p\} \right\}.$$

Die Klausel-Menge K_4 enthält die leere Klausel und ist damit unlösbar.

2. Also bilden wir jetzt die Menge

$$K_5 := K \cup \{\{\neg p\}\}$$

und versuchen, diese Menge zu lösen. Dazu bilden wir

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

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

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

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

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

Da K_8 die leere Klausel enthält, ist K_8 und damit auch die ursprünglich gegebene Menge K unlösbar.

3.6.6 Implementierung des Algorithmus von Davis und Putnam

Wir zeigen jetzt die Implementierung der Prozedur **davisPutnam**, mit der die Frage, ob eine Menge von Klauseln erfüllbar ist, beantwortet werden kann. Die Implementierung ist in Abbildung 3.9 auf Seite 69 gezeigt. Die Prozedur erhält zwei Argumente: **clauses** und **literals**. **clauses** ist eine Menge von Klauseln und **literals** ist eine Menge von Literalen. Falls die Vereinigung dieser beiden Mengen erfüllbar ist, so liefert der Aufruf **davisPutnam(clauses, literals)** eine Menge von Unit-Klauseln **r**, so dass jede Belegung \mathcal{I} , die alle Unit-Klauseln aus **r** erfüllt, auch die Menge **clauses** \cup **literals** erfüllt. Falls die Menge **clauses** \cup **literals** nicht erfüllbar ist, liefert der Aufruf **davisPutnam(Clauses, Literals)** als Ergebnis die Menge $\{\{\}\}$ zurück, denn die leere Klausel repräsentiert die unerfüllbare Formel \perp .

Sie fragen sich vielleicht, wozu wir in der Prozedur **davisPutnam** die Menge **literals** brauchen. Der Grund ist, dass wir uns bei den rekursiven Aufrufen merken müssen, welche Literale wir schon benutzt haben. Diese Literale sammeln wir in der Menge **literals**.

Die in Abbildung 3.9 gezeigte Implementierung funktioniert wie folgt:

1. In Zeile 2 reduzieren wir solange wie möglich die gegebene Klausel-Menge mit Hilfe von Unit-Schnitten und entfernen solche Klauseln die durch Unit-Klauseln subsumiert werden.
2. Anschließend testen wir in Zeile 3, ob die so vereinfachte Klausel-Menge die leere Klausel enthält und geben in diesem Fall als Ergebnis die Menge $\{\{\}\}$ zurück.
3. Dann testen wir in Zeile 6, ob bereits alle Klauseln c aus der Menge **clauses** Unit-Klauseln sind. Wenn dies so ist, dann ist die Menge **clauses** trivial und wir geben diese Menge als Ergebnis zurück.
4. Andernfalls wählen wir in Zeile 9 ein Literal l aus der Menge **clauses**, dass wir noch nicht benutzt haben. Wir untersuchen dann in Zeile 10 rekursiv, ob die Menge

$$\mathbf{clauses} \cup \{\{l\}\}$$

lösbar ist. Dann gibt es zwei Fälle:

- (a) Falls diese Menge lösbar ist, geben wir die Lösung dieser Menge als Ergebnis zurück.
- (b) Sonst prüfen wir rekursiv, ob die Menge

$$\mathbf{clauses} \cup \{\{\neg l\}\}$$

lösbar ist.

```

1  davisPutnam := procedure(clauses, literals) {
2      clauses := saturate(clauses);
3      if ({ } in clauses) {
4          return { { } };
5      }
6      if (forall (c in clauses | #c == 1)) {
7          return clauses;
8      }
9      l := selectLiteral(clauses, literals);
10     r := davisPutnam(clauses + { {l} }, literals + { l });
11     if (r != { { } }) {
12         return r;
13     }
14     notL := negateLiteral(l);
15     return davisPutnam(clauses + { {notL} }, literals + { notL });
16 };

```

Abbildung 3.9: Die Prozedur `davisPutnam`.

Wir diskutieren nun die Hilfsprozeduren, die bei der Implementierung der Prozedur `davisPutnam` verwendet wurden. Als erstes besprechen wir die Funktion `saturate`. Diese Prozedur erhält eine Menge s von Klauseln als Eingabe und führt alle möglichen Unit-Schnitte und Unit-Subsumptionen durch. Die Prozedur `saturate` ist in Abbildung 3.10 auf Seite 69 gezeigt.

```

1  saturate := procedure(s) {
2      units := { k in s | #k == 1 };
3      used := {};
4      while (units != {}) {
5          unit := arb(units);
6          used := used + { unit };
7          l := arb(unit);
8          s := reduce(s, l);
9          units := { k in s | #k == 1 } - used;
10     }
11     return s;
12 };

```

Abbildung 3.10: Die Prozedur `saturate`.

Die Implementierung von `saturate` funktioniert wie folgt:

1. Zunächst berechnen wir in Zeile 2 die Menge `units` aller Unit-Klauseln.
2. Dann initialisieren wir in Zeile 3 die Menge `used` als die leere Menge. In dieser Menge merken wir uns, welche Unit-Klauseln wir schon für Unit-Schnitte und Subsumptionen benutzt haben.
3. Solange die Menge `units` der Unit-Klauseln nicht leer ist, wählen wir in Zeile 5 eine beliebige Unit-Klausel `unit` aus.
4. In Zeile 6 fügen wir die Klausel `unit` zu der Menge `used` der benutzten Klausel hinzu.
5. In Zeile 7 extrahieren mit `arb` das Literal `l` der Klausel `unit`.

6. In Zeile 8 wird die eigentliche Arbeit durch einen Aufruf der Prozedur **reduce** geleistet.
7. Dabei können jetzt neue Unit-Klauseln entstehen, die wir in Zeile 9 aufsammeln. Wir sammeln dort aber nur die Unit-Klauseln auf, die wir noch nicht benutzt haben.
8. Die Schleife in den Zeilen 4 – 10 wird nun solange durchlaufen, wie wir Unit-Klauseln finden, die wir noch nicht benutzt haben.

Die dabei verwendete Prozedur **reduce()** ist in Abbildung 3.11 gezeigt. Im vorigen Abschnitt hatten wir die Funktion $reduce(K, l)$, die eine Klausel-Menge K mit Hilfe des Literals l reduziert, als

$$reduce(K, l) = \{ k \setminus \{\bar{l}\} \mid k \in K \wedge \bar{l} \in k \} \cup \{ k \in K \mid \bar{l} \notin k \wedge l \notin k \} \cup \{\{l\}\}$$

definiert. Die Implementierung setzt diese Definition unmittelbar um.

```

1  reduce := procedure(s, l) {
2      notL := negateLiteral(l);
3      return  { k - { notL } : k in s | notL in k }
4              + { k in s | !notL in k && !l in k }
5              + { {l} };
6  };

```

Abbildung 3.11: Die Prozedur **reduce**.

Die Implementierung des Algorithmus von Davis und Putnam benutzt außer den bisher diskutierten Prozeduren noch zwei weitere Hilfsprozeduren, deren Implementierung in Abbildung 3.12 auf Seite 70 gezeigt wird.

1. Die Prozedur **selectLiteral** wählt ein beliebiges Literal aus einer gegebenen Menge s von Klauseln aus, das außerdem nicht in der Menge **forbidden** von Literalen vorkommen darf, die bereits benutzt worden sind. Dazu werden alle Klauseln, die ja Mengen von Literalen sind, vereinigt. Von dieser Menge wird dann die Menge der bereits benutzten Literalen abgezogen und aus der resultierenden Menge wird mit Hilfe der Funktion **arb()** ein Literal ausgewählt.
2. Die Prozedur **negateLiteral** bildet die Negation \bar{l} eines gegebenen Literals l . Um die Implementierung zu verstehen, müssen Sie sich erinnern, dass ein Literal der Form $\neg p$ als Liste der Form $["-", p]$ dargestellt wird. Falls das Literal l die Form $\neg p$ hat, wird aber statt der Formel $\neg\neg p$ das Literal p zurückgegeben.

```

1  selectLiteral := procedure(s, forbidden) {
2      return arb(+/ s - forbidden);
3  };
4
5  negateLiteral := procedure(l) {
6      if (l(1) == "-") {
7          return l(2);
8      } else {
9          return [ "-", l ];
10     }
11 };

```

Abbildung 3.12: Die Prozeduren **select** und **negateLiteral**.

Die oben dargestellte Version des Verfahrens von Davis und Putnam lässt sich in vielerlei Hinsicht verbessern. Aus Zeitgründen können wir auf solche Verbesserungen leider nicht weiter eingehen. Der interessierte Leser sei hier auf [MMZ⁺01] verwiesen.

Chaff: Engineering an Efficient SAT Solver

von M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik

3.7 Das 8-Damen-Problem

In diesem Abschnitt zeigen wir, wie bestimmte kombinatorische Problem in aussagenlogische Probleme umformuliert werden können. Diese können dann anschließend mit dem Algorithmus von Davis und Putnam bzw. mit Verbesserungen dieses Algorithmus gelöst werden. Als konkretes Beispiel betrachten wir das 8-Damen-Problem. Dabei geht es darum, 8 Damen so auf einem Schach-Brett aufzustellen, dass keine Dame eine andere Dame schlagen kann. Beim Schach-Spiel kann eine Dame dann eine andere Figur schlagen falls diese Figur entweder

- in der selben Zeile,
- in der selben Spalte, oder
- in der selben Diagonale

steht. Abbildung 3.13 auf Seite 71 zeigt ein Schachbrett, in dem sich in der dritten Zeile in der vierten Spalte eine Dame befindet. Diese Dame kann auf alle die Felder ziehen, die mit Pfeilen markierte sind, und kann damit Figuren, die sich auf diesen Feldern befinden, schlagen.

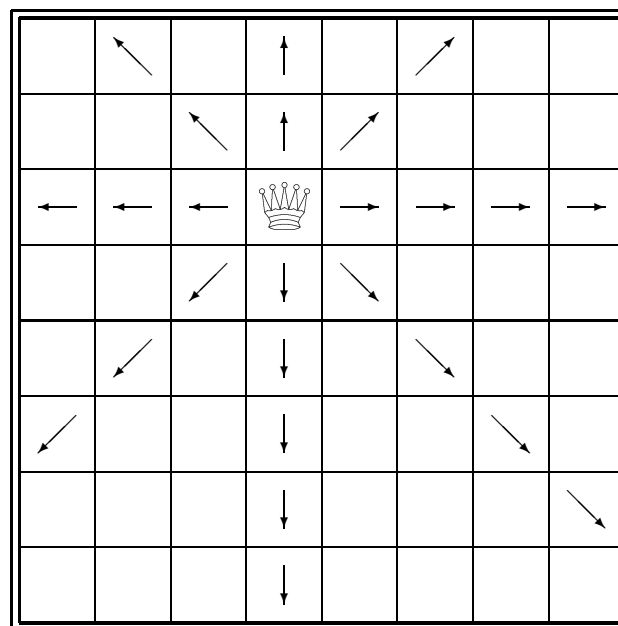


Abbildung 3.13: Das 8-Damen-Problem.

Als erstes überlegen wir uns, wie wir ein Schach-Brett mit den darauf positionierten Damen aussagenlogisch repräsentieren können. Eine Möglichkeit besteht darin, für jedes Feld eine aussagenlogische Variable einzuführen. Diese Variable drückt aus, dass auf dem entsprechenden Feld eine Dame steht. Wir ordnen diesen Variablen wie folgt Namen zu: Die Variable, die das j -te Feld in der i -ten Zeile bezeichnet, erhält den Namen

$$p_{ij} \quad \text{mit } i, j \in \{1, \dots, 8\}.$$

Wir numerieren die Zeilen dabei von oben beginnend von 1 bis 8 durch, während die Spalten von links nach rechts numeriert werden. Abbildung 3.14 auf Seite 72 zeigt die Zuordnung der Variablen zu den Feldern.

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

Abbildung 3.14: Zuordnung der Variablen.

Um die obigen Überlegungen in *SetIX* umzusetzen, implementieren wir die Prozedur `createBoard()`, die eine Liste von Listen von Variablen berechnet, die das oben gezeigte Schach-Brett repräsentieren. Abbildung 3.15 auf Seite 72 zeigt die Implementierung.

```

1  createBoard := procedure(n) {
2      return [ [ "p" + i + j : j in [1..n] ] : i in [1..n] ];
3  };

```

Abbildung 3.15: Die Prozeduren `createBoard` und `createRow`.

Um zu verstehen, wie diese Prozedur funktioniert, rufen wir sie mit dem Parameter 8 auf um die Repräsentation eines Schach-Bretts zu erzeugen. Wir erhalten dann das folgende Ergebnis:

```

[ ["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"] ]

```

Wir sehen, dass das Brett als eine Liste dargestellt wird. Diese Liste enthält 8 Elemente, die ihrerseits Listen von Variablen sind und jeweils eine Zeile des Schach-Bretts darstellen.

Als nächstes überlegen wir uns, wie wir die einzelnen Bedingungen des 8-Damen-Problems als aussagenlogische Formeln kodieren können. Letztlich lassen sich alle Aussagen der Form

- “In einer Zeile steht höchstens eine Dame”,

- “In einer Spalte steht höchstens eine Dame”, oder
- “In einer Diagonale steht höchstens eine Dame”

auf das selbe Grundmuster zurückführen: Ist eine Menge von aussagenlogischen Variablen

$$V = \{x_1, \dots, x_n\}$$

gegeben, so brauchen wir eine Formel die aussagt, dass **höchstens** eine der Variablen aus V den Wert **true** hat. Das ist aber gleichbedeutend damit, dass für jedes Paar $x_i, x_j \in V$ mit $x_i \neq x_j$ die folgende Formel gilt:

$$\neg(x_i \wedge x_j).$$

Diese Formel drückt aus, dass die Variablen x_i und x_j nicht gleichzeitig den Wert **true** annehmen. Nach den DeMorgan’schen Gesetzen gilt

$$\neg(x_i \wedge x_j) \leftrightarrow \neg x_i \vee \neg x_j$$

und die Formel auf der rechten Seite dieser Äquivalenz schreibt sich in Mengen-Schreibweise als

$$\{\neg x_i, \neg x_j\}.$$

Die Formel, die für eine Variablen-Menge V ausdrückt, dass keine zwei verschiedenen Variablen gleichzeitig gesetzt sind, kann daher als Klausel-Menge wie folgt geschrieben werden:

$$\{\{\neg p, \neg q\} \mid p \in V \wedge q \in V \wedge p \neq q\}.$$

Wir setzen diese Überlegungen in eine *SetIX*-Prozedur um. Die in Abbildung 3.16 gezeigte Prozedur **atMostOne()** bekommt als Eingabe eine Menge V von aussagenlogischen Variablen. Der Aufruf **atMostOne(V)** berechnet eine Menge von Klauseln. Diese Klauseln sind genau dann wahr, wenn höchstens eine der Variablen aus V den Wert **true** hat.

```

1  atMostOne := procedure(s) {
2      return { { [ "-", p ], [ "-", q ] } : p in s, q in s | p != q };
3  };

```

Abbildung 3.16: Die Prozedur **atMostOne**.

Mit Hilfe der Prozedur **atMostOne** können wir nun die Prozedur **atMostOneInRow** implementieren. Der Aufruf

atMostOneInRow(board, row)

berechnet für ein gegebenes Brett *board* und eine Zahl *row* eine Formel die ausdrückt, dass in der Zeile *row* höchstens eine Dame steht. Dabei wird natürlich vorausgesetzt, das *board* eine Struktur hat, wie wir Sie oben diskutiert haben. Eine solche Struktur können wir mit der Prozedur **createBoard()** erzeugen. Abbildung 3.17 zeigt die Prozedur **atMostOneInRow()**: Wir sammeln alle Variablen der durch *row* spezifizierten Zeile in der Menge

$$\{\text{board}(\text{row})(j) \mid j \in \{1, \dots, 8\}\}$$

auf und rufen mit dieser Menge die Hilfs-Prozedur **atMostOne()** auf, die das Ergebnis als Menge von Klauseln liefert.

Als nächstes berechnen wir eine Formel die aussagt, dass mindestens eine Dame in einer gegebenen Spalte steht. Für die erste Spalte hätte diese Formel die Form

$$p11 \vee p21 \vee p31 \vee p41 \vee p51 \vee p61 \vee p71 \vee p81$$

und wenn allgemein eine Spalte c mit $c \in \{1, \dots, 8\}$ gegeben ist, lautet die Formel

$$p1c \vee p2c \vee p3c \vee p4c \vee p5c \vee p6c \vee p7c \vee p8c.$$

```

1  atMostOneInRow := procedure(board, row) {
2      return atMostOne({ board(row)(j) : j in [1 .. #board] });
3  };

```

Abbildung 3.17: Die Prozedur `atMostOneInRow`.

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

$$\{ \{ p_{1c}, p_{2c}, p_{3c}, p_{4c}, p_{5c}, p_{6c}, p_{7c}, p_{8c} \} \}.$$

Abbildung 3.18 zeigt eine *SetIX*-Prozedur, die für eine gegebene Spalte `column` die entsprechende Klausel-Menge berechnet. Der Schritt, von einer einzelnen Klausel zu einer Menge von Klauseln überzugehen ist notwendig, da unsere Implementierung des Algorithmus von Davis und Putnam ja mit einer Menge von Klauseln arbeitet.

```

1  oneInColumn := procedure(board, column) {
2      return { { board(row)(column) : row in { 1 .. #board } } };
3  };

```

Abbildung 3.18: Die Prozedur `oneInColumn`.

An dieser Stelle erwarten Sie vielleicht, dass wir als noch Formeln angeben die ausdrücken, dass in einer gegebenen Spalte höchstens eine Dame steht und dass in jeder Reihe mindestens eine Dame steht. Solche Formeln sind aber unnötig, denn wenn wir wissen, dass in jeder Spalte mindestens eine Dame steht, so wissen wir bereits, dass auf dem Brett mindestens 8 Damen stehen. Wenn wir nun zusätzlich wissen, dass in jeder Zeile höchstens eine Dame steht, so ist automatisch klar, dass in jeder Zeile genau eine Dame stehen muß, denn sonst kommen wir insgesamt nicht auf 8 Damen. Weiter folgt aus der Tatsache, dass in jeder Spalte eine Dame steht und daraus, dass es insgesamt nicht mehr als 8 Damen sind, dass in jeder Spalte höchstens eine Dame stehen kann.

Als nächstes überlegen wir uns, wie wir die Variablen, die auf der selben Diagonale stehen, charakterisieren können. Es gibt grundsätzlich zwei verschiedene Arten von Diagonalen: absteigende Diagonalen und aufsteigende Diagonalen. Wir betrachten zunächst die aufsteigenden Diagonalen. Die längste aufsteigende Diagonale, wir sagen dazu auch *Hauptdiagonale*, besteht aus den Variablen

$$p_{81}, p_{72}, p_{63}, p_{54}, p_{45}, p_{36}, p_{27}, p_{18}.$$

Die Indizes i und j dieser Variablen p_{ij} erfüllen offenbar die Gleichung

$$i + j = 9.$$

Allgemein erfüllen die Indizes der Variablen einer aufsteigenden Diagonale die Gleichung

$$i + j = k,$$

wobei k einen Wert aus der Menge $\{3, \dots, 15\}$ annimmt. Diesen Wert k geben wir nun als Argument bei der Prozedur `atMostOneInUpperDiagonal` mit. Diese Prozedur ist in Abbildung 3.19 gezeigt.

Um zu sehen, wie die Variablen einer fallenden Diagonale charakterisiert werden können, betrachten wir die fallende Hauptdiagonale, die aus den Variablen

$$p_{11}, p_{22}, p_{33}, p_{44}, p_{55}, p_{66}, p_{77}, p_{88}$$

besteht. Die Indizes dieser Variablen i und j dieser Variablen p_{ij} erfüllen offenbar die Gleichung

$$i - j = 0.$$

```

1  atMostOneInUpperDiagonal := procedure(board, k) {
2      n := #board;
3      s := { board(r)(c) : c in [1..n], r in [1..n] | r + c == k };
4      return atMostOne(s);
5  };

```

Abbildung 3.19: Die Prozedur `atMostOneInUpperDiagonal`.

Allgemein erfüllen die Indizes der Variablen einer absteigenden Diagonale die Gleichung

$$i - j = k,$$

wobei k einen Wert aus der Menge $\{-6, \dots, 6\}$ annimmt. Diesen Wert k geben wir nun als Argument bei der Prozedur `atMostOneInLowerDiagonal` mit. Diese Prozedur ist in Abbildung 3.20 gezeigt.

```

1  atMostOneInLowerDiagonal := procedure(board, k) {
2      n := #board;
3      s := { board(r)(c) : c in [1..n], r in [1..n] | r - c == k };
4      return atMostOne(s);
5  };

```

Abbildung 3.20: Die Prozedur `atMostOneInLowerDiagonal`.

Jetzt sind wir in der Lage, unsere Ergebnisse zusammen zu fassen. Wir können nun eine Menge von Klauseln konstruieren, die das 8-Damen-Problem vollständig beschreibt. Abbildung 3.21 zeigt die Implementierung der Prozedur `allClauses`. Der Aufruf

`allClauses(board)`

rechnet für ein gegebenes Schach-Brett *board* eine Menge von Klauseln aus, die genau dann erfüllt sind, wenn auf dem Schach-Brett

1. in jeder Zeile höchstens eine Dame steht (Zeile 2),
2. in jeder absteigenden Diagonale höchstens eine Dame steht (Zeile 3),
3. in jeder aufsteigenden Diagonale höchstens eine Dame steht (Zeile 4) und
4. in jeder Spalte mindestens eine Dame steht (Zeile 5).

Die Ausdrücke in den einzelnen Zeilen liefern Mengen, deren Elemente Klausel-Mengen sind. Was wir als Ergebnis brauchen ist aber eine Klausel-Menge und keine Menge von Klausel-Mengen. Daher bilden wir mit dem Operator “+” die Vereinigung dieser Mengen.

Als letztes zeigen wir in Abbildung 3.22 die Prozedur `solve`, mit der wir das 8-Damen-Problem lösen können. Hierbei ist `printBoard()` eine Prozedur, welche die Lösung in lesbarere Form als Schachbrett ausdrückt. Das funktioniert allerdings nur, wenn ein Font verwendet wird, bei dem alle Zeichen die selbe Breite haben. Diese Prozedur ist der Vollständigkeit halber in Abbildung 3.23 gezeigt, wir wollen die Implementierung aber nicht weiter diskutieren.

Die durch den Aufruf `davisPutnam(Clauses, {})` berechnete Menge i enthält für jede der Variablen p_{ij} entweder die Unit-Klausel $\{p_{ij}\}$ (falls auf diesem Feld eine Dame steht) oder aber die Unit-Klausel $\{\neg p_{ij}\}$ (falls das Feld leer bleibt). Eine graphische Darstellung des durch die berechnete Belegung dargestellten Schach-Bretts sehen Sie in Abbildung 3.24.

```

1  allClauses := procedure(board) {
2      n := #board;
3      return  +/ { atMostOneInRow(board, row)          : row in {1..n}          }
4              + +/ { atMostOneInLowerDiagonal(board, k) : k in {-(n-2) .. n-2} }
5              + +/ { atMostOneInUpperDiagonal(board, k) : k in {3 .. 2*n - 1} }
6              + +/ { oneInColumn(board, column)         : column in {1 .. n}    };
7  };

```

Abbildung 3.21: Die Prozedur `allClauses`.

```

1  solve := procedure(n) {
2      board      := createBoard(n);
3      clauses    := allClauses(board);
4      solution := davisPutnam(clauses, {});
5      if (solution != { {} }) {
6          printBoard(solution, board);
7      } else {
8          print("The problem is not solvable for " + numberQueens + " queens!");
9          print("Try to increase the number of queens.");
10     }
11 };

```

Abbildung 3.22: Die Prozedur `solve`.

Das 8-Damen-Problem ist natürlich nur eine spielerische Anwendung der Aussagen-Logik. Trotzdem zeigt es die Leistungsfähigkeit des Algorithmus von Davis und Putnam sehr gut, denn die Menge der Klauseln, die von der Prozedur `allClauses` berechnet wird, füllt unformatiert fünf Bildschirm-Seiten, falls diese eine Breite von 80 Zeichen haben. In dieser Klausel-Menge kommen 64 verschiedene Variablen vor. Der Algorithmus von Davis und Putnam benötigt zur Berechnung einer Belegung, die diese Klauseln erfüllt, auf einem herkömmlichen PC weniger als fünf Sekunden³!

In der Praxis gibt es viele Probleme, die sich in ganz ähnlicher Weise auf die Lösung einer Menge von Klauseln zurückführen lassen. Dazu gehört zum Beispiel das Problem, einen Stundenplan zu erstellen, der gewissen Nebenbedingungen genügt. Verallgemeinerungen des Stundenplan-Problems werden in der Literatur als *Scheduling-Problemen* bezeichnet und sind Gegenstand der aktuellen Forschung.

³Das System, auf dem der Test lief, war mit einem auf 2,16 Ghz getakteten Intel Core 2 Duo Prozessor und 1,5 GB Hauptspeicher bestückt.

```

1  printBoard := procedure(i, board) {
2      if (i == { {} }) {
3          return;
4      }
5      n := #board;
6      print( "          " + ((8*n+1) * "-") );
7      for (row in [1..n]) {
8          line := "          |";
9          for (col in [1..n]) {
10             line += "          |";
11         }
12         print(line);
13         line := "          |";
14         for (col in [1..n]) {
15             if ({ board(row)(col) } in i) {
16                 line += "      Q      |";
17             } else {
18                 line += "          |";
19             }
20         }
21         print(line);
22         line := "          |";
23         for (col in [1..n]) {
24             line += "          |";
25         }
26         print(line);
27         print( "          " + ((8*n+1) * "-") );
28     }
29 };

```

Abbildung 3.23: Die Prozedur `printBoard()`.

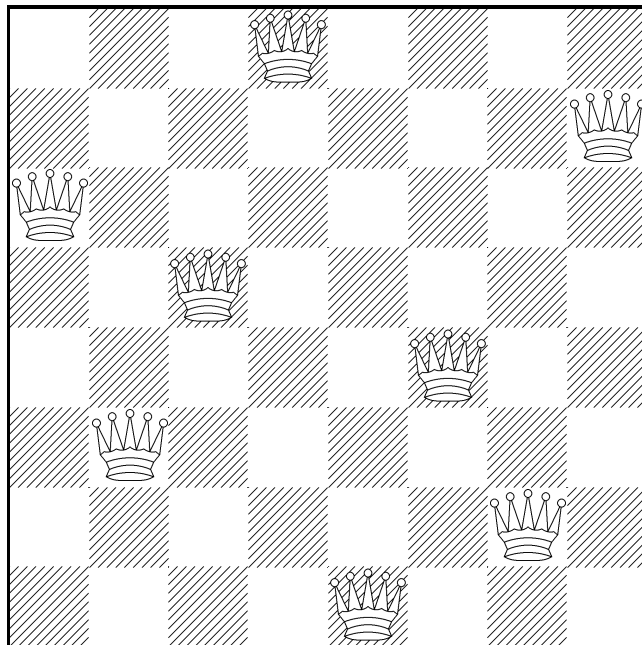


Abbildung 3.24: Eine Lösung des 8-Damen-Problems.

Kapitel 4

Prädikatenlogik

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

1. Als Bezeichnungen für Objekte werden *Terme* verwendet.
2. Diese Terme werden aus *Variablen* und *Funktions-Zeichen* zusammengesetzt:

$$\text{vater}(x), \quad \text{mutter}(\text{isaac}), \quad x + 7, \quad \dots$$

3. Verschiedene Objekte werden durch *Prädikats-Zeichen* in Relation gesetzt:

$$\text{istBruder}(\text{albert}, \text{vater}(\text{bruno})), \quad x + 7 < x \cdot 7, \quad n \in \mathbb{N}, \quad \dots$$

Die dabei entstehenden Formeln werden als *atomare* Formeln bezeichnet.

4. Atomare Formeln lassen sich durch aussagenlogische Junktoren verknüpfen:

$$x > 1 \rightarrow x + 7 < x \cdot 7$$

5. Schließlich werden *Quantoren* eingeführt, um zwischen *existentiell* und *universell* quantifizierten Variablen unterscheiden zu können:

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

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

4.1 Syntax der Prädikatenlogik

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

Definition 20 (Signatur) Eine *Signatur* ist ein 4-Tupel

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

für das folgendes gilt:

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

$$\text{arity} : \mathcal{F} \cup \mathcal{P} \rightarrow \mathbb{N}.$$

Wir sagen, dass das Funktions- oder Prädikats-Zeichen f ein n -stelliges Zeichen ist, falls $\text{arity}(f) = n$ gilt.

5. Da wir in der Lage sein müssen, Variablen, Funktions- und Prädikats-Zeichen unterscheiden zu können, vereinbaren wir, dass die Mengen \mathcal{V} , \mathcal{F} und \mathcal{P} paarweise disjunkt sein müssen:

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

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

Definition 21 (Terme) Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ eine Signatur, so definieren wir die Menge der Σ -Terme \mathcal{T}_Σ induktiv:

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

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

Falls $c \in \mathcal{F}$ 0-stellig ist, lassen wir auch die Schreibweise c anstelle von $c()$ zu. In diesem Fall nennen wir c eine *Konstante*. \square

Zur Veranschaulichung der zuletzt eingeführten Begriffe geben wir ein Beispiel. Es sei die Menge der Variablen $\mathcal{V} = \{x, y, z\}$, die Menge der Funktions-Zeichen $\mathcal{F} = \{0, 1, +, -, \cdot\}$, und die Menge der Prädikats-Zeichen $\mathcal{P} = \{=, \leq\}$ gegeben. Ferner sei die Funktion *arity* als die Relation

$$\text{arity} = \{ \langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle +, 2 \rangle, \langle -, 2 \rangle, \langle *, 2 \rangle, \langle =, 2 \rangle, \langle \leq, 2 \rangle \}$$

definiert. Schließlich sei die Signatur Σ_{arith} durch das 4-Tupel $\langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ gegeben.

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

1. $x, y, z \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
denn alle Variablen sind auch Σ_{arith} -Terme.
2. $0, 1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
denn 0 und 1 sind 0-stellige Funktions-Zeichen.
3. $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
denn es gilt $0 \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $x \in \mathcal{T}_{\Sigma_{\text{arith}}}$ und $+$ ist ein 2-stelliges Funktions-Zeichen.
4. $*+(0, x), 1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
denn $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$ und $*$ ist ein 2-stelliges Funktions-Zeichen.

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

Definition 22 (Atomare Formeln) Gegeben sei eine Signatur $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$. Die Menge der atomaren Σ -Formeln \mathcal{A}_Σ wird wie folgt definiert: Ist $p \in \mathcal{P}$ ein n -stelliges Prädikats-Zeichen und sind n Σ -Terme t_1, \dots, t_n gegeben, so ist $p(t_1, \dots, t_n)$ eine atomare Σ -Formel:

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

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

Setzen wir das obige Beispiel fort, so können wir sehen, dass

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

und diese Formel ist nicht mehr gültig.

In der Prädikatenlogik haben die Quantoren “ \forall ” (*für alle*) und “ \exists ” (*es gibt*) eine ähnliche Bedeutung wie der Integral-Operator “ $\int \dots dt$ ” in der Analysis. Die oben gemachten Ausführungen

zeigen, dass es zwei verschiedene Arten von Variable gibt: *freie Variable* und *gebundene Variable*. Um diese Begriffe präzisieren zu können, definieren wir zunächst für einen Σ -Term t die Menge der in t enthaltenen Variablen.

Definition 23 ($\text{Var}(t)$) Ist t ein Σ -Term, mit $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$, so definieren wir die Menge $\text{Var}(t)$ der Variablen, die in t auftreten, durch Induktion nach dem Aufbau des Terms:

1. $\text{Var}(x) := \{x\}$ für alle $x \in \mathcal{V}$,
2. $\text{Var}(f(t_1, \dots, t_n)) := \text{Var}(t_1) \cup \dots \cup \text{Var}(t_n)$. □

Definition 24 (Σ -Formel, gebundene und freie Variablen)

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

1. Es gilt $\perp \in \mathbb{F}_\Sigma$ und $\top \in \mathbb{F}_\Sigma$ und wir definieren

$$FV(\perp) := FV(\top) := BV(\perp) := BV(\top) := \{\}$$

2. Ist $F = p(t_1, \dots, t_n)$ eine atomare Σ -Formel, so gilt $F \in \mathbb{F}_\Sigma$. Weiter definieren wir:

- (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. Ist $F \in \mathbb{F}_\Sigma$, so gilt $\neg F \in \mathbb{F}_\Sigma$. Weiter definieren wir:

- (a) $FV(\neg F) := FV(F)$.
- (b) $BV(\neg F) := BV(F)$.

4. Sind $F, G \in \mathbb{F}_\Sigma$ und gilt außerdem

$$FV(F) \cap BV(G) = \{\} \quad \text{und} \quad FV(G) \cap BV(F) = \{\},$$

so gilt auch

- (a) $(F \wedge G) \in \mathbb{F}_\Sigma$,
- (b) $(F \vee G) \in \mathbb{F}_\Sigma$,
- (c) $(F \rightarrow G) \in \mathbb{F}_\Sigma$,
- (d) $(F \leftrightarrow G) \in \mathbb{F}_\Sigma$.

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

- (a) $FV(F \odot G) := FV(F) \cup FV(G)$.
- (b) $BV(F \odot G) := BV(F) \cup BV(G)$.

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

- (a) $(\forall x: F) \in \mathbb{F}_\Sigma$.
- (b) $(\exists x: F) \in \mathbb{F}_\Sigma$.

Weiter definieren wir

- (a) $FV((\forall x: F)) := FV((\exists x: F)) := FV(F) \setminus \{x\}$.
- (b) $BV((\forall x: F)) := BV((\exists x: F)) := BV(F) \cup \{x\}$.

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

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

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

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

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

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

Wenn wir Formeln immer in dieser Form anschreiben würden, dann würde die Lesbarkeit unverhältnismäßig leiden. Zur Abkürzung vereinbaren wir, dass in der Prädikatenlogik die selben Regeln zur Klammer-Ersparnis gelten sollen, die wir schon in der Aussagenlogik verwendet haben. Zusätzlich werden gleiche Quantoren zusammengefaßt: Beispielsweise schreiben wir

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

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

$$\forall x: p(x) \wedge G \quad \text{als} \quad (\forall x: p(x)) \wedge G$$

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

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

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

$$\exists x: y + x \leq y$$

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

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

$$(\exists x \in M : F) \stackrel{\text{def}}{\iff} \exists x : (x \in M \wedge F).$$

4.2 Semantik der Prädikatenlogik

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

Definition 25 (Struktur) Es sei eine Signatur

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

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

1. \mathcal{U} ist eine nicht-leere Menge. Diese Menge nennen wir auch das *Universum* der Σ -Struktur. Dieses Universum enthält die Werte, die sich später bei der Auswertung der Terme ergeben werden.
2. \mathcal{J} ist die *Interpretation* der Funktions- und Prädikats-Zeichen. Formal definieren wir \mathcal{J} als eine Abbildung mit folgenden Eigenschaften:

- (a) Jedem Funktions-Zeichen $f \in \mathcal{F}$ mit $\text{arity}(f) = m$ wird eine m -stellige Funktion

$$f^{\mathcal{J}}: \mathcal{U} \times \cdots \times \mathcal{U} \rightarrow \mathcal{U}$$

zugeordnet, die m -Tupel des Universums \mathcal{U} in das Universum \mathcal{U} abbildet.

- (b) Jedem Prädikats-Zeichen $p \in \mathcal{P}$ mit $\text{arity}(p) = n$ wird eine n -stelliges Prädikat

$$p^{\mathcal{J}}: \mathcal{U} \times \cdots \times \mathcal{U} \rightarrow \mathbb{B}$$

zugeordnet, die jedem n -Tupel des Universums \mathcal{U} einen Wahrheitswert aus der Menge $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ zuordnet.

- (c) Ist das Zeichen “=” ein Element der Menge der Prädikats-Zeichen \mathcal{P} , so gilt

$$=^{\mathcal{J}}(u, v) = \mathbf{true} \quad \text{g.d.w.} \quad u = v,$$

das Gleichheits-Zeichen wird also durch die identische Relation $\text{id}_{\mathcal{U}}$ interpretiert.

Beispiel: Wir geben ein Beispiel für eine Σ_{arith} -Struktur $\mathcal{S}_{\text{arith}} = \langle \mathcal{U}_{\text{arith}}, \mathcal{J}_{\text{arith}} \rangle$, indem wir definieren:

1. $\mathcal{U}_{\text{arith}} = \mathbb{N}$.
2. Die Abbildung $\mathcal{J}_{\text{arith}}$ legen wir dadurch fest, dass die Funktions-Zeichen $0, 1, +, -, \cdot$ durch die entsprechend benannten Funktionen auf der Menge \mathbb{N} der natürlichen Zahlen zu interpretieren sind.
Ebenso sollen die Prädikats-Zeichen $=$ und \leq durch die Gleichheits-Relation und die Kleiner-Gleich-Relation interpretiert werden.

Beispiel: Wir geben ein weiteres Beispiel. Die Signatur Σ_G der Gruppen-Theorie sei definiert als

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

1. $\mathcal{V} := \{x, y, z\}$
2. $\mathcal{F} := \{1, *\}$
3. $\mathcal{P} := \{=\}$
4. $\text{arity} = \{ \langle 1, 0 \rangle, \langle *, 2 \rangle, \langle =, 2 \rangle \}$

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

1. $1^{\mathcal{J}} := a$
2. $*^{\mathcal{J}} := \left\{ \langle \langle a, a \rangle, a \rangle, \langle \langle a, b \rangle, b \rangle, \langle \langle b, a \rangle, b \rangle, \langle \langle b, b \rangle, a \rangle \right\}$
3. $=^{\mathcal{J}}$ ist die Identität:

$$=^{\mathcal{J}} := \left\{ \langle \langle a, a \rangle, \mathbf{true} \rangle, \langle \langle a, b \rangle, \mathbf{false} \rangle, \langle \langle b, a \rangle, \mathbf{false} \rangle, \langle \langle b, b \rangle, \mathbf{true} \rangle \right\}$$

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

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

Definition 26 (Variablen-Belegung) Es sei eine Signatur

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

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

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

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

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

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

Definition 27 (Semantik der Terme) Ist $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ eine Σ -Struktur und \mathcal{I} eine \mathcal{S} -Variablen-Belegung, so definieren wir für jeden Term t den Wert $\mathcal{S}(\mathcal{I}, t)$ durch Induktion über den Aufbau von t :

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

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

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

$$\mathcal{S}(\mathcal{I}, f(t_1, \dots, t_n)) := f^{\mathcal{J}}(\mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n)). \quad \square$$

Beispiel: Mit der oben definierten Σ_{arith} -Struktur $\mathcal{S}_{\text{arith}}$ definieren wir eine $\mathcal{S}_{\text{arith}}$ -Variablen-Belegung \mathcal{I} durch

$$\mathcal{I} := \{ \langle x, 0 \rangle, \langle y, 7 \rangle, \langle z, 42 \rangle \},$$

es gilt also

$$\mathcal{I}(x) := 0, \quad \mathcal{I}(y) := 7, \quad \text{und} \quad \mathcal{I}(z) := 42.$$

Dann gilt offenbar

$$\mathcal{S}(\mathcal{I}, x + y) = 7.$$

Definition 28 (Semantik der atomaren Σ -Formeln) Ist \mathcal{S} eine Σ -Struktur und \mathcal{I} eine \mathcal{S} -Variablen-Belegung, so definieren wir für jede atomare Σ -Formel $p(t_1, \dots, t_n)$ den Wert $\mathcal{S}(\mathcal{I}, p(t_1, \dots, t_n))$ wie folgt:

$$\mathcal{S}(\mathcal{I}, p(t_1, \dots, t_n)) := p^{\mathcal{J}}(\mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n)). \quad \square$$

Beispiel: In Fortführung des obigen Beispiels gilt:

$$\mathcal{S}(\mathcal{I}, z \leq x + y) = \text{false}.$$

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

1. $\neg : \mathbb{B} \rightarrow \mathbb{B}$,
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. $\ominus : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$.

Die Semantik dieser Funktionen hatten wir durch die Tabelle in Abbildung 3.1 auf Seite 40 gegeben.

Definition 29 (Semantik der Σ -Formeln) Ist \mathcal{S} eine Σ -Struktur und \mathcal{I} eine \mathcal{S} -Variablen-Belegung, so definieren wir für jede Σ -Formel F den Wert $\mathcal{S}(\mathcal{I}, F)$ durch Induktion über den Aufbau von F :

1. $\mathcal{S}(\mathcal{I}, \top) := \text{true}$ und $\mathcal{S}(\mathcal{I}, \perp) := \text{false}$.
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. $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) := \bigoplus(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
7. $\mathcal{S}(\mathcal{I}, \forall x: F) := \begin{cases} \text{true} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \text{true} \text{ für alle } c \in \mathcal{U} \text{ gilt;} \\ \text{false} & \text{sonst.} \end{cases}$
8. $\mathcal{S}(\mathcal{I}, \exists x: F) := \begin{cases} \text{true} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \text{true} \text{ für ein } c \in \mathcal{U} \text{ gilt;} \\ \text{false} & \text{sonst.} \end{cases} \quad \square$

Beispiel: In Fortführung des obigen Beispiels gilt

$$\mathcal{S}(\mathcal{I}, \forall x: x * 0 < 1) = \text{true}.$$

Definition 30 (Allgemeingültig) Ist F eine Σ -Formel, so dass für jede Σ -Struktur \mathcal{S} und für jede \mathcal{S} -Variablen-Belegung \mathcal{I}

$$\mathcal{S}(\mathcal{I}, F) = \text{true}$$

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

$$\models F. \quad \square$$

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

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

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

Definition 31 (Äquivalent) Zwei Formeln F und G heißen *äquivalent* g.d.w. gilt

$$\models F \leftrightarrow G \quad \square$$

Alle aussagenlogischen Äquivalenzen sind auch prädikatenlogische Äquivalenzen.

Definition 32 (Erfüllbar) Eine Menge $M \subseteq \mathbb{F}_\Sigma$ ist genau dann *erfüllbar*, wenn es eine Struktur \mathcal{S} und eine Variablen-Belegung \mathcal{I} gibt, so dass

$$\forall m \in M : \mathcal{S}(\mathcal{I}, m) = \text{true}$$

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

$$M \models \perp \quad \square$$

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

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

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

$$M \not\models \perp$$

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

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

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

Bisher sind alle bekannten perfekten Zahlen durch 2 teilbar. Die Frage, ob es auch ungerade Zahlen gibt, die perfekt sind, ist ein offenes mathematisches Problem. Um dieses Problem zu lösen könnten wir eine Programm schreiben, dass der Reihe nach für alle ungerade Zahlen überprüft, ob die Zahl perfekt ist. Abbildung 4.1 auf Seite 88 zeigt ein solches Programm. Wenn es eine ungerade perfekte Zahl gibt, dann wird dieses Programm diese Zahl auch irgendwann finden. Wenn es aber keine ungerade perfekte Zahl gibt, dann wird das Programm bis zum St. Nimmerleinstag rechnen und wir werden nie mit Sicherheit wissen, dass es keine ungeraden perfekten Zahlen gibt.

In den nächsten Abschnitten gehen wir daran, den oben erwähnten Kalkül \vdash zu definieren. Es zeigt sich, dass die Arbeit wesentlich einfacher wird, wenn wir uns auf bestimmte Formeln, sogenannte *Klauseln*, beschränken. Wir zeigen daher zunächst im nächsten Abschnitt, dass jede Formel-Menge M so in eine Menge von Klauseln K transformiert werden kann, dass M genau dann erfüllbar ist, wenn K erfüllbar ist. Daher ist die Beschränkung auf Klauseln keine echte Einschränkung.

4.2.1 Implementierung prädikatenlogischer Strukturen in SetlX

Der im letzten Abschnitt präsentierte Begriff einer prädikatenlogischen Struktur erscheint zunächst sehr abstrakt. Wir wollen in diesem Abschnitt zeigen, dass sich dieser Begriff in einfacher Weise in *SetlX* implementieren lässt. Dadurch gelingt es, diesen Begriff zu veranschaulichen. Als konkretes Beispiel wollen wir Strukturen zu Gruppen-Theorie betrachten. Die Signatur Σ_G der Gruppen-Theorie war im letzten Abschnitt durch die Definition

$$\Sigma_G = \langle \{x, y, z\}, \{1, *\}, \{=\}, \{\langle 1, 0 \rangle, \langle *, 2 \rangle, \langle =, 2 \rangle\} \rangle$$

gegeben worden. Hierbei ist also “1” ein 0-stelliges Funktions-Zeichen, “*” ist eine 2-stelliges Funktions-Zeichen und “=” ist ein 2-stelliges Prädikats-Zeichen. Wir hatten bereits eine Struktur \mathcal{S} angegeben, deren Universum aus der Menge $\{a, b\}$ besteht. In *SetlX* können wir diese Struktur durch den in Abbildung 4.2 gezeigten Code implementieren.

```

1  perfect := procedure(n) {q
2      return +/ { x in {1 .. n-1} | n % x == 0 } == n;
3  };
4
5  findPerfect := procedure() {
6      n := 1;
7      while (true) {
8          if (perfect(n)) {
9              if (n % 2 == 0) {
10                 print(n);
11             } else {
12                 print("Heureka: Odd perfect number $n$ found!");
13             }
14         }
15         n := n + 1;
16     }
17 };
18
19 findPerfect();

```

Abbildung 4.1: Suche nach einer ungeraden perfekten Zahl.

```

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 := { [ "e", a ], [ "*", multiply ], [ "=", equal ] };
7  s := [ u, j ];
8  // i is a variable assignment.
9  i := { [ "x", a ], [ "y", b ], [ "z", a ] };

```

Abbildung 4.2: Implementierung einer Struktur zur Gruppen-Theorie

1. Zur Abkürzung haben wir in den Zeile 1 und 2 die Variablen a und b als die Strings "a" und "b" definiert. Dadurch können wir weiter unten die Interpretation des Funktions-Zeichens "*" kürzer angeben.
2. Das in Zeile 3 definierte Universum u besteht aus den beiden Strings "a" und "b".
3. In Zeile 4 definieren wir eine Funktion `multiply` als binäre Relation. Für die so definierte Funktion gilt

$$\begin{aligned}
 \text{multiply}(\langle "a", "a" \rangle) &= "a", & \text{multiply}(\langle "a", "b" \rangle) &= "b", \\
 \text{multiply}(\langle "b", "a" \rangle) &= "b", & \text{multiply}(\langle "b", "b" \rangle) &= "a".
 \end{aligned}$$

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

4. Ebenso haben wir in Zeile 5 die Interpretation $=^{\mathcal{J}}$ des Prädikats-Zeichens "=" als binäre Relation dargestellt.
5. In Zeile 6 fassen wir die einzelnen Interpretationen zu der Relation j zusammen, so dass für ein Funktions-Zeichen f die Interpretation $f^{\mathcal{J}}$ durch den Wert $j(f)$ gegeben ist.

6. Die Interpretation j wird dann in Zeile 7 mit dem Universum u zu der Struktur s zusammen gefasst.
7. Schließlich zeigt Zeile 9, dass eine Variablen-Belegung ebenfalls als Relation dargestellt werden kann. Die erste Komponente der Paare, aus denen diese Relation besteht, sind die Variablen. Die zweite Komponente ist ein Wert aus dem Universum.

```

1  evalFormula := procedure(f, s, i) {
2      switch {
3          case f == 1      : return true;
4          case f == 0      : return false;
5          case f(1) == "!" : return !evalFormula(f(2), s, i);
6          case f(2) == "&&":
7              return evalFormula(f(1), s, i) && evalFormula(f(3), s, i);
8          case f(2) == "||":
9              return evalFormula(f(1), s, i) || evalFormula(f(3), s, i);
10         case f(2) == "->" :
11             return !evalFormula(f(1), s, i) || evalFormula(f(3), s, i);
12         case f(2) == "<->" :
13             return evalFormula(f(1), s, i) == evalFormula(f(3), s, i);
14         case 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         case 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         default : return evalAtomic(f, s, i); // atomic formula
25     }
26 };

```

Abbildung 4.3: Auswertung prädikatenlogischer Formeln

Als nächstes überlegen wir uns, wie wir prädikatenlogische Formeln in einer solchen Struktur auswerten können. Abbildung 4.3 zeigt die Implementierung der Prozedur $evalFormula(f, S, I)$, der als Argumente eine prädikatenlogische Formel f , eine Struktur S und eine Variablen-Belegung I übergeben werden. Die Formel wird dabei als Liste dargestellt, ganz ähnlich, wie wir das bei der Implementierung der Aussagenlogik schon praktiziert haben. Beispielsweise können wir die Formel

$$\forall x : \forall y : x * y = y * x$$

durch die Liste

`["forall", "x", ["forall", "y", ["=", ["*", "x", "y"], ["*", "y", "x"]]]]`

darstellen. Die Auswertung einer solchen Formel ist nun analog zu der in Abbildung 3.1 auf Seite 43 gezeigten Auswertung aussagenlogischer Formeln. Neu ist nur die Behandlung der Quantoren. In den Zeilen 14 bis 16 behandeln wir die Auswertung allquantifizierter Formeln. Ist f eine Formel der Form $\forall x: g$, so wird die Formel f durch die Liste

$$f = [\text{forall}, x, g]$$

dargestellt. Die zweite Komponente dieser Liste ist die Variable x , es gilt also $x = f(2)$, die dritte Komponente ist die Formel g , also gilt $g = f(3)$. Die Auswertung von $\forall x: g$ geschieht nach der Formel

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

Um die Auswertung implementieren zu können, verwenden wir eine Prozedur *modify()*, welche die Variablen-Belegung i an der Stelle x zu c abändert, es gilt also

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

Die Implementierung dieser Prozedur wird später in Abbildung 4.4 gezeigt. Bei der Auswertung eines All-Quantors können wir ausnutzen, dass die Sprache *SetIX* selber den Quantor *forall* unterstützt. Wir können also direkt testen, ob die Formel für alle möglichen Werte c , die wir für die Variable x einsetzen können, richtig ist. Die Auswertung eines Existenz-Quantors ist analog zur Auswertung eines All-Quantors.

```

1  evalAtomic := procedure(a, s, i) {
2      j      := s(2);
3      p      := a(1); // predicate symbol
4      pJ     := j(p);
5      args   := a(2..);
6      argsVal := evalTermList(args, s, i);
7      return argsVal in pJ;
8  };
9
10 evalTerm := procedure(t, s, i) {
11     if (isString(t)) {
12         return i(t);
13     }
14     j      := s(2);
15     f      := t(1); // function symbol
16     fJ     := j(f);
17     args   := t(2..);
18     argsVal := evalTermList(args, s, i);
19     if (#argsVal > 0) {
20         result := fJ(argsVal);
21     } else {
22         result := fJ; // t is a constant
23     }
24     return result;
25 };
26
27 evalTermList := procedure(tl, s, i) {
28     return [ evalTerm(t, s, i) : t in tl ];
29 };
30
31 modify := procedure(i, x, c) {
32     i(x) := c;
33     return i;
34 };

```

Abbildung 4.4: Auswertung von Termen und atomaren Formeln.

Abbildung 4.4 zeigt die Auswertung atomarer Formeln und prädikatenlogischer Terme. Um eine atomare Formel der Form

$$a = [p, t_1, \dots, t_n]$$

auszuwerten, verschaffen wir uns in Zeile 4 zunächst die dem Prädikats-Zeichen p in der Struktur S zugeordnete Menge pJ . Anschließend werden wir die Argumente t_1, \dots, t_n aus und überprüfen dann, ob das Ergebnis dieser Auswertung tatsächlich ein Element der Menge pJ ist.

Die Prozedur `evalTerm()` arbeitet wie folgt: Das erste Argument t der Prozedur `evalTerm(t, S, I)` ist der auszuwertende Term. Das zweite Argument S ist eine prädikatenlogische Struktur und das dritte Argument I ist eine Variablen-Belegung.

1. Falls t eine Variable ist, wobei Variablen durch Strings dargestellt werden, so geben wir in Zeile 3 einfach den Wert zurück, der in der Variablen-Belegung I für diese Variable eingetragen ist. Die Variablen-Belegung wird dabei durch eine zweistellige Relation dargestellt, die wir als Funktion benutzen.
2. Falls der auszuwertende Term t die Form

$$t = f(t_1, \dots, t_n)$$

hat, so wird dieser Term durch eine Liste der Form

$$[f, t_1, \dots, t_n]$$

dargestellt. Um einen solchen Term auszuwerten, werden in Zeile 18 zunächst rekursiv die Subterme t_1, \dots, t_n ausgewertet. Anschließend wird die Interpretation $f^{\mathcal{I}}$ des Funktions-Zeichens f herangezogen, um die Funktion f für die gegebenen Argumente auszuwerten, wobei in Zeile 20 der Fall betrachtet wird, dass tatsächlich Argumente vorhanden sind, während in Zeile 22 der Fall behandelt wird, dass es sich bei der Funktion f um eine Konstante handelt, deren Wert dann unmittelbar durch $f^{\mathcal{I}}$ gegeben ist.

Die Implementierung der Prozedur `evalTermList()` wendet die Funktion `evalTerm()` auf alle Terme der gegebenen Liste an. Bei der Implementierung der in Zeile 31 gezeigten Prozedur `modify(I, x, c)`, die als Ergebnis die Variablen-Belegung $I[x/c]$ berechnet, nutzen wir aus, dass wir bei einer Funktion, die als binäre Relation gespeichert ist, den Wert, der in dieser Relation für ein Argument x eingetragen ist, durch eine Zuweisung der Form $I(x) := c$ abändern können.

```

1  g1 := parse("forall x: =(*(x, e), x)");
2  g2 := parse("forall x: exists y: =(*(x, y), e)");
3  g3 := parse("forall x: forall y: forall z: =( *(*(x, y), z), *(x, *(y, z)) )");
4  gt := { g1, g2, g3, g4 };
5
6  print("checking group theory in the structure \n", s);
7  for (f in gt) {
8      print( "checking ", f, ": ", evalFormula(f, s, i) );
9  }

```

Abbildung 4.5: Axiome der Gruppen-Theorie

Wir zeigen nun, wie sich die in Abbildung 4.3 gezeigte Funktion `evalFormula(f, S, I)` benutzen lässt um zu überprüfen, ob die in Abbildung 4.2 gezeigte Struktur die Axiome der *Gruppen-Theorie* erfüllt. Die Axiome der Gruppen-Theorie sind wie folgt:

1. Die Konstante 1 ist das rechts-neutrale Element der Multiplikation:

$$\forall x: x * 1 = x.$$

2. Für jedes Element x gibt es ein rechts-inverses Element y , dass mit dem Element x multipliziert die 1 ergibt:

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

3. Es gilt das Assoziativ-Gesetz:

$$\forall x: \forall y: \forall z: (x * y) * z = x * (y * z).$$

Diese Axiome sind in den Zeilen 1 bis 3 der Abbildung 4.5 wiedergegeben. Da der Parser zur Erkennung prädikatenlogischer Formeln keine Infix-Notation verarbeiten kann, sind die Formeln in Präfix-Notation angegeben worden. Die Schleife in den Zeilen 7 bis 9 überprüft schließlich, ob die Formeln in der oben definierten Struktur erfüllt sind.

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

4.3 Normalformen für prädikatenlogische Formeln

In diesem Abschnitt werden wir verschiedenen Möglichkeiten zur Umformung prädikatenlogischer Formeln kennenlernen. Zunächst geben wir einige Äquivalenzen an, mit deren Hilfe Quantoren manipuliert werden können.

Satz 33 Es gelten die folgenden Äquivalenzen:

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. Falls x eine Variable ist, für die $x \notin FV(f)$ ist, so haben wir

$$\models (\forall x: f) \leftrightarrow f \quad \text{und} \quad \models (\exists x: f) \leftrightarrow f.$$
8. Falls x eine Variable ist, für die $x \notin FV(g) \cup BV(g)$ gilt, so haben wir die folgenden Äquivalenzen:
 - (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) $\models g \wedge (\exists x: f) \leftrightarrow \exists x: (g \wedge f)$

Um die Äquivalenzen der letzten Gruppe anwenden zu können, ist es notwendig, gebundene Variablen umzubenennen. Ist f eine prädikatenlogische Formel und sind x und y zwei Variablen, so bezeichnet $f[x/y]$ die Formel, die aus f dadurch entsteht, dass jedes Auftreten der Variablen x in f durch y ersetzt wird. Beispielsweise gilt

$$(\forall u: \exists v: p(u, v))[u/z] = \forall z: \exists v: p(z, v)$$

Damit können wir eine letzte Äquivalenz angeben: Ist f eine prädikatenlogische Formel, ist $x \in BV(f)$ und ist y eine Variable, die in f nicht auftritt, so gilt

$$\models f \leftrightarrow f[x/y].$$

Mit Hilfe der oben stehenden Äquivalenzen können wir eine Formel so umformen, dass die Quantoren nur noch außen stehen. Eine solche Formel ist dann in *pränexer Normalform*. Wir führen das Verfahren an einem Beispiel vor: Wir zeigen, dass die Formel

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

allgemeingültig ist:

$$\begin{aligned} & (\forall x: p(x)) \rightarrow (\exists x: p(x)) \\ \leftrightarrow & \neg(\forall x: p(x)) \vee (\exists x: p(x)) \\ \leftrightarrow & (\exists x: \neg p(x)) \vee (\exists x: p(x)) \\ \leftrightarrow & \exists x: (\neg p(x) \vee p(x)) \\ \leftrightarrow & \exists x: \top \\ \leftrightarrow & \top \end{aligned}$$

Um Formeln noch stärker normalisieren zu können, führen wir einen weiteren Äquivalenz-Begriff ein. Diesen Begriff wollen wir vorher durch ein Beispiel motivieren. Wir betrachten die

beiden Formeln

$$f_1 = \forall x: \exists y: p(x, y) \quad \text{und} \quad f_2 = \forall x: p(x, s(x)).$$

Die beiden Formeln f_1 und f_2 sind nicht äquivalent, denn sie entstammen noch nicht einmal der gleichen Signatur: In der Formel f_2 wird das Funktions-Zeichen s verwendet, das in der Formel f_1 überhaupt nicht auftritt. Auch wenn die beiden Formeln f_1 und f_2 nicht äquivalent sind, so besteht zwischen ihnen doch die folgende Beziehung: Ist S_1 eine prädikatenlogische Struktur, in der die Formel f_1 gilt:

$$S_1 \models f_1,$$

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

$$S_2 \models f_2.$$

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

Definition 34 (Skolemisierung) Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ eine Signatur. Ferner sei f eine geschlossene Σ -Formel der Form

$$f = \forall x_1, \dots, x_n: \exists y: g(x_1, \dots, x_n, y).$$

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

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

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

$$f' := \text{Skolem}(f) := \forall x_1: \dots \forall x_n: g(x_1, \dots, x_n, s(x_1, \dots, x_n))$$

Wir lassen also den Existenz-Quantor $\exists y$ weg und ersetzen jedes Auftreten der Variable y durch den Term $s(x_1, \dots, x_n)$. Wir sagen, dass die Formel f' aus der Formel f durch einen Skolemisierungsschritt hervorgegangen ist. \square

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

Definition 35 (Erfüllbarkeits-Äquivalenz) Zwei geschlossene Formeln f und g heißen *erfüllbarkeits-äquivalent* falls f und g entweder beide erfüllbar oder beide unerfüllbar sind. Wenn f und g erfüllbarkeits-äquivalent sind, so schreiben wir

$$f \approx_e g.$$

\square

Satz 36 Falls die Formel f' aus der Formel f durch einen Skolemisierungsschritt hervorgegangen ist, so sind f und f' erfüllbarkeits-äquivalent.

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

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

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

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

Dabei sind die k_i Disjunktionen von *Literalen*. (In der Prädikatenlogik ist ein Literal entweder eine atomare Formel oder die Negation einer atomaren Formel.) Wenden wir hier die Äquivalenz $(\forall x: f_1 \wedge f_2) \leftrightarrow (\forall x: f_1) \wedge (\forall x: f_2)$ an, so können wir die All-Quantoren auf die einzelnen k_i verteilen und die resultierende Formel hat die Gestalt

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

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

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

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

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

$$\models f$$

gilt. Wir wissen, dass

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

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

$$\neg f \approx_e k_1 \wedge \dots \wedge k_n.$$

Dabei sind k_1, \dots, k_n prädikatenlogische Klauseln. Anschließend versuchen wir, aus den Klauseln k_1, \dots, k_n einen Widerspruch herzuleiten:

$$\{k_1, \dots, k_n\} \vdash \perp$$

Wenn dies gelingt wissen wir, dass die Menge $\{k_1, \dots, k_n\}$ unerfüllbar ist. Dann ist auch $\neg f$ unerfüllbar und damit ist dann f allgemeingültig. Damit wir aus den Klauseln k_1, \dots, k_n einen Widerspruch herleiten können, brauchen wir natürlich noch einen Kalkül, der mit prädikatenlogischen Klauseln arbeitet. Einen solchen Kalkül werden wir am Ende dieses Kapitel vorstellen.

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

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

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

$$\left\{ \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \right\} \models \perp$$

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

$$\begin{aligned}
& \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \\
\leftrightarrow & \neg \left(\neg (\exists x: \forall y: p(x, y)) \vee (\forall y: \exists x: p(x, y)) \right) \\
\leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge \neg (\forall y: \exists x: p(x, y)) \\
\leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists y: \neg \exists x: p(x, y)) \\
\leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists y: \forall x: \neg p(x, y))
\end{aligned}$$

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

$$\begin{aligned}
& (\exists x: \forall y: p(x, y)) \wedge (\exists y: \forall x: \neg p(x, y)) \\
\leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists v: \forall u: \neg p(u, v)) \\
\leftrightarrow & \exists v: \left((\exists x: \forall y: p(x, y)) \wedge (\forall u: \neg p(u, v)) \right) \\
\leftrightarrow & \exists v: \exists x: \left((\forall y: p(x, y)) \wedge (\forall u: \neg p(u, v)) \right) \\
\leftrightarrow & \exists v: \exists x: \forall y: \left(p(x, y) \wedge (\forall u: \neg p(u, v)) \right) \\
\leftrightarrow & \exists v: \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, v) \right)
\end{aligned}$$

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

$$\begin{aligned}
& \exists v: \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, v) \right) \\
\approx_e & \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, s_1) \right) \\
\approx_e & \forall y: \forall u: \left(p(s_2, y) \wedge \neg p(u, s_1) \right)
\end{aligned}$$

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

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

Wir zeigen nun, dass die Menge M widersprüchlich ist. Dazu betrachten wir zunächst die Klausel $\{p(s_2, y)\}$ und setzen in dieser Klausel für y die Konstante s_1 ein. Damit erhalten wir die Klausel

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

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

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

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

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

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

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

$$\left\{ \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \right\}$$

unerfüllbar und folglich gilt

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

4.4 Unifikation

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

Definition 37 (Substitution) Es sei eine Signatur

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

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

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

Dabei gilt:

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

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

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

Außerdem definieren wir den *Domain* einer Substitution als

$$\text{dom}(\sigma) = \{x_1, \dots, x_n\}.$$

Die Menge aller Substitutionen bezeichnen wir mit *Subst*. □

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

Definition 38 (Anwendung einer Substitution)

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

1. Falls t eine Variable ist, gibt es zwei Fälle:
 - (a) $t = x_i$ für ein $i \in \{1, \dots, n\}$. Dann definieren wir $x_i\sigma := t_i$.
 - (b) $t = y$ mit $y \in \mathcal{V}$, aber $y \notin \{x_1, \dots, x_n\}$. Dann definieren wir $y\sigma := y$.
2. Andernfalls muß t die Form $t = f(s_1, \dots, s_m)$ haben. Dann können wir $t\sigma$ durch

$$f(s_1, \dots, s_m)\sigma := f(s_1\sigma, \dots, s_m\sigma).$$

definieren, denn nach Induktions-Voraussetzung sind die Ausdrücke $s_i\sigma$ bereits definiert. □

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

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

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

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. $\{p(x_2), q(d, h(x_3, x_1))\}\sigma = \{p(f(d)), q(d, h(x_3, c))\}$.

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

Definition 39 (Komposition von Substitutionen) Es seien

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

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

$$\sigma\tau := [x_1 \mapsto s_1\tau, \dots, x_m \mapsto s_m\tau, y_1 \mapsto t_1, \dots, y_n \mapsto t_n] \quad \square$$

Beispiel: Wir führen das obige Beispiel fort und setzen

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

Dann gilt:

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

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

Satz 40 Ist t ein Term und sind σ und τ Substitutionen mit $\text{dom}(\sigma) \cap \text{dom}(\tau) = \{\}$, so gilt

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

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

Definition 41 (Syntaktische Gleichung) Unter einer *syntaktischen Gleichung* verstehen wir in diesem Abschnitt ein Konstrukt der Form $s \doteq t$, wobei einer der beiden folgenden Fälle vorliegen muß:

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

Weiter definieren wir ein *syntaktisches Gleichungs-System* als eine Menge von syntaktischen Gleichungen. \square

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

Definition 42 (Unifikator) Eine Substitution σ *löst* eine syntaktische Gleichung $s \doteq t$ genau dann, wenn $s\sigma = t\sigma$ ist, wenn also durch die Anwendung von σ auf s und t tatsächlich identische Objekte entstehen. Ist E ein syntaktisches Gleichungs-System, so sagen wir, dass σ ein *Unifikator* von E ist wenn σ jede syntaktische Gleichung in E löst. \square

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

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

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

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

und definieren die Substitution

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

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

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

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

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

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

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

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

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

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

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

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

1. Falls $y \in \mathcal{V}$ eine Variable ist, die nicht in dem Term t auftritt, so können wir die folgende Reduktion durchführen:

$$\langle E \cup \{y \doteq t\}, \sigma \rangle \rightsquigarrow \langle E[y \mapsto t], \sigma[y \mapsto t] \rangle$$

Diese Reduktions-Regel ist folgendermaßen zu lesen: Enthält die zu untersuchende Menge von syntaktischen Gleichungen eine Gleichung der Form $y \doteq t$, wobei die Variable y nicht in t auftritt, dann können wir diese Gleichung aus der gegebenen Menge von Gleichungen entfernen. Gleichzeitig wird die Substitution σ in die Substitution $\sigma[y \mapsto t]$ transformiert und auf die restlichen syntaktischen Gleichungen wird die Substitution $[y \mapsto t]$ angewendet.

2. Wenn die Variable y in dem Term t auftritt, falls also $y \in \text{var}(t)$ ist und wenn außerdem $t \neq y$ ist, dann hat das Gleichungssystem $E \cup \{y \doteq t\}$ keine Lösung, wir schreiben

$$\langle E \cup \{y \doteq t\}, \sigma \rangle \rightsquigarrow \Omega.$$

3. Falls $y \in \mathcal{V}$ eine Variable ist und t keine Variable ist, so haben wir folgende Reduktions-Regel:

$$\langle E \cup \{t \doteq y\}, \sigma \rangle \rightsquigarrow \langle E \cup \{y \doteq t\}, \sigma \rangle.$$

Diese Regel wird benötigt, um anschließend eine der ersten beiden Regeln anwenden zu können.

4. Triviale syntaktische Gleichungen von Variablen können wir einfach weglassen:

$$\langle E \cup \{x \doteq x\}, \sigma \rangle \rightsquigarrow \langle E, \sigma \rangle.$$

5. Ist f ein n -stelliges Funktions-Zeichen, so gilt

$$\langle E \cup \{f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)\}, \sigma \rangle \rightsquigarrow \langle E \cup \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}, \sigma \rangle.$$

Eine syntaktische Gleichung der Form $f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)$ wird also ersetzt durch die n syntaktische Gleichungen $s_1 \doteq t_1, \dots, s_n \doteq t_n$.

Diese Regel ist im Übrigen der Grund dafür, dass wir mit Mengen von syntaktischen Gleichungen arbeiten müssen, denn auch wenn wir mit nur einer syntaktischen Gleichung starten, kann durch die Anwendung dieser Regel die Zahl der syntaktischen Gleichungen erhöht werden.

Ein Spezialfall dieser Regel ist

$$\langle E \cup \{c \doteq c\}, \sigma \rangle \rightsquigarrow \langle E, \sigma \rangle.$$

Hier steht c für eine Konstante, also ein 0-stelliges Funktions-Zeichen. Triviale Gleichungen über Konstanten können also einfach weggelassen werden.

6. Das Gleichungs-System $E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}$ hat keine Lösung, falls die Funktions-Zeichen f und g verschieden sind, wir schreiben

$$\langle E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}, \sigma \rangle \rightsquigarrow \Omega \quad \text{falls } f \neq g.$$

Haben wir ein nicht-leeres Gleichungs-System E gegeben und starten mit dem Paar $\langle E, [] \rangle$, so läßt sich immer eine der obigen Regeln anwenden. Diese geht solange bis einer der folgenden Fälle eintritt:

1. Die 2. oder 6. Regel ist anwendbar. Dann ist das Ergebnis Ω und das Gleichungs-System E hat keine Lösung.
2. Das Paar $\langle E, [] \rangle$ wird reduziert zu einem Paar $\langle \{\}, \sigma \rangle$. Dann ist σ ein Unifikator von E . In diesem Falls schreiben wir $\sigma = \text{mgu}(E)$. Falls $E = \{s \doteq t\}$ ist, schreiben wir auch $\sigma = \text{mgu}(s, t)$. Die Abkürzung mgu steht hier für “*most general unifier*”.

Beispiel: Wir wenden das oben dargestellte Verfahren an, um die syntaktische Gleichung

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

zu lösen. Wir haben die folgenden Reduktions-Schritte:

$$\begin{aligned} & \langle \{p(x_1, f(x_4)) \doteq p(x_2, x_3)\}, [] \rangle \\ & \rightsquigarrow \langle \{x_1 \doteq x_2, f(x_4) \doteq x_3\}, [] \rangle \\ & \rightsquigarrow \langle \{f(x_4) \doteq x_3\}, [x_1 \mapsto x_2] \rangle \\ & \rightsquigarrow \langle \{x_3 \doteq f(x_4)\}, [x_1 \mapsto x_2] \rangle \\ & \rightsquigarrow \langle \{\}, [x_1 \mapsto x_2, x_3 \mapsto f(x_4)] \rangle \end{aligned}$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$[x_1 \mapsto x_2, x_3 \mapsto f(x_4)]$$

als Lösung der oben gegebenen syntaktischen Gleichung.

Wir geben ein weiteres Beispiel und betrachten das Gleichungs-System

$$E = \{p(h(x_1, c)) \doteq p(x_2), q(x_2, d) \doteq q(h(d, c), x_4)\}$$

Wir haben folgende Reduktions-Schritte:

$$\begin{aligned} & \langle \{p(h(x_1, c)) \doteq p(x_2), q(x_2, d) \doteq q(h(d, c), x_4)\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c), d \doteq x_4\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c), x_4 \doteq d\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c)\}, [x_4 \mapsto d] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(h(d, c))\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{h(x_1, c) \doteq h(d, c)\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq d, c \doteq c\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq d, \}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{\}, [x_4 \mapsto d, x_2 \mapsto h(d, c), x_1 \mapsto d] \rangle \end{aligned}$$

Damit haben wir die Substitution $[x_4 \mapsto d, x_2 \mapsto h(d, c), x_1 \mapsto d]$ als Lösung des anfangs gegebenen syntaktischen Gleichungs-Systems gefunden. \square

4.5 Ein Kalkül für die Prädikatenlogik

Der Kalkül, den wir in diesem Abschnitt für die Prädikatenlogik einführen, besteht aus zwei Schluß-Regeln, die wir jetzt definieren.

Definition 43 (Resolution) Es gelte:

1. k_1 und k_2 sind prädikatenlogische Klauseln,
2. $p(s_1, \dots, s_n)$ und $p(t_1, \dots, t_n)$ sind atomare Formeln,
3. die syntaktische Gleichung $p(s_1, \dots, s_n) \doteq p(t_1, \dots, t_n)$ ist lösbar mit

$$\mu = mgu(p(s_1, \dots, s_n), p(t_1, \dots, t_n)).$$

Dann ist

$$\frac{k_1 \cup \{p(s_1, \dots, s_n)\} \quad \{\neg p(t_1, \dots, t_n)\} \cup k_2}{k_1\mu \cup k_2\mu}$$

eine Anwendung der *Resolutions-Regel*. □

Die Resolutions-Regel ist eine Kombination aus der *Substitutions-Regel* und der Schnitt-Regel. Die Substitutions-Regel hat die Form

$$\frac{k}{k\sigma}.$$

Hierbei ist k eine prädikatenlogische Klausel und σ ist eine Substitution. Unter Umständen kann es sein, dass wir bei der Anwendung der Resolutions-Regel die Variablen in einer der beiden Klauseln erst umbenennen müssen bevor wir die Regel anwenden können. Betrachten wir dazu ein Beispiel. Die Klausel-Menge

$$M = \left\{ \{p(x)\}, \{\neg p(f(x))\} \right\}$$

ist widersprüchlich. Wir können die Resolutions-Regel aber nicht unmittelbar anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(x))$$

ist unlösbar. Das liegt daran, dass **zufällig** in beiden Klauseln die selbe Variable verwendet wird. Wenn wir die Variable x in der zweiten Klausel jedoch zu y umbenennen, erhalten wir die Klausel-Menge

$$\left\{ \{p(x)\}, \{\neg p(f(y))\} \right\}.$$

Hier können wir die Resolutions-Regel anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(y))$$

hat die Lösung $[x \mapsto f(y)]$. Dann erhalten wir

$$\{p(x)\}, \quad \{\neg p(f(y))\} \quad \vdash \quad \{\}.$$

und haben damit die Inkonsistenz der Klausel-Menge M nachgewiesen.

Die Resolutions-Regel alleine ist nicht ausreichend, um aus einer Klausel-Menge M , die inkonsistent ist, in jedem Fall die leere Klausel ableiten zu können: Wir brauchen noch eine zweite Regel. Um das einzusehen, betrachten wir die Klausel-Menge

$$M = \left\{ \{p(f(x), y), p(u, g(v))\}, \{\neg p(f(x), y), \neg p(u, g(v))\} \right\}$$

Wir werden gleich zeigen, dass die Menge M widersprüchlich ist. Man kann nachweisen, dass mit der Resolutions-Regel alleine ein solcher Nachweis nicht gelingt. Ein einfacher, aber für die Vorlesung zu aufwendiger Nachweis dieser Behauptung kann geführt werden, indem wir ausgehend von der Menge alle möglichen Resolutions-Schritte durchführen. Dabei würden wir dann sehen, dass die leere Klausel nie berechnet wird. Wir stellen daher jetzt die Faktorisierungs-Regel vor, mir der wir dann später zeigen werden, dass M widersprüchlich ist.

Definition 44 (Faktorisierung) *Es gelte*

1. k ist eine prädikatenlogische Klausel,
2. $p(s_1, \dots, s_n)$ und $p(t_1, \dots, t_n)$ sind atomare Formeln,
3. die syntaktische Gleichung $p(s_1, \dots, s_n) \doteq p(t_1, \dots, t_n)$ ist lösbar,
4. $\mu = mgu(p(s_1, \dots, s_n), p(t_1, \dots, t_n))$.

Dann sind

$$\frac{k \cup \{p(s_1, \dots, s_n), p(t_1, \dots, t_n)\}}{k\mu \cup \{p(s_1, \dots, s_n)\mu\}} \quad \text{und} \quad \frac{k \cup \{\neg p(s_1, \dots, s_n), \neg p(t_1, \dots, t_n)\}}{k\mu \cup \{\neg p(s_1, \dots, s_n)\mu\}}$$

Anwendungen der *Faktorisierungs-Regel*. □

Wir zeigen, wie sich mit Resolutions- und Faktorisierungs-Regel die Widersprüchlichkeit der Menge M beweisen läßt.

1. Zunächst wenden wir die Faktorisierungs-Regel auf die erste Klausel an. Dazu berechnen wir den Unifikator

$$\mu = mgu(p(f(x), y), p(u, g(v))) = [y \mapsto g(v), u \mapsto f(x)].$$

Damit können wir die Faktorisierungs-Regel anwenden:

$$\{p(f(x), y), p(u, g(v))\} \quad \vdash \quad \{p(f(x), g(v))\}.$$

2. Jetzt wenden wir die Faktorisierungs-Regel auf die zweite Klausel an. Dazu berechnen wir den Unifikator

$$\mu = mgu(\neg p(f(x), y), \neg p(u, g(v))) = [y \mapsto g(v), u \mapsto f(x)].$$

Damit können wir die Faktorisierungs-Regel anwenden:

$$\{\neg p(f(x), y), \neg p(u, g(v))\} \quad \vdash \quad \{\neg p(f(x), g(v))\}.$$

3. Wir schließen den Beweis mit einer Anwendung der Resolutions-Regel ab. Der dabei verwendete Unifikator ist die leere Substitution, es gilt also $\mu = []$.

$$\{p(f(x), g(v))\}, \quad \{\neg p(f(x), g(v))\} \quad \vdash \quad \{\}.$$

Ist M eine Menge von prädikatenlogischen Klauseln und ist k eine prädikatenlogische Klausel, die durch Anwendung der Resolutions-Regel und der Faktorisierungs-Regel aus M hergeleitet werden kann, so schreiben wir

$$M \vdash k.$$

Dies wird als M *leitet k her* gelesen.

Definition 45 (Allabschluß) Ist k eine prädikatenlogische Klausel und ist $\{x_1, \dots, x_n\}$ die Menge aller Variablen, die in k auftreten, so definieren wir den *Allabschluß* $\forall(k)$ der Klausel k als

$$\forall(k) := \forall x_1, \dots, x_n: k.$$

Die für uns wesentlichen Eigenschaften des Beweis-Begriffs $M \vdash k$ werden in den folgenden beiden Sätzen zusammengefaßt.

Satz 46 (Korrektheits-Satz)

Ist $M = \{k_1, \dots, k_n\}$ eine Menge von Klauseln und gilt $M \vdash k$, so folgt

$$\models \forall(k_1) \wedge \dots \wedge \forall(k_n) \rightarrow \forall(k).$$

Falls also eine Klausel k aus einer Menge M hergeleitet werden kann, so ist k tatsächlich eine Folgerung aus M . \square

Die Umkehrung des obigen Korrektheits-Satzes gilt nur für die leere Klausel.

Satz 47 (Widerlegungs-Vollständigkeit)

Ist $M = \{k_1, \dots, k_n\}$ eine Menge von Klauseln und gilt $\models \forall(k_1) \wedge \dots \wedge \forall(k_n) \rightarrow \perp$, so folgt

$$M \vdash \{\}. \quad \square$$

Damit haben wir nun ein Verfahren in der Hand, um für eine gegebene prädikatenlogischer Formel f die Frage, ob $\models f$ gilt, untersuchen zu können.

1. Wir berechnen zunächst die Skolem-Normalform von $\neg f$ und erhalten dabei so etwas wie

$$\neg f \approx_e \forall x_1, \dots, x_m: g.$$

2. Anschließend bringen wir die Matrix g in konjunktive Normalform:

$$g \leftrightarrow k_1 \wedge \dots \wedge k_n.$$

Daher haben wir nun

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

und es gilt:

$$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \perp \quad \text{g.d.w.} \quad \{k_1, \dots, k_n\} \models \perp.$$

3. Nach dem Korrektheits-Satz und dem Satz über die Widerlegungs-Vollständigkeit gilt

$$\{k_1, \dots, k_n\} \models \perp \quad \text{g.d.w.} \quad \{k_1, \dots, k_n\} \vdash \perp.$$

Wir versuchen also, nun die Widersprüchlichkeit der Menge $M = \{k_1, \dots, k_n\}$ zu zeigen, indem wir aus M die leere Klausel ableiten. Wenn diese gelingt, haben wir damit die Allgemeingültigkeit der ursprünglich gegebenen Formel f gezeigt.

Zum Abschluß demonstrieren wir das skizzierte Verfahren an einem Beispiel. Wir gehen von folgenden Axiomen aus:

1. Jeder Drache ist glücklich, wenn alle seine Kinder fliegen können.
2. Rote Drachen können fliegen.
3. Die Kinder eines roten Drachens sind immer rot.

Wie werden zeigen, dass aus diesen Axiomen folgt, dass alle roten Drachen glücklich sind. Als erstes formalisieren wir die Axiome und die Behauptung in der Prädikatenlogik. Wir wählen die folgende Signatur

$$\Sigma_{\text{Drache}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle \quad \text{mit}$$

1. $\mathcal{V} := \{x, y, z\}$.
2. $\mathcal{F} = \{\}$.
3. $\mathcal{P} := \{\text{rot}, \text{fliegt}, \text{glücklich}, \text{kind}\}$.
4. $\text{arity} := \{\langle \text{rot}, 1 \rangle, \langle \text{fliegt}, 1 \rangle, \langle \text{glücklich}, 1 \rangle, \langle \text{kind}, 2 \rangle\}$

Das Prädikat $\text{kind}(x, y)$ soll genau dann wahr sein, wenn x ein Kind von y ist. Formalisieren wir die Axiome und die Behauptung, so erhalten wir die folgenden Formeln f_1, \dots, f_4 :

1. $f_1 := \forall x : (\forall y : (\text{kind}(y, x) \rightarrow \text{fliegt}(y)) \rightarrow \text{glücklich}(x))$
2. $f_2 := \forall x : (\text{rot}(x) \rightarrow \text{fliegt}(x))$
3. $f_3 := \forall x : (\text{rot}(x) \rightarrow \forall y : \text{kind}(y, x) \rightarrow \text{rot}(y))$
4. $f_4 := \forall x : (\text{rot}(x) \rightarrow \text{glücklich}(x))$

Wir wollen zeigen, dass die Formel

$$f := f_1 \wedge f_2 \wedge f_3 \rightarrow f_4$$

allgemeingültig ist. Wir betrachten also die Formel $\neg f$ und stellen fest

$$\neg f \leftrightarrow f_1 \wedge f_2 \wedge f_3 \wedge \neg f_4.$$

Als nächstes müssen wir diese Formel in eine Menge von Klauseln umformen. Da es sich hier um eine Konjunktion mehrerer Formeln handelt, können wir die einzelnen Formeln f_1, f_2, f_3 und $\neg f_4$ getrennt in Klauseln umwandeln.

1. Die Formel f_1 kann wie folgt umgeformt werden:

$$\begin{aligned}
f_1 &= \forall x : (\forall y : (\text{kind}(y, x) \rightarrow \text{fliegt}(y)) \rightarrow \text{glücklich}(x)) \\
&\leftrightarrow \forall x : (\neg \forall y : (\text{kind}(y, x) \rightarrow \text{fliegt}(y)) \vee \text{glücklich}(x)) \\
&\leftrightarrow \forall x : (\neg \forall y : (\neg \text{kind}(y, x) \vee \text{fliegt}(y)) \vee \text{glücklich}(x)) \\
&\leftrightarrow \forall x : (\exists y : \neg (\neg \text{kind}(y, x) \vee \text{fliegt}(y)) \vee \text{glücklich}(x)) \\
&\leftrightarrow \forall x : (\exists y : (\text{kind}(y, x) \wedge \neg \text{fliegt}(y)) \vee \text{glücklich}(x)) \\
&\leftrightarrow \forall x : \exists y : ((\text{kind}(y, x) \wedge \neg \text{fliegt}(y)) \vee \text{glücklich}(x)) \\
&\approx_e \forall x : ((\text{kind}(s(x), x) \wedge \neg \text{fliegt}(s(x))) \vee \text{glücklich}(x))
\end{aligned}$$

Im letzten Schritt haben wir dabei die Skolem-Funktion s mit $\text{arity}(s) = 1$ eingeführt. Anschaulich berechnet diese Funktion für jeden Drachen x , der nicht glücklich ist, ein Kind y , das nicht fliegen kann. Wenn wir in der Matrix dieser Formel das “ \vee ” noch Ausmultiplizieren, so erhalten wir die beiden Klauseln

$$\begin{aligned}
k_1 &:= \{\text{kind}(s(x), x), \text{glücklich}(x)\}, \\
k_2 &:= \{\neg \text{fliegt}(s(x)), \text{glücklich}(x)\}.
\end{aligned}$$

2. Analog finden wir für f_2 :

$$\begin{aligned}
f_2 &= \forall x : (\text{rot}(x) \rightarrow \text{fliegt}(x)) \\
&\leftrightarrow \forall x : (\neg \text{rot}(x) \vee \text{fliegt}(x))
\end{aligned}$$

Damit ist f_2 zu folgender Klauseln äquivalent:

$$k_3 := \{\neg \text{rot}(x), \text{fliegt}(x)\}.$$

3. Für f_3 sehen wir:

$$\begin{aligned} f_3 &= \forall x : \left(\text{rot}(x) \rightarrow \forall y : (\text{kind}(y, x) \rightarrow \text{rot}(y)) \right) \\ &\leftrightarrow \forall x : \left(\neg \text{rot}(x) \vee \forall y : (\neg \text{kind}(y, x) \vee \text{rot}(y)) \right) \\ &\leftrightarrow \forall x : \forall y : (\neg \text{rot}(x) \vee \neg \text{kind}(y, x) \vee \text{rot}(y)) \end{aligned}$$

Das liefert die folgende Klausel:

$$k_4 := \{ \neg \text{rot}(x), \neg \text{kind}(y, x), \text{rot}(y) \}.$$

4. Umformung der Negation von f_4 liefert:

$$\begin{aligned} \neg f_4 &= \neg \forall x : (\text{rot}(x) \rightarrow \text{glücklich}(x)) \\ &\leftrightarrow \neg \forall x : (\neg \text{rot}(x) \vee \text{glücklich}(x)) \\ &\leftrightarrow \exists x : \neg (\neg \text{rot}(x) \vee \text{glücklich}(x)) \\ &\leftrightarrow \exists x : (\text{rot}(x) \wedge \neg \text{glücklich}(x)) \\ &\approx_e \text{rot}(d) \wedge \neg \text{glücklich}(d) \end{aligned}$$

Die hier eingeführte Skolem-Konstante d steht für einen unglücklichen roten Drachen. Das führt zu den Klauseln

$$\begin{aligned} k_5 &= \{ \text{rot}(d) \}, \\ k_6 &= \{ \neg \text{glücklich}(d) \}. \end{aligned}$$

Wir müssen also untersuchen, ob die Menge M , die aus den folgenden Klauseln besteht, widersprüchlich ist:

1. $k_1 = \{ \text{kind}(s(x), x), \text{glücklich}(x) \}$
2. $k_2 = \{ \neg \text{fliegt}(s(x)), \text{glücklich}(x) \}$
3. $k_3 = \{ \neg \text{rot}(x), \text{fliegt}(x) \}$
4. $k_4 = \{ \neg \text{rot}(x), \neg \text{kind}(y, x), \text{rot}(y) \}$
5. $k_5 = \{ \text{rot}(d) \}$
6. $k_6 = \{ \neg \text{glücklich}(d) \}$

Sei also $M := \{k_1, k_2, k_3, k_4, k_5, k_6\}$. Wir zeigen, dass $M \vdash \perp$ gilt:

1. Es gilt

$$\text{mgu}(\text{rot}(d), \text{rot}(x)) = [x \mapsto d].$$

Daher können wir die Resolutions-Regel auf die Klauseln k_5 und k_4 wie folgt anwenden:

$$\{ \text{rot}(d) \}, \{ \neg \text{rot}(x), \neg \text{kind}(y, x), \text{rot}(y) \} \vdash \{ \neg \text{kind}(y, d), \text{rot}(y) \}.$$

2. Wir wenden nun auf die resultierende Klausel und auf die Klausel k_1 die Resolutions-Regel an. Dazu berechnen wir zunächst

$$\text{mgu}(\text{kind}(y, d), \text{kind}(s(x), x)) = [y \mapsto s(d), x \mapsto d].$$

Dann haben wir

$$\{ \neg \text{kind}(y, d), \text{rot}(y) \}, \{ \text{kind}(s(x), x), \text{glücklich}(x) \} \vdash \{ \text{glücklich}(d), \text{rot}(s(d)) \}.$$

3. Jetzt wenden wir auf die eben abgeleitete Klausel und die Klausel k_6 die Resolutions-Regel an. Wir haben:

$$\text{mgu}(\text{glücklich}(d), \text{glücklich}(d)) = []$$

Also erhalten wir

$$\{\text{glücklich}(d), \text{rot}(s(d))\}, \{\neg \text{glücklich}(d)\} \vdash \{\text{rot}(s(d))\}.$$

4. Auf die Klausel $\{\text{rot}(s(d))\}$ und die Klausel k_3 wenden wir die Resolutions-Regel an. Zunächst haben wir

$$\text{mgu}(\text{rot}(s(d)), \neg \text{rot}(x)) = [x \mapsto s(d)]$$

Also liefert die Anwendung der Resolutions-Regel:

$$\{\text{rot}(s(d))\}, \{\neg \text{rot}(x), \text{fliegt}(x)\} \vdash \{\text{fliegt}(s(d))\}$$

5. Um die so erhaltenen Klausel $\{\text{fliegt}(s(d))\}$ mit der Klausel k_3 resolvieren zu können, berechnen wir

$$\text{mgu}(\text{fliegt}(s(d)), \text{fliegt}(s(x))) = [x \mapsto d]$$

Dann liefert die Resolutions-Regel

$$\{\text{fliegt}(s(d))\}, \{\neg \text{fliegt}(s(x)), \text{glücklich}(x)\} \vdash \{\text{glücklich}(d)\}.$$

6. Auf das Ergebnis $\{\text{glücklich}(d)\}$ und die Klausel k_6 können wir nun die Resolutions-Regel anwenden:

$$\{\text{glücklich}(d)\}, \{\neg \text{glücklich}(d)\} \vdash \{\}.$$

Da wir im letzten Schritt die leere Klausel erhalten haben, ist insgesamt $M \vdash \perp$ nachgewiesen worden und damit haben wir gezeigt, dass alle kommunistischen Drachen glücklich sind.

Aufgabe: Die von Bertrand Russell definierte *Russell-Menge* R ist definiert als die Menge aller der Mengen, die sich nicht selbst enthalten. Damit gilt also

$$\forall x : (x \in R \leftrightarrow \neg x \in R).$$

Zeigen Sie mit Hilfe des in diesem Abschnitt definierten Kalküls, dass diese Formel widersprüchlich ist.

Hausaufgabe: Gegeben seien folgende Axiome:

1. Jeder Barbier rasiert alle Personen, die sich nicht selbst rasieren.
2. Kein Barbier rasiert jemanden, der sich selbst rasiert.

Zeigen Sie, dass aus diesen Axiomen logisch die folgende Aussage folgt:

Alle Barbieri sind blond.

4.6 *Prover9* und *Mace4*

Der im letzten Abschnitt beschriebene Kalkül lässt sich automatisieren und bildet die Grundlage moderner automatischer Beweiser. Gleichzeitig lässt sich auch die Suche nach Gegenbeispielen automatisieren. Wir stellen in diesem Abschnitt zwei Systeme vor, die diesen Zwecken dienen.

1. *Prover9* dient dazu, automatisch prädikatenlogische Formeln zu beweisen.
2. *Mace4* untersucht, ob eine gegebene Menge prädikatenlogischer Formeln in einer endlichen Struktur erfüllbar ist. Gegebenenfalls wird diese Struktur berechnet.

Die beiden Programme *Prover9* und *Mace4* wurden von William McCune [McC10] entwickelt, stehen unter der GPL und können unter der Adresse

<http://www.cs.unm.edu/~mccune/prover9/download>

als Quelltext heruntergeladen werden. Wir diskutieren zunächst *Prover9* und schauen uns anschließend *Mace4* an.

4.6.1 Der automatische Beweiser *Prover9*

Prover9 ist ein Programm, das als Eingabe zwei Mengen von Formeln bekommt. Die erste Menge von Formeln wird als Menge von *Axiomen* interpretiert, die zweite Menge von Formeln sind die zu beweisenden *Theoreme*, die aus den Axiomen gefolgert werden sollen. Wollen wir beispielsweise zeigen, dass aus den drei Formeln der Gruppentheorie

1. $\forall x : e \cdot x = x$,
2. $\forall x : \exists y : x \cdot y = e$ und
3. $\forall x : \forall y : \forall z : (x \cdot y) \cdot z = x \cdot (y \cdot z)$

die beiden Formeln

1. $\forall x : x \cdot e = x$,
2. $\forall x : \exists y : y \cdot x = e$ und

logisch folgen, so stellen wir diese Formeln wie in Abbildung 4.6 auf Seite 109 gezeigt als Text dar. Der Anfang der Axiome wird in dieser Datei durch “**formulas(sos)**” eingeleitet und durch das Schlüsselwort “**end_of_list**” beendet. Zu beachten ist, dass sowohl die Schlüsselwörter als auch die einzelnen Formel jeweils durch einen Punkt “.” beendet werden. Die Axiome in den Zeilen 2, 3, und 4 drücken aus, dass

1. **e** ein links-neutrales Element ist,
2. zu jedem Element x ein links-inverses Element y existiert und
3. das Assoziativ-Gesetz gilt.

Aus diesen Axiomen folgt, dass das **e** auch ein rechts-neutrales Element ist und dass außerdem zu jedem Element x ein rechts-neutrales Element y existiert. Diese beiden Formeln sind die zu beweisenden *Ziele* und werden in der Datei durch “**formulas(goal)**” markiert. Trägt die in Abbildung 4.6 gezeigte Datei den Namen “**group2.in**”, so können wir das Programm *Prover9* mit dem Befehl

```
prover9 -f group2.in
```

```

1  formulas(sos).
2  all x (e * x = x).                % left neutral
3  all x exists y (y * x = e).        % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)). % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x (x * e = x).                % right neutral
9  all x exists y (x * y = e).        % right inverse
10 end_of_list.

```

Abbildung 4.6: Textuelle Darstellung der Axiome der Gruppentheorie.

starten und erhalten als Ergebnis die Information, dass die beiden in Zeile 8 und 9 gezeigten Formeln tatsächlich aus den vorher angegebenen Axiomen folgen. Hätte *Prover9* keinen Beweis gefunden, so wäre das Programm solange weitergelaufen, bis kein freier Speicher mehr zur Verfügung gestanden hätte und wäre dann mit einer Fehlermeldung abgebrochen.

Prover9 versucht, einen indirekten Beweis zu führen. Zunächst werden die Axiome in prädikatenlogische Klauseln überführt. Dann wird jedes zu beweisende Theorem negiert und die negierte Formel wird ebenfalls in Klauseln überführt. Anschließend versucht *Prover9* aus der Menge aller Axiome zusammen mit den Klauseln, die sich aus der Negation eines der zu beweisenden Theoreme ergeben, die leere Klausel herzuleiten. Gelingt dies, so ist bewiesen, dass das jeweilige Theorem tatsächlich aus den Axiomen folgt. Abbildung 4.7 zeigt eine Eingabe-Datei für *Prover9*, bei der versucht wird, das Kommutativ-Gesetz aus den Axiomen der Gruppentheorie zu folgern. Der Beweis-Versuch mit *Prover9* schlägt allerdings fehl. In diesem Fall wird die Beweissuche nicht endlos fortgesetzt. Dies liegt daran, dass es *Prover9* gelingt, in endlicher Zeit alle aus den gegebenen Voraussetzungen folgenden Formeln abzuleiten. Ein solcher Fall ist allerdings eher die Ausnahme als die Regel.

```

1  formulas(sos).
2  all x (e * x = x).                % left neutral
3  all x exists y (y * x = e).        % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)). % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x all y (x * y = y * x).        % * is commutative
9  end_of_list.

```

Abbildung 4.7: Gilt das Kommutativ-Gesetz in allen Gruppen?

4.6.2 Mace4

Dauert ein Beweisversuch mit *Prover9* endlos, so ist zunächst nicht klar, ob das zu beweisende Theorem gilt. Um sicher zu sein, dass eine Formel nicht aus einer gegebenen Menge von Axiomen folgt, reicht es aus, eine Struktur zu konstruieren, in der alle Axiome erfüllt sind, in der das zu beweisende Theorem aber falsch ist. Das Programm *Mace4* dient genau dazu, solche Strukturen zu finden. Das funktioniert natürlich nur, solange die Strukturen endlich sind. Abbildung 4.8 zeigt eine Eingabe-Datei, mit deren Hilfe wir die Frage, ob es endliche nicht-kommutative Gruppen gibt, unter Verwendung von *Mace4* beantworten können. In den Zeilen 2, 3 und 4 stehen die Axiome

der Gruppen-Theorie. Die Formel in Zeile 5 postuliert, dass für die beiden Elemente a und b das Kommutativ-Gesetz nicht gilt, dass also $a \cdot b \neq b \cdot a$ ist. Ist der in Abbildung 4.8 gezeigte Text in einer Datei mit dem Namen “group.in” gespeichert, so können wir *Mace4* durch das Kommando

mace4 -f group.in

starten. *Mace4* sucht für alle positiven natürlichen Zahlen $n = 1, 2, 3, \dots$, ob es eine Struktur $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ mit $\text{card}(\mathcal{U}) = n$ gibt, in der die angegebenen Formeln gelten. Bei $n = 6$ wird *Mace4* fündig und berechnet tatsächlich eine Gruppe mit 6 Elementen, in der das Kommutativ-Gesetz verletzt ist.

```

1  formulas(theory).
2  all x (e * x = x).                                % left neutral
3  all x exists y (y * x = e).                        % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)).    % associativity
5  a * b != b * a.                                    % a and b do not commute
6  end_of_list.

```

Abbildung 4.8: Gibt es eine Gruppe, in der das Kommutativ-Gesetz nicht gilt?

Abbildung 4.9 zeigt einen Teil der von *Mace4* produzierten Ausgabe. Die Elemente der Gruppe sind die Zahlen $0, \dots, 5$, die Konstante a ist das Element 0, b ist das Element 1, e ist das Element 2. Weiter sehen wir, dass das Inverse von 0 wieder 0 ist, das Inverse von 1 ist 1 das Inverse von 2 ist 2, das Inverse von 3 ist 4, das Inverse von 4 ist 3 und das Inverse von 5 ist 5. Die Multiplikation wird durch die folgende Gruppen-Tafel realisiert:

\circ	0	1	2	3	4	5
0	2	3	0	1	5	4
1	4	2	1	5	0	3
2	0	1	2	3	4	5
3	5	0	3	4	2	1
4	1	5	4	2	3	0
5	3	4	5	0	1	2

Diese Gruppen-Tafel zeigt, dass

$$a \circ b = 0 \circ 1 = 3, \quad \text{aber} \quad b \circ a = 1 \circ 0 = 4$$

gilt, mithin ist das Kommutativ-Gesetz tatsächlich verletzt.

Bemerkung: Der Theorem-Beweiser *Prover9* ist ein Nachfolger des Theorem-Beweisers *Otter*. Mit Hilfe von *Otter* ist es McCune 1996 gelungen, die Robbin’sche Vermutung zu beweisen [McC97]. Dieser Beweis war damals sogar der *New York Times* eine Schlagzeile wert, nachzulesen unter

<http://www.nytimes.com/library/cyber/week/1210math.html>.

Das Ergebnis zeigt, dass automatische Theorem-Beweiser durchaus nützliche Werkzeuge sein können. Nichtdestoweniger ist die Prädikatenlogik unentscheidbar und bisher sind auch nur wenige offene mathematische Probleme mit Hilfe von automatischen Beweisern gelöst worden. Das wird sich vermutlich auch in der näheren Zukunft nicht ändern.

```

1  ===== DOMAIN SIZE 6 =====
2
3  === Mace4 starting on domain size 6. ===
4
5  ===== MODEL =====
6
7  interpretation( 6, [number=1, seconds=0], [
8
9      function(a, [ 0 ]),
10
11     function(b, [ 1 ]),
12
13     function(e, [ 2 ]),
14
15     function(f1(_), [ 0, 1, 2, 4, 3, 5 ]),
16
17     function(*(_,_), [
18         2, 3, 0, 1, 5, 4,
19         4, 2, 1, 5, 0, 3,
20         0, 1, 2, 3, 4, 5,
21         5, 0, 3, 4, 2, 1,
22         1, 5, 4, 2, 3, 0,
23         3, 4, 5, 0, 1, 2 ])
24 ]).
25
26 ===== end of model =====

```

Abbildung 4.9: Ausgabe von *Mace4*.

Kapitel 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 113 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

$$\forall x_1, \dots, x_m : (B_1 \wedge \dots \wedge B_m \rightarrow A)$$

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).
```

Abbildung 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, _J, 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 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 andere 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 118 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 “1” (*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.

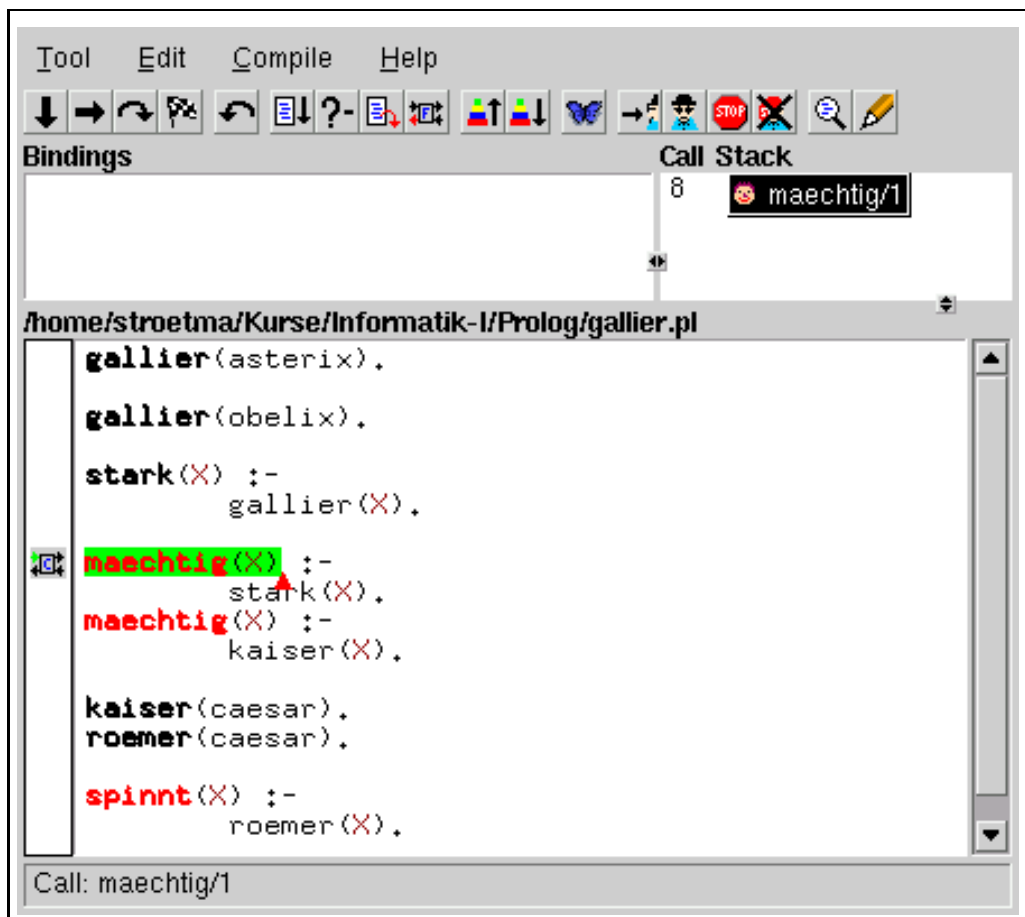


Abbildung 5.2: Der Debugger des SWI-Prolog-Systems.

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 $\mu_i = \text{mgu}(Q_1, A^{(i)})$ 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 die *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) .

```

Abbildung 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 er 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.

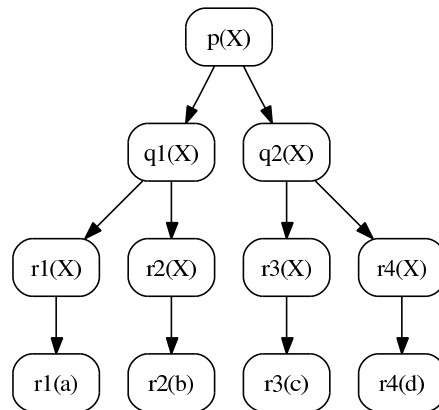


Abbildung 5.4: Der Suchbaum für das in Abbildung 5.3 gezeigte Programm.

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).

```

Abbildung 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:

$$\begin{aligned} \text{Name} &\in \{\text{julia}, \text{thomas}, \text{michael}\}, & \text{Car} &\in \{\text{ford}, \text{toyota}, \text{audi}\}, \\ \text{Language} &\in \{\text{java}, \text{prolog}, \text{setl}\}. \end{aligned}$$

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:

$$\text{first_name}(\text{person}(\text{Name}, \text{Car}, \text{Language}), \text{Name}).$$

2. Das Prädikat `car/2` extrahiert die Auto-Marke:

$$\text{car}(\text{person}(\text{Name}, \text{Car}, \text{Language}), \text{Car}).$$

3. Das Prädikat `language/2` extrahiert die Programmier-Sprache:

$$\text{language}(\text{person}(\text{Name}, \text{Car}, \text{Language}), \text{Language}).$$

Um zu verstehen wie diese Prädikate arbeiten, zeigen wir, wie die Anfrage

$$\text{car}(\text{person}(\text{hans}, \text{seat}, \text{setl}), \text{X}).$$

von dem *Prolog*-System beantwortet wird. Die einzige Regel, die zur Beantwortung dieser Anfrage herangezogen werden kann, ist die Regel

$$\text{car}(\text{person}(\text{Name}, \text{Car}, \text{Language}), \text{Car}) \text{ :- true.}$$

Um diese Regel anwenden zu können, ist die syntaktische Gleichung

$$\text{car}(\text{person}(\text{hans}, \text{seat}, \text{setl}), \text{X}) \doteq \text{car}(\text{person}(\text{Name}, \text{Car}, \text{Language}), \text{Car})$$

zu lösen. Bei der Unifikation findet sich die Lösung

$$\mu = [\text{Name} \mapsto \text{hans}, \text{Car} \mapsto \text{seat}, \text{Language} \mapsto \text{setl}, \text{X} \mapsto \text{seat}].$$

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

$$\text{sequence}(\text{First}, \text{Second}, \text{Third}).$$

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

$$\text{did_better}(\text{Better}, \text{Worse}, \text{Sequence})$$

berechnet werden, dessen Implementierung in den Zeilen 38 – 40 der Abbildung 5.6 auf Seite 125 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 Rätsel 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.

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, das zwei Listen aneinander hängen kann. Das Programm implementiert das dreistellige Prädikat **myAppend**³. Die Intention ist, dass **myAppend**(*l*₁,*l*₂,*l*₃) für drei Listen *l*₁, *l*₂ und *l*₃ genau dann wahr sein soll, wenn die Liste *l*₃ dadurch entsteht, dass die Liste *l*₂ hinten an die Liste *l*₁ angehängt wird. Das Programm besteht aus den folgenden beiden Klauseln:

³In dem *SWI-Prolog*-System gibt es das vordefinierte Prädikat **append/3**, das genau das selbe leistet wie unsere Implementierung von **myAppend/3**.

```

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).

```

Abbildung 5.6: Wer fährt Toyota?

```
myAppend( [], L, L ).
myAppend( [ X | L1 ], L2, [ X | L3 ] ) :- myAppend( 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 “myAppend.pl” abgespeichert sind und dass wir diese Datei mit dem Befehl “consult(myAppend).” geladen haben. Dann stellen wir die Anfrage

```
?- myAppend( [ 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 myAppend 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

```
myAppend(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:

```
?- myAppend(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.

```

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 ).

```

Abbildung 5.7: Sortieren durch Einfügen.

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

`[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. `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.

```

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 ).

```

Abbildung 5.8: Sortieren durch Mischen.

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**($[]$) = $[]$.

$$2. \text{ odd}([h|t]) = [h|\text{even}(t)],$$

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.

$$3. \text{ even}([]) = [].$$

$$4. \text{ even}([h|t]) = \text{odd}(t),$$

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 129 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:

$$n \in \text{Expr} \quad \text{für alle } n \in \mathbb{Z},$$

$$r \in \text{Expr} \quad \text{für alle } r \in \mathbb{R}.$$

3. Das Negative eines arithmetischen Ausdrucks ist ein arithmetischer Ausdruck. In *Prolog* kann das Negative durch den unären Operator “-” dargestellt werden, also haben wir

$$-t \in \text{Expr} \quad \text{falls } t \in \text{Expr}.$$

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

$$s + t \in \text{Expr} \quad \text{falls } s, t \in \text{Expr}.$$

$$s - t \in Expr \quad \text{falls } s, t \in Expr.$$

$$s * t \in Expr \quad \text{falls } s, t \in Expr.$$

$$s / t \in Expr \quad \text{falls } s, t \in Expr.$$

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

$$y \neq x \rightarrow \text{diff}(x, x) = 0.$$

2. Die Ableitung einer Zahl n ergibt 0:

$$\frac{dn}{dx} = 0.$$

Damit haben wir

$$\text{diff}(n, x) = 0.$$

3. Die Ableitung eines Ausdrucks mit negativen Vorzeichen ist durch

$$\frac{d}{dx}(-f) = -\frac{df}{dx}$$

gegeben. Die rekursive Gleichung lautet

$$\text{diff}(-f, x) = -\text{diff}(f, x).$$

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

$$\text{diff}(f + g, x) = \text{diff}(f, x) + \text{diff}(g, x).$$

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

$$\text{diff}(f - g, x) = \text{diff}(f, x) - \text{diff}(g, x).$$

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

$$\text{diff}(f * g, x) = \text{diff}(f, x) * g + f * \text{diff}(g, x).$$

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

$$\text{diff}(f/g, x) = (\text{diff}(f, x) * g - f * \text{diff}(g, x)) / (g * g).$$

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

$$\text{diff}(\ln(f), x) = \text{diff}(f, x) / f.$$

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.

1. 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.

2. 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 Variablenamen ein Unterstrichs “*_*” angefügt wird.

3. 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.

4. 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).

```

Abbildung 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  difference := procedure(l1, l2) {
2      return [ x in l1 | !(x in l2) ];
3  };
```

Abbildung 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 ).
```

Abbildung 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).
```

Abbildung 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 des Aufrufs 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).
```

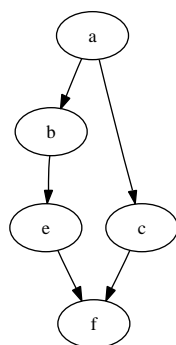


Abbildung 5.13: Ein einfacher Graph ohne Zykeln

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 ).
```

Abbildung 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).

```

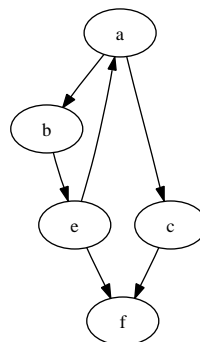


Abbildung 5.15: Ein Graph mit einem Zykel

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 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 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 ).

```

Abbildung 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**, denn 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 finden, wir erhalten folgendes Ergebnis:

```

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 143 zeigt ein *Prolog*-Programm, was das Rätsel löst. Den von diesem Programm berechneten Fahrplan finden Sie in Abbildung 5.17 auf Seite 142. Wir diskutieren dieses Programm nun Zeile für Zeile.

1. Wir beginnen mit dem Hilfs-Prädikat `otherSide/4`, das in den Zeilen 24 – 26 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 29 – 34 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.

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

Abbildung 5.17: Fahrplan für Missionare und Kannibalen

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 37 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.

- Bei der Implementierung des Prädikats `edge/2` verwenden wir in den Zeilen 9 und 10 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, ..., *High*.

Beispielsweise gibt die Anfrage

`between(1,3,N), write(N), nl, fail.`

nacheinander die Zahlen 1, 2 und 3 am Bildschirm aus.

- 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 9 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 10 die Zahl *KB* der Kannibalen, die im Boot übersetzen. In Zeile 11 testen wir, dass es mindestens einen Passagier gibt, der mit dem Boot übersetzt und in Zeile 12 testen wir, dass es höchstens zwei Passagiere sind. In Zeile 13 und 14 berechnen wir die

```

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 ).

```

Abbildung 5.18: Missionare und Kannibalen

Zahl MN der Missionare und die Zahl KN der Kannibalen, die nach der Überfahrt auf dem linken Ufer verbleiben und testen dann in Zeile 15, 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 19 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 21 berechnen, wieviele Missionare und Kannibalen sich dann am gegenüberliegenden Ufer befinden.

5. In den Zeilen 1 – 4 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 hier aus Platzgründen nicht angegeben wird.

5.6 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 “`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 unifzieren. 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 unifzieren. 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 unifziert werden.

3. Um die Teilanfrage $a(U)$ zu beantworten, wird $a(U)$ mit $a(1)$ unifziert. 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 $b(1)$ unifzieren 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.

5.6.1 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.

```

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 ] ).

```

Abbildung 5.19: Der Bubble-Sort Algorithmus

Abbildung 5.19 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 **append/3** die neue Liste

$$\text{append}(L_1, [Y, X|L_2])$$

gebildet und diese Liste wird rekursiv sortiert. Wenn es nicht möglich ist, die Liste L so in zwei

Listen L_1 und $[X, Y \mid L_2]$ 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 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 5.20 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 L_1 und $[X, Y \mid L_2]$ 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 L_1 und $[X, Y \mid L_2]$ 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 ).
```

Abbildung 5.20: 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 5.21 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.

```

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 ).

```

Abbildung 5.21: Sortieren durch Mischen mit Cut-Operatoren.

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.

5.7 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 die folgende Hinweise auf die Literatur geben:

1. *The Art of Prolog* von Leon Sterling und Ehud Shapiro [SS94]. Dieses Werk ist ein ausgezeichnete 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 Programmming* 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.
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.

Literaturverzeichnis

- [Bra90] Ivan Bratko. *PROLOG Programming for Artificial Intelligence*. Addison-Wesley, 2nd edition, 1990.
- [DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1987.
- [McC97] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19:263–276, December 1997.
- [McC10] William McCune. Prover9 and Mace4. <http://www.cs.unm.edu/~mccune/prover9/>, 2005–2010.
- [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.
- [Sch87] Uwe Schöning. *Logik für Informatiker*. 1987.
- [SDSD86] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming With Sets: An Introduction to SETL*. Springer-Verlag, 1986.
- [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>.