

# Grundlagen der Informatik

Karl Stroetmann

22. März 2005

# Inhaltsverzeichnis

<b>1</b>	<b>Mathematische Grundlagen</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Prädikatenlogische Formeln . . . . .	5
1.2.1	Warum Formeln . . . . .	5
1.2.2	Formeln als Kurzschreibweise . . . . .	6
1.2.3	Beispiele für Terme und Formeln . . . . .	9
1.3	Mengen und Relationen . . . . .	10
1.3.1	Erzeugung von Mengen durch explizites Auflisten . . . . .	11
1.3.2	Die Menge der natürlichen Zahlen . . . . .	11
1.3.3	Das Auswahl-Prinzip . . . . .	12
1.3.4	Potenz-Mengen . . . . .	12
1.3.5	Vereinigungs-Mengen . . . . .	12
1.3.6	Schnitt-Menge . . . . .	13
1.3.7	Differenz . . . . .	13
1.3.8	Bild-Mengen . . . . .	13
1.3.9	Kartesische Produkte . . . . .	13
1.3.10	Gleichheit von Mengen . . . . .	14
1.3.11	Rechenregeln für das Arbeiten mit Mengen . . . . .	15
1.4	Binäre Relationen . . . . .	15
1.4.1	Binäre Relationen und Funktionen . . . . .	15
1.4.2	Spezielle Relationen . . . . .	19
<b>2</b>	<b>Die Programmier-Sprache SETL2</b>	<b>23</b>
2.1	Einführende Beispiele . . . . .	23
2.2	Darstellung von Mengen . . . . .	26
2.3	Paare und Funktionen . . . . .	29
2.4	Allgemeine Tupel . . . . .	30
2.5	Kontroll-Strukturen . . . . .	33
2.5.1	Schleifen . . . . .	34
2.6	Fallstudie: Berechnung des kürzesten Wegs . . . . .	37
2.6.1	Berechnung des transitiven Abschlusses einer Relation . . . . .	38
2.6.2	Berechnung des kürzesten Weges . . . . .	39
2.6.3	Ausblick . . . . .	41
<b>3</b>	<b>Aussagenlogik</b>	<b>43</b>
3.1	Motivation . . . . .	43
3.2	Anwendungen der Aussagenlogik . . . . .	45
3.3	Formale Definition der aussagenlogischen Formeln . . . . .	46
3.3.1	Implementierung in SETL2 . . . . .	49
3.3.2	Eine Anwendung . . . . .	51
3.4	Tautologien . . . . .	53
3.4.1	Testen der Allgemeingültigkeit in SETL2 . . . . .	54

3.4.2	Nachweis der Allgemeingültigkeit durch Äquivalenz-Umformungen . . . . .	56
3.4.3	Berechnung der konjunktiven Normalform in SETL2 . . . . .	59
3.5	Der Herleitungs-Begriff . . . . .	64
3.5.1	Eigenschaften des Herleitungs-Begriffs . . . . .	66
3.6	Das Verfahren von Davis und Putnam . . . . .	67
3.6.1	Vereinfachung mit der Schnitt-Regel . . . . .	68
3.6.2	Vereinfachung durch Subsumption . . . . .	68
3.6.3	Vereinfachung durch Fallunterscheidung . . . . .	69
3.6.4	Der Algorithmus von Davis und Putnam . . . . .	69
3.6.5	Ein Beispiel . . . . .	70
3.6.6	Implementierung des Algorithmus von Davis und Putnam . . . . .	71
3.7	Das 8-Damen-Problem . . . . .	74
<b>4</b>	<b>Prädikatenlogik</b>	<b>81</b>
4.1	Syntax der Prädikatenlogik . . . . .	81
4.2	Semantik der Prädikatenlogik . . . . .	84
4.3	Normalformen für prädikatenlogische Formeln . . . . .	88
4.4	Unifikation . . . . .	92
4.5	Ein Kalkül für die Prädikatenlogik . . . . .	96
<b>5</b>	<b><i>Prolog</i></b>	<b>101</b>
5.1	Wie arbeitet <i>Prolog</i> ? . . . . .	104
5.2	Ein komplexeres Beispiel . . . . .	108
5.3	Listen . . . . .	110
5.3.1	Sortieren durch Einfügen . . . . .	112
5.3.2	Sortieren durch Mischen . . . . .	114
5.3.3	Symbolisches Differenzieren . . . . .	115
5.4	Negation . . . . .	120
5.5	Der Cut-Operator . . . . .	122
5.5.1	Verbesserung der Effizienz von <i>Prolog</i> -Programmen durch den Cut-Operator	124
5.6	Die Tiefen-Suche in <i>Prolog</i> . . . . .	128
5.6.1	Missionare und Kannibalen . . . . .	131
	<b>Literatur-Verzeichnis</b> . . . . .	<b>134</b>

# Kapitel 1

## Mathematische Grundlagen

Die erste Informatik-Vorlesung an der Berufs-Akademie beschäftigt sich mit der *mathematischen Logik*, der *Mengenlehre* und dem *logischen Programmieren*. Da insbesondere die mathematische Logik und die Mengenlehre sehr abstrakte Gebiete sind, bereiten sie erfahrungsgemäß vielen Studenten Schwierigkeiten. Der Umgang mit dieser, zugegebenermaßen sehr trockenen Materie fällt auch deshalb schwer, weil zunächst noch gar nicht klar ist, wozu die Logik in der Informatik überhaupt benötigt wird. Aus diesem Grunde möchte ich an den Anfang dieser Vorlesung eine Motivation stellen. Diese Motivation soll Ihnen zeigen, dass es zwingend notwendig ist, den Entwurf von Software und Hardware auf eine solide Grundlage zu stellen.

### 1.1 Motivation

Wir beginnen mit der Feststellung, dass informationstechnische Systeme (im folgenden kurz als IT-Systeme bezeichnet) zu den komplexesten Systemen gehören, die die Menschheit je entwickelt hat. Das läßt sich schon an dem Aufwand erkennen, der bei der Erstellung von IT-Systemen anfällt. So sind im Bereich der Telekommunikations-Industrie IT-Projekte, bei denen mehr als 1000 Entwickler über mehrere Jahre zusammenarbeiten, die Regel. Es ist offensichtlich, dass ein Scheitern solcher Projekte mit enormen Kosten verbunden ist. Einige Beispiele mögen dies verdeutlichen.

1. Am 9. Juni 1996 stürzte die Rakete Ariane 5 auf ihrem Jungfernflug ab. Ursache war ein Kette von Software-Fehlern: Ein Sensor im Navigations-System der Ariane 5 misst die horizontale Neigung und speichert diese zunächst als Gleitkomma-Zahl mit einer Genauigkeit von 64 Bit ab. Später wird dieser Wert dann in eine 16 Bit Festkomma-Zahl konvertiert. Bei dieser Konvertierung trat ein Überlauf ein, da die zu konvertierende Zahl zu groß war, um als 16 Bit Festkomma-Zahl dargestellt werden zu können. In der Folge gab das Navigations-System auf dem Datenbus, der dieses System mit der Steuerungs-Einheit verbindet, eine Fehlermeldung aus. Die Daten dieser Fehlermeldung wurden von der Steuerungs-Einheit als Flugdaten interpretiert. Die Steuer-Einheit leitete daraufhin eine Korrektur des Fluges ein, die dazu führte, dass die Rakete auseinander brach und die automatische Selbstzerstörung eingeleitet werden mußte. Die Rakete war mit 4 Satelliten beladen. Der wirtschaftliche Schaden, der durch den Verlust dieser Satelliten entstanden ist, lag bei mehreren 100 Millionen Dollar.

Ein vollständiger Bericht über die Ursache des Absturzes des Ariane 5 findet sich im Internet unter der Adresse

<http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html>

2. Die Therac 25 ist ein medizinisches Bestrahlungs-Gerät, das durch Software kontrolliert wird. Durch Fehler in dieser Software erhielten mindestens 6 Patienten eine Überdosis an Strahlung. Drei dieser Patienten sind an den Folgen dieser Überdosierung gestorben.

Einen detaillierten Bericht über diese Unfälle finden Sie unter

[http://courses.cs.vt.edu/~cs3604/lib/Therac\\_25/Therac\\_1.html](http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html)

3. Im ersten Golfkrieg konnte eine irakische *Scud* Rakete von dem *Patriot* Flugabwehrsystem aufgrund eines Programmier-Fehlers in der Kontrollsoftware des Flugabwehrsystems nicht abgefangen werden. 28 Soldaten verloren dadurch ihr Leben.

4. Im Internet finden Sie unter

<http://www.cs.tau.ac.il/~nachumd/horror.html>

eine Auflistung von schweren Unfällen, die auf Software-Fehler zurückgeführt werden konnten.

Diese Beispiele zeigen, dass bei der Konstruktion von IT-Systemen mit großer Sorgfalt und Präzision gearbeitet werden sollte. Die Erstellung von IT-Systemen muß auf einer wissenschaftlich fundierten Basis erfolgen, denn nur dann ist es möglich, die Korrektheit solcher Systeme zu *verifizieren*, also mathematisch zu beweisen. Diese oben geforderte wissenschaftliche Basis für die Entwicklung von IT-Systemen ist die Informatik, und diese hat ihre Wurzeln in der mathematischen Logik, mit der wir uns im ersten Semester des Informatik-Studiums beschäftigen.

Es gibt noch einen anderen praktischen Grund der es erforderlich macht, sich mit der mathematischen Logik zu beschäftigen. Dies ist die Programmier-Sprache *Prolog*, die vorwiegend im Bereich der KI (Künstliche Intelligenz) eingesetzt wird. Der Name *Prolog* steht für *Programming in Logic* und in der Tat bildet ein spezieller automatischer Beweiser den Kern eines *Prolog*-Interpreters. Für eine erfolgreiche Beherrschung dieser Sprache sind daher fundierte Kenntnisse in der mathematischen Logik unabdingbar.

Und auch der für Sie wichtigste Grund, sich mit der mathematischen Logik intensiv zu beschäftigen, soll hier nicht verschwiegen werden. Die Ausbildung der BA verfolgt neben dem Praxisbezug auch einen wissenschaftlichen Anspruch. Um diesem gerecht zu werden, wurde die Behandlung der mathematischen Logik in dem Curriculum der BA verbindlich festgelegt.

**Überblick über den Rest der Vorlesung.** Im Rest dieses Kapitels werden wird zunächst den Begriff der *prädikatenlogischen Formel* auf einer informellen Ebene einführen. Auf dieser Ebene werden wir prädikatenlogische Formeln zunächst nur als Abkürzungen vorstellen: Die Sprache der Prädikaten-Logik bietet uns als erstes einen Weg, komplizierte Zusammenhänge prägnanter und kürzer darzustellen, als dies mit den Mitteln der natürlichen Sprache möglich ist. Um allerdings die Bedeutung prädikatenlogischer Formeln mathematisch präzisieren zu können, benötigen wir einige Grundbegriffe aus der Mengenlehre, mit der wir uns im Rest des ersten Kapitels beschäftigen.

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 SETL2 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 SETL2 möglich, Algorithmen in der Sprache der Mengenlehre zu formulieren. Solche Algorithmen sind zwar oft nicht effizient im Vergleich zu klassischen Implementierungen, aber dafür oft wesentlich kürzer (und damit schneller zu implementieren) als ein entsprechendes C Programm. Zusätzlich hat SETL2 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*. Diese kann als ein Teil der Prädikaten-Logik aufgefasst werden. 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.

Im vierten Kapitel behandeln wir die Prädikatenlogik und analysieren den Begriff des prädikatenlogischen Beweises mit Hilfe eines *Kalküls*. Ein *Kalkül* ist dabei ein Verfahren, einen mathematischen Beweis zu führen. Dieses Verfahren läßt sich programmieren und bildet dann die Grundlage für die Programmier-Sprache *Prolog*, deren Grundzüge wir im fünften Kapitel skizzieren. Ähnlich wie SETL2 ist auch *Prolog* eine sogenannte *deklarative* Programmier-Sprache. Mit dem Attribut "*deklarativ*" soll dabei zum Ausdruck gebracht werden, dass in einem *Prolog*- oder SETL2-Programm eigentlich nur das zu lösende Problem beschrieben wird, während die Details der Abarbeitung nicht spezifiziert werden. In der Folge sind solche Programme im allgemeinen nicht sehr effizient, aber dafür sind sie oft wesentlich kürzer und dadurch schneller zu entwickeln als Programme in klassischen Programmier-Sprachen.

Im letzten Kapitel beschäftigen wir uns abschließend mit den Grenzen der Berechenbarkeit. Unter anderem werden wir zeigen, dass die Frage, ob ein Programm für alle Eingaben terminiert, im allgemeinen nicht beantwortet werden kann. Wir werden darüber hinaus einige ähnliche Probleme, die ebenfalls unentscheidbar sind, vorstellen.

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 BA 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 daher 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 Ideen Buchstaben einzeln lesen, sondern das Wort als Gesamtes. Wie sie sehen, ist das tatsächlich der Fall. Ehrt krsas, oder nicht? :-)

## 1.2 Prädikatenlogische Formeln

Der Begriff der *prädikatenlogischen Formel* wird in dieser Vorlesung eine zentrale Rolle spielen. Um die Bedeutung prädikatenlogischer Formeln definieren zu können, benötigen wir einige Begriffe aus der *Mengenlehre*. Andererseits werden zur Einführung der Mengenlehre ebenfalls prädikatenlogische Formeln benötigt. Wir werden daher den Begriff einer prädikatenlogische Formel in zwei Stufen einführen: Zunächst werden wir auf einer **informalen** Ebene prädikatenlogische Formeln einfach als *Abkürzungen* definieren. Um eine mathematische Präzisierung der Semantik machen wir uns an dieser Stelle noch keine Gedanken. Dieser Umgang mit Formeln entspricht dem Gebrauch von Formeln, wie er in der Mathematik üblich ist. Dieser Rahmen wird uns für die Einführung der Mengenlehre im nächsten Abschnitt ausreichen. In den späteren Kapiteln werden wir dann die Definition der prädikatenlogischen Formel mit den Mitteln der Mengenlehre präzisieren.

### 1.2.1 Warum Formeln

Betrachten wir einmal den folgenden mathematischen Text:

*Addieren wir zwei Zahlen und bilden dann das Quadrat dieser Summe, so ist das Ergebnis das selbe, wie wenn wir zunächst beide Zahlen einzeln quadrieren, summieren und dann das Produkt der beiden Zahlen zweifach hinzu addieren.*

Der mathematische Satz, der hier ausgedrückt wird, ist Ihnen aus der Schule bekannt, es handelt sich um den ersten Binomischen Satz. Um dies zu sehen, führen wir für die in dem Text genannten zwei Zahlen die Variablen  $a$  und  $b$  ein und übersetzen dann die in dem obigen Text auftretenden Teilsätze in Terme. Die folgende Tabelle zeigt diesen Prozeß:

Addieren wir zwei Zahlen	$a + b$
bilden das Quadrat dieser Summe	$(a + b)^2$
beide Zahlen einzeln quadrieren	$a^2, b^2$
summieren	$a^2 + b^2$
das Produkt der beiden Zahlen ...	$a * b$
... zweifach hinzu addieren	$a^2 + b^2 + 2 * a * b$

Insgesamt finden wir so, dass der obige Text zu der folgenden Formel äquivalent ist:

$$(a + b)^2 = a^2 + b^2 + 2 * a * b.$$

Für den mathematisch Geübten ist diese Formel offensichtlich leichter zu verstehen als der oben angegebene Text. Aber die Darstellung von mathematischen Zusammenhängen durch Formeln bietet neben der verbesserten Lesbarkeit noch zwei weitere Vorteile:

1. Formeln sind *manipulierbar*, d. h. wir können mit Formeln *rechnen*. Außerdem lassen Formeln sich aufgrund ihrer vergleichsweise einfachen Struktur auch mit Hilfe von Programmen bearbeiten und analysieren. Beim heutigen Stand der Technik ist es hingegen nicht möglich, natürlichsprachlichen Text mit dem Rechner vollständig zu analysieren und zu verstehen.
2. Darüber hinaus lässt sich die Bedeutung von Formeln mathematisch definieren und steht damit zweifelsfrei fest. Eine solche mathematische Definition der Bedeutung ist für natürlichsprachlichen Text so nicht möglich, da natürlichsprachlicher Text oft mehrdeutig ist und die genaue Bedeutung nur aus dem Zusammenhang hervorgeht.

### 1.2.2 Formeln als Kurzschreibweise

Nach dieser kurzen Motivation führen wir zunächst Formeln als Abkürzungen ein und stellen der Reihe nach die Ingredienzen vor, die wir zum Aufbau einer Formel benötigen.

#### 1. Variablen

Variablen dienen uns als Namen für verschiedenen Objekte. Oben haben wir beispielsweise für die beiden zu addierenden Zahlen die Variablen  $a$  und  $b$  eingeführt. Die Idee bei der Einführung einer Variable ist, dass diese ein Objekt bezeichnet, dessen Identität noch nicht feststeht.

#### 2. Konstanten

Konstanten bezeichnen Objekte, deren Identität schon feststeht. In der Mathematik werden beispielsweise Zahlen wie 1 oder  $\pi$  als Konstanten verwendet. Würden wir Aussagen über den biblischen Stammbaum als Formeln darstellen, so würden wir Adam und Eva als Konstanten verwenden.

Dieses letzte Beispiel mag Sie vielleicht verwundern, weil Sie davon ausgehen, dass Formeln nur dazu benutzt werden, mathematische oder allenfalls technische Zusammenhänge zu beschreiben. Der logische Apparat ist aber keineswegs auf eine Anwendung in diesen Bereichen beschränkt. Gerade auch Sachverhalte aus dem täglichen Leben lassen sich mit Hilfe von Formeln präzise beschreiben. Das ist auch notwendig, denn wir wollen ja später unsere Formeln zur Analyse von Programmen benutzen und diese Programme können sich durchaus auch mit der Lösung von Problemen beschäftigen, die ihren Ursprung außerhalb der Technik haben.

Variablen und Konstanten werden zusammenfassend auch als *atomare Terme* bezeichnet. Das Attribut *atomar* bezieht sich hierbei auf die Tatsache, dass diese Terme sich nicht weiter in Bestandteile zerlegen lassen. Im Gegensatz dazu stehen die *zusammengesetzten Terme*. Dies sind Terme, die mit Hilfe von Funktions-Zeichen aus anderen Termen aufgebaut werden.

#### 3. Funktions-Zeichen

Funktions-Zeichen benutzen wir, um aus Variablen und Konstanten neue Ausdrücke aufzubauen, die wiederum Objekte bezeichnen. In dem obigen Beispiel haben wir das Funktions-Zeichen “+” benutzt und mit diesem Funktions-Zeichen aus den Variablen  $a$  und  $b$  den

Ausdruck  $a + b$  gebildet. Allgemein nennen wir Ausdrücke, die sich aus Variablen, Konstanten und Funktions-Zeichen bilden lassen, *Terme*.

Das Funktions-Zeichen “+” ist zweistellig, aber natürlich gibt es auch einstellige und mehrstellige Funktions-Zeichen. Ein Beispiel aus der Mathematik für ein einstelliges Funktions-Zeichen ist das Zeichen “ $\sqrt{\phantom{x}}$ ”. Ein weiteres Beispiel ist durch das Zeichen “sin” gegeben, dass in der Mathematik für die Sinus-Funktion verwendet wird.

Allgemein gilt: Ist  $f$  ein  $n$ -stelliges Funktions-Zeichen und sind  $t_1, \dots, t_n$  Terme, so kann mit Hilfe des Funktions-Zeichen  $f$  daraus der neue Term

$$f(t_1, \dots, t_n)$$

gebildet werden. Diese Schreibweise, bei der zunächst das Funktions-Zeichen gefolgt von einer öffnenden Klammer angegeben wird und anschließend die Argumente der Funktion durch Kommata getrennt aufgelistet werden, gefolgt von einer schließenden Klammer, ist der “Normalfall”. Diese Notation wird auch als Präfix-Notation bezeichnet. Bei einigen zweistelligen Funktions-Zeichen hat es sich aber eingebürgert, diese in einer *Infix-Notation* darzustellen, d. h. solche Funktions-Zeichen werden zwischen die Terme geschrieben. In der Mathematik liefern die Funktions-Zeichen “+”, “-”, “\*” und “/” hierfür Beispiele.

#### 4. Prädikate

Prädikate stellen zwischen verschiedenen Objekten eine Beziehung her. Ein wichtiges Prädikat ist das Gleichheits-Prädikat, dass durch das Gleichheits-Zeichen “=” dargestellt wird. Setzen wir zwei Terme  $t_1$  und  $t_2$  durch das Gleichheits-Zeichen in Beziehung, so erhalten wir die *Formel*  $t_1 = t_2$ .

Genau wie Funktions-Zeichen auch hat jedes Prädikat eine vorgegebene *Stelligkeit*. Diese gibt an, wie viele Objekte durch das Prädikat in Relation gesetzt werden. Im Falle des Gleichheits-Zeichens ist die Stelligkeit 2, aber es gibt auch Prädikate mit anderen Stelligkeiten. Zum Beispiel könnten wir ein Prädikat “istQuadrat” definieren, dass für natürliche Zahlen ausdrückt, dass diese Zahl eine Quadrat-Zahl ist. Ein solches Prädikat wäre dann einstellig.

Ist allgemein  $p$  ein  $n$ -stelliges Prädikats-Zeichen und sind die Ausdrücke  $t_1, \dots, t_n$  Terme, so kann aus diesen Bestandteilen die *Formel*

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

gebildet werden. Formeln von dieser Bauart bezeichnen wir auch als *atomare Formel*, denn sie ist zwar aus Termen, nicht jedoch aus anderen Formeln zusammengesetzt.

Genau wie bei zweistelligen Funktions-Zeichen hat sich auch bei zweistelligen Prädikats-Zeichen eine *Infix-Notation* eingebürgert. Das Prädikats-Zeichen “=” liefert ein Beispiel hierfür, denn wir schreiben “ $a = b$ ” statt “ $= (a, b)$ ”. Andere Prädikats-Zeichen, für die sich eine Infix-Notation eingebürgert hat, sind die Prädikats-Zeichen “<”, “≤”, “>” und “≥”, die zum Vergleich von Zahlen benutzt werden.

#### 5. Junktoren

Junktoren werden dazu benutzt, Formeln mit einander in Beziehung zu setzen. Der einfachste Junktor ist das “und”. Haben wir zwei Formeln  $F_1$  und  $F_2$  und wollen ausdrücken, dass sowohl  $F_1$  als auch  $F_2$  gültig ist, so schreiben wir

$$F_1 \wedge F_2$$

und lesen dies als “ $F_1$  und  $F_2$ ”. Die nachfolgende Tabelle listet alle Junktoren auf, die wir verwenden werden:



Junktor	Bedeutung
$\neg F$	nicht $F$
$F_1 \wedge F_2$	$F_1$ und $F_2$
$F_1 \vee F_2$	$F_1$ oder $F_2$
$F_1 \rightarrow F_2$	wenn $F_1$ , dann $F_2$
$F_1 \leftrightarrow F_2$	$F_1$ genau dann, wenn $F_2$

Hier ist noch zu bemerken, dass es bei komplexeren Formeln notwendig ist, diese geeignet zu klammern um Mehrdeutigkeiten zu vermeiden. Bezeichnen beispielsweise  $P$ ,  $Q$  und  $R$  atomare Formeln, so können wir unter Zuhilfenahme von Klammern daraus die folgenden Formeln bilden:

$$P \rightarrow (Q \vee R) \quad \text{und} \quad (P \rightarrow Q) \vee R.$$

Umgangssprachlich würden beide Formeln wie folgt interpretiert:

*Aus  $P$  folgt  $Q$  oder  $R$ .*

Offensichtlich ist die mathematische Schreibweise hier klarer.

Die Verwendung von vieler Klammern vermindert die Lesbarkeit einer Formel. Um Klammern einsparen zu können, vereinbaren wir daher ähnliche Bindungsregeln, wie wir sie aus der Schulmathematik kennen. Dort sagt man, dass “+” und “−” schwächer binden als “\*” und “/” und meint damit, dass

$$x + y * z \quad \text{als} \quad x + (y * z)$$

gelesen wird. Ähnlich vereinbaren wir hier, dass “ $\neg$ ” stärker bindet als “ $\wedge$ ” und “ $\vee$ ” und dass diese beiden Operatoren stärker binden als “ $\rightarrow$ ”. Schließlich bindet der Operator “ $\leftrightarrow$ ” schwächer als alle anderen Operatoren. Mit dieser Vereinbarung lautet die Formel

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

dann in einer vollständig geklammerten Schreibweise

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

6. *Quantoren* geben an, in welcher Weise eine Variable in einer Formel verwendet wird. Wir kennen zwei Quantoren, den All-Quantor “ $\forall$ ” und den Existenz-Quantor “ $\exists$ ”. Eine Formel der Form

$$\forall x : F$$

lesen wir als “für alle  $x$  gilt  $F$ ” und eine Formel der Form

$$\exists x : F$$

wird als “es gibt ein  $x$ , so dass  $F$  gilt” gelesen. In dieser Vorlesung werden wir üblicherweise *qualifizierte Quantoren* verwenden. Die Qualifizierung gibt dabei an, welchem Bereich die durch die Variablen bezeichneten Objekte entspringen müssen. Im Falle des All-Quantors schreiben wir dann

$$\forall x \in Q : F$$

und lesen dies als “für alle  $x$  aus  $Q$  gilt  $F$ ”. Dies ist nur abkürzende Schreibweise, die wir wie folgt definieren können:

$$\forall x \in Q : F \stackrel{\text{def}}{\iff} \forall x : (x \in Q \rightarrow F)$$

Entsprechend lautet die Notation für den Existenz-Quantor

$$\exists x \in Q : F$$

und das wird dann als “es gibt ein  $x$  aus  $Q$ , so dass  $F$  gilt” gelesen. Formal läßt sich das als

$$\exists x \in Q : F \stackrel{\text{def}}{\iff} \exists x : (x \in Q \wedge F)$$

definieren. Wir verdeutlichen die Schreibweisen durch eine Beispiel. Die Formel

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

lesen wir wie folgt:

*Für alle  $x$  aus  $\mathbb{R}$  gilt: Es gibt ein  $n$  aus  $\mathbb{N}$ , so dass  $n$  größer als  $x$  ist.*

Hier steht  $\mathbb{R}$  für die reellen Zahlen und  $\mathbb{N}$  bezeichnet die natürlichen Zahlen. Die obige Formel drückt also aus, dass es zu jeder reellen Zahl  $x$  eine natürliche Zahl  $n$  gibt, so dass  $n$  größer als  $x$  ist.

Treten in einer Formel Quantoren und Junktoren gemischt auf, so stellt sich die Frage, was stärker bindet. Wir vereinbaren, dass Quantoren schwächer binden als Junktoren. Eine Formel der Art

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

wird also implizit wie folgt geklammert:

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

### 1.2.3 Beispiele für Terme und Formeln

Um die Konzepte “Term” und “Formel” zu verdeutlichen, geben wir im folgenden einige Beispiele an. Um die Erklärungen einfach halten zu können, wollen wir ein Beispiel aus dem täglichen Leben verwenden und Terme und Formeln angeben, die sich mit Verwandtschaftsbeziehungen beschäftigen. Wir beginnen damit, dass wir die Konstanten, Variablen, Funktions-Zeichen und Prädikats-Zeichen festlegen.

1. Als *Konstanten* verwenden wir die Wörter

“adam”, “eva”, “kain” und “abel”, “lisa”.

2. Als *Variablen* verwenden wir die Buchstaben

“ $x$ ”, “ $y$ ” und “ $z$ ”.

3. Als *Funktions-Zeichen* verwenden wir die Wörter

“vater” und “mutter”.

Diese beiden Funktions-Zeichen sind einstellig.

4. Als *Prädikats-Zeichen* verwenden wir die Wörter

“bruder”, “schwester”, “onkel”, “männlich” und “weiblich”.

Alle diese Prädikats-Zeichen sind zweistellig. Als weiteres zweistelliges Prädikats-Zeichen verwenden wir das Gleichheits-Zeichen “=”. Außerdem verwenden wir das einstellige Prädikats-Zeichen “männlich”.

Eine solche Ansammlung von Konstanten, Variablen, Funktions-Zeichen und Prädikats-Zeichen bezeichnen wir auch als *Signatur*. Wir geben zunächst einige Terme an, die sich mit dieser Signatur bilden lassen:

1. “kain” ist ein Term, denn “kain” ist eine Konstante.
2. “vater(kain)” ist ein Term, denn “kain” ist ein Term und “vater” ist ein einstelliges Funktions-Zeichen.
3. “mutter(vater(kain))” ist ein Term, denn “vater(kain)” ist ein Term und “mutter” ist ein einstelliges Funktions-Zeichen,
4. “männlich(kain)” ist eine Formel, denn “kain” ist ein Term und “männlich” ist ein einstelliges Prädikats-Zeichen.

5. “**männlich(lisa)**” ist ebenfalls eine Formel, denn “**lisa**” ist ein Term.

Dieses Beispiel zeigt, dass Formeln durchaus auch falsch sein können. Die bisher gezeigten Formeln sind alle atomar. Wir geben nun Beispiele für zusammengesetzte Formeln.

6. “**vater**( $x$ ) = **vater**( $y$ )  $\wedge$  **mutter**( $x$ ) = **mutter**( $y$ )  $\rightarrow$  **bruder**( $x, y$ )  $\vee$  **schwester**( $x, y$ )”

ist eine Formel, die aus den beiden Formeln

$$\text{“vater}(x) = \text{vater}(y) \wedge \text{mutter}(x) = \text{mutter}(y)\text{”} \quad \text{und}$$

$$\text{“bruder}(x, y) \vee \text{schwester}(x, y)\text{”}$$

aufgebaut ist.

7. “ $\forall x: \forall y: (\text{bruder}(x, y) \vee \text{schwester}(x, y))$ ” ist eine Formel.

Die Formel Nr. 6 ist intuitiv gesehen wahr, während wir die Formel Nr. 7 als falsch erkennen. Um die Begriffe “*wahr*” und “*falsch*” für Formeln streng definieren zu können, ist es notwendig, den Begriff der *Interpretation* der verwendeten Signatur zu definieren. Anschaulich gesehen legt eine *Interpretation* die Bedeutung der Symbole aus der Signatur fest. Exakt kann der Begriff aber erst angegeben werden, wenn Hilfsmittel aus der Mengenlehre zur Verfügung stehen. Dieser wenden wir uns jetzt zu.

## 1.3 Mengen und Relationen

Die Mengenlehre ist gegen Ende des 19-ten Jahrhunderts aus dem Bestreben heraus entstanden, die Mathematik auf eine solide Grundlage zu stellen. Die Schaffung einer solchen Grundlage wurde als notwendig erachtet, da der Begriff der Unendlichkeit den Mathematikern zunehmend Kopfzerbrechen bereitete.

Begründet wurde die Mengenlehre in wesentlichen Teilen von Georg Cantor (1845 – 1918). Die erste Definition des Begriffs der Menge lautete etwa wie folgt: Eine *Menge* ist eine *wohldefinierte* Ansammlung von *Elementen*. Das Attribut “*wohldefiniert*” drückt dabei aus, dass wir für eine vorgegebene Menge  $M$  und ein Element  $x$  stets *entscheiden* können, ob das Element  $x$  zu der Menge  $M$  gehört oder nicht. In diesem Fall schreiben wir

$$x \in M$$

und lesen diese Formel als “ $x$  ist ein Element der Menge  $M$ ”. Das Zeichen “ $\in$ ” wird in der Mengenlehre also als zweistelliges Prädikats-Zeichen gebraucht, für das sich eine Infix-Notation eingebürgert hat. Um den Begriff der *wohldefinierten Ansammlung von Elementen* mathematisch zu präzisieren, führte Cantor das sogenannte *Komprehensions-Axiom* ein. Wir können dieses zunächst wie folgt formalisieren: Ist  $p(x)$  eine Eigenschaft, die ein Objekt  $x$  entweder hat oder nicht, so können wir die Menge  $M$  aller Objekte, welche die Eigenschaft  $p(x)$  haben, bilden. Wie schreiben dann

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

und lesen dies als “ $M$  ist die Menge aller  $x$ , auf welche die Eigenschaft  $p(x)$  zutrifft”. Eine Eigenschaft  $p(x)$  ist dabei nichts anderes als eine Formel, in der die Variable  $x$  vorkommt. Wir veranschaulichen das Komprehensions-Axiom durch ein Beispiel: Es sei  $\mathbb{N}$  die Menge der natürlichen Zahlen. Ausgehend von der Menge  $\mathbb{N}$  wollen wir die Menge der *geraden Zahlen* definieren. Zunächst müssen wir dazu die Eigenschaft einer Zahl  $x$ , *gerade* zu sein, durch eine Formel  $p(x)$  mathematisch erfassen. Eine natürliche Zahl  $x$  ist genau dann gerade, wenn es eine natürliche Zahl  $y$  gibt, so dass  $x$  das Doppelte von  $y$  ist. Damit können wir die Eigenschaft  $p(x)$  folgendermaßen definieren:

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

Also kann die Menge der geraden Zahlen als

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

geschrieben werden.

Leider führt die uneingeschränkte Anwendung des Komprehensions-Axiom schnell zu Problemen. Betrachten wir dazu die Eigenschaft einer Menge, sich selbst zu enthalten, wir setzen also  $p(x) := \neg x \in x$  und definieren die Menge  $R$  als

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

Intuitiv würden wir vielleicht erwarten, dass keine Menge sich selbst enthält. Wir wollen jetzt zunächst für die eben definierte Menge  $R$  überprüfen, wie die Dinge liegen. Es können zwei Fälle auftreten:

1. Fall:  $\neg R \in R$ . Dann trifft die Eigenschaft, die die Menge  $R$  definiert, auf  $R$  zu und wir haben  $R \in \{x \mid \neg x \in x\}$ . Da  $R = \{x \mid \neg x \in x\}$  heißt dass gerade  $R \in R$  und steht im Widerspruch zur Voraussetzung  $\neg R \in R$ .
2. Fall:  $R \in R$ . Setzen wir hier die Definition von  $R$  ein, so haben wir

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

Dass heißt dann aber gerade  $\neg R \in R$  und steht im Widerspruch zur Voraussetzung  $R \in R$ .

Wie wir es auch drehen und wenden, es kann weder  $R \in R$  noch  $\neg R \in R$  gelten. Als Ausweg können wir nur feststellen, dass das vermittle

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

definierte Objekt keine Menge ist. Das heißt dann aber, dass das Komprehensions-Axiom zu allgemein ist. Wir stellen also fest, dass nicht jede in der Form

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

angegebene Menge existieren muss. Die Konstruktion der Menge " $\{x \mid \neg x \in x\}$ " stammt von dem britischen Logiker und Philosophen Bertrand Russell (1872 – 1970). Sie wird deswegen auch als *Russell'sche Antinomie* bezeichnet.

Um Paradoxien wie die Russell'sche Antinomie zu vermeiden, ist es erforderlich, bei der Konstruktion von Mengen vorsichtiger vorzugehen. Wir werden im folgenden Konstruktions-Prinzipien für Mengen vorstellen, die schwächer sind als das Komprehensions-Axiom, die aber für die Praxis der Informatik ausreichend sind. Wir wollen dabei die dem Komprehensions-Axiom zugrunde liegende Notation beibehalten und Mengen  $M$  in der Form

$$M = \{x \mid p(x)\} \tag{*}$$

notieren. Um Paradoxien zu vermeiden, werden wir nur bestimmte Sonderfälle der Gleichung (\*) zulassen. Diese Sonderfälle stellen wir jetzt vor.

### 1.3.1 Erzeugung von Mengen durch explizites Auflisten

Die einfachste Möglichkeit, eine Menge festzulegen, besteht in der expliziten *Auflistung* aller ihrer Elemente. Diese Elemente werden in den geschweiften Klammern "{" und "}" eingefaßt und durch Kommas getrennt. Definieren wir beispielsweise

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

so haben wir damit festgelegt, dass die Menge  $M$  aus den Elementen 1, 2 und 3 besteht. In der Schreibweise des Komprehensions-Axioms können wir diese Menge als

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

angeben.

Als ein weiteres Beispiel für eine Menge, die durch explizite Aufzählung ihrer Elemente angegeben werden kann, betrachten wir die Menge der kleinen Buchstaben, die wir wie folgt definieren:

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

### 1.3.2 Die Menge der natürlichen Zahlen

Alle durch explizite Auflistung definierten Mengen haben offensichtlich nur endlich viele Elemente. Aus der mathematischen Praxis kennen wir aber auch Mengen mit unendlich vielen Elementen.

Ein Beispiel ist die Menge der natürlichen Zahlen, die wir mit  $\mathbb{N}$  bezeichnen. Mit den bisher behandelten Verfahren läßt diese Menge sich nicht definieren. Wir müssen daher die Existenz dieser Menge als Axiom fordern:

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

Neben der Menge  $\mathbb{N}$  der natürlichen Zahlen verwenden wir noch die folgenden Mengen von Zahlen:

1.  $\mathbb{Z}$ : Menge der ganzen Zahlen.
2.  $\mathbb{Q}$ : Menge der rationalen Zahlen.
3.  $\mathbb{R}$ : Menge der reellen Zahlen.

In der Mathematik wird gezeigt, wie sich diese Mengen aus der Menge der natürlichen Zahlen ableiten lassen.

### 1.3.3 Das Auswahl-Prinzip

Das *Auswahl-Prinzip* ist eine Abschwächung des Komprehensions-Axiom. Die Idee ist, mit Hilfe einer Eigenschaft  $p$  aus einer schon vorhandenen Menge  $M$  die Menge  $N$  der Elemente  $x$  *auszuwählen*, die eine bestimmte Eigenschaft  $p(x)$  besitzen:

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

In der Notation des Komprehensions-Axioms schreibt sich diese Menge als

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

Im Unterschied zu dem Komprehensions-Axiom können wir uns hier nur auf die Elemente einer bereits vorgegebenen Menge  $M$  beziehen und nicht auf völlig beliebige Objekte.

**Beispiel:** Die Menge der geraden Zahlen kann mit dem Auswahl-Prinzip wie folgt definiert werden:

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

### 1.3.4 Potenz-Mengen

Um den Begriff der *Potenz-Menge* einführen zu können, müssen wir zunächst den Begriff einer *Teilmenge* festlegen. Sind  $M$  und  $N$  zwei Mengen, so heißt  $M$  eine *Teilmenge* von  $N$  genau dann, wenn jedes Element der Menge  $M$  auch ein Element der Menge  $N$  ist. In diesem Fall schreiben wir  $M \subseteq N$ . Formal können wir den Begriff der Teilmenge also wie folgt einführen:

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

Unter der *Potenz-Menge* einer Menge  $M$  wollen wir nun die Menge aller Teilmengen von  $M$  verstehen. Wir schreiben  $2^M$  für die Potenz-Menge von  $M$ . Dann gilt:

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

**Beispiel:** Wir bilden die Potenz-Menge der Menge  $\{1, 2, 3\}$ . Es gilt:

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

Diese Menge hat  $8 = 2^3$  Elemente. Allgemein kann durch Induktion gezeigt werden, dass die Potenz-Menge einer Menge  $M$ , die aus  $m$  verschiedenen Elementen besteht, insgesamt  $2^m$  Elemente enthält. Dies erklärt die Schreibweise  $2^M$  für die Potenz-Menge von  $M$ .

### 1.3.5 Vereinigungs-Mengen

Sind zwei Mengen  $M$  und  $N$  gegeben, so enthält die Vereinigung von  $M$  und  $N$  alle Elemente, die in der Menge  $M$  oder in der Menge  $N$  liegen. Für diese Vereinigung schreiben wir  $M \cup N$ . Formal kann die Vereinigung wie folgt definiert werden:

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

**Beispiel:** Ist  $M = \{1, 2, 3\}$  und  $N = \{2, 5\}$ , so gilt:

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

Der Begriff der Vereinigung von Mengen lässt sich verallgemeinern. Betrachten wir dazu eine Menge  $X$ , deren Elemente selbst wieder Mengen sind. Beispielsweise ist die Potenz-Menge einer Menge von dieser Art. Wir können dann die Vereinigung aller Mengen, die Elemente von der Menge  $X$  sind, bilden. Diese Vereinigung schreiben wir als  $\bigcup X$ . Formal kann das wie folgt definiert werden:

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

**Beispiel:** Die Menge  $X$  sei wie folgt gegeben:

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

Dann gilt

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

### 1.3.6 Schnitt-Menge

Sind zwei Mengen  $M$  und  $N$  gegeben, so definieren wir den *Schnitt* von  $M$  und  $N$  als die Menge aller Elemente, die sowohl in  $M$  als auch in  $N$  auftreten. Wir bezeichnen den Schnitt von  $M$  und  $N$  mit  $M \cap N$ . Formal können wir  $M \cap N$  wie folgt definieren:

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

**Beispiel:** Wir berechnen den Schnitt der Mengen  $M = \{1, 3, 5\}$  und  $N = \{2, 3, 5, 6\}$ . Es gilt

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

### 1.3.7 Differenz

Sind zwei Mengen  $M$  und  $N$  gegeben, so bezeichnen wir die *Differenz* von  $M$  und  $N$  als die Menge aller Elemente, die in  $M$  aber nicht in  $N$  auftreten. Wir schreiben hierfür  $M \setminus N$ . Das wird als *M ohne N* gelesen und kann formal wie folgt definiert werden:

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

**Beispiel:** Wir berechnen die Differenz der Mengen  $M = \{1, 3, 5, 7\}$  und  $N = \{2, 3, 5, 6\}$ . Es gilt

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

Die beiden letzten Prinzipien zur Mengen-Bildung (Schnitt-Menge und Differenz-Menge) sind aus dem Auswahl-Prinzip herleitbar. Bei einer axiomatischen Definition der Mengenlehre werden diese Prinzipien daher weggelassen.

### 1.3.8 Bild-Mengen

Es sei  $M$  eine Menge und  $f$  sei eine Funktion, die für alle  $x$  aus  $M$  definiert ist. Dann heißt die Menge aller Abbilder  $f(x)$  von Elementen  $x$  aus der Menge  $M$  das *Bild* von  $M$  unter  $f$ . Wir schreiben  $f(M)$  für dieses Bild. Formal kann  $f(M)$  wie folgt definiert werden:

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

**Beispiel:** Die Menge  $Q$  aller Quadrat-Zahlen kann wie folgt definiert werden:

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

### 1.3.9 Kartesische Produkte

Um den Begriff des kartesischen Produktes einführen zu können, benötigen wir zunächst den Begriff des geordneten Paares zweier Objekte  $x$  und  $y$ . Dieses wird als

$$\langle x, y \rangle$$

geschrieben. Wir sagen, dass  $x$  die *erste Komponente* des Paares  $\langle x, y \rangle$  ist, und  $y$  ist die *zweite*

*Komponente.* Zwei geordnete Paare  $\langle x_1, y_1 \rangle$  und  $\langle x_2, y_2 \rangle$  sind genau dann gleich, wenn sie komponentenweise gleich sind, d.h. es gilt

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

Das kartesische Produkt zweier Mengen  $M$  und  $N$  ist nun die Menge aller geordneten Paare, deren erste Komponente in  $M$  liegt und deren zweite Komponente in  $N$  liegt. Das kartesische Produkt von  $M$  und  $N$  wird als  $M \times N$  geschrieben, formal gilt:

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

**Beispiel:** Wir setzen  $M = \{1, 2, 3\}$  und  $N = \{5, 7\}$ . Dann gilt

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

Der Begriff des geordneten Paares lässt sich leicht zum Begriff des  $n$ -Tupels verallgemeinern: Ein  $n$ -Tupel hat die Form

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

Analog kann auch der Begriff des kartesischen Produktes auf  $n$  Mengen  $M_1, \dots, M_n$  verallgemeinert werden. Das sieht dann so aus:

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

Ist  $f$  eine Funktion, die auf  $M_1 \times \dots \times M_n$  definiert ist, so vereinbaren wir folgende Vereinfachung der Schreibweise:

$$f(x_1, \dots, x_n) \text{ steht für } f(\langle x_1, \dots, x_n \rangle).$$

Gelegentlich werden  $n$ -Tupel auch als *endliche Folgen* oder auch als *Listen* bezeichnet.

### 1.3.10 Gleichheit von Mengen

Wir haben nun alle Verfahren, die wir zur Konstruktion von Mengen benötigen, vorgestellt. Wir klären jetzt die Frage, wann zwei Mengen gleich sind. Dazu postulieren wir das folgende *Extensionalitäts-Axiom* für Mengen:

*Zwei Mengen sind genau dann gleich, wenn sie die selben Elemente besitzen.*

Mathematisch können wir diesen Sachverhalt wie folgt ausdrücken:

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

Eine wichtige Konsequenz aus diesem Axiom ist die Tatsache, dass die Reihenfolge, mit der Elemente in einer Menge aufgelistet werden, keine Rolle spielt. Beispielsweise gilt

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

denn beide Mengen enthalten die selben Elemente.

Falls Mengen durch explizite Aufzählung ihrer Elemente definiert sind, ist die Frage nach der Gleichheit zweier Mengen trivial. Ist eine der Mengen mit Hilfe des Auswahl-Prinzips definiert, so kann es beliebig schwierig sein zu entscheiden, ob zwei Mengen gleich sind. Hierzu ein Beispiel: Für natürliche Zahlen  $n$  sei das Prädikat **fermat**( $n$ ) wie folgt definiert:

$$\mathbf{fermat}(n) \Leftrightarrow \exists x \in \mathbb{N} : \exists y \in \mathbb{N} : \exists z \in \mathbb{N} : x > 0 \wedge y > 0 \wedge x^n + y^n = z^n.$$

Definieren wir nun die Menge  $F$  als

$$F := \{n \in \mathbb{N} \mid \mathbf{fermat}(n)\},$$

so lässt sich zeigen, dass

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

gilt. Allerdings ist der Nachweis dieser Gleichheit sehr schwer, denn er ist äquivalent zum Beweis der *Fermat'schen Vermutung*. Diese Vermutung wurde 1637 von *Pierre de Fermat* aufgestellt und konnte erst 1995 von Andrew Wiles bewiesen werden. Es gibt andere, ähnlich aufgebaute Mengen, wo bis heute unklar ist, welche Elemente in der Menge liegen und welche nicht.

### 1.3.11 Rechenregeln für das Arbeiten mit Mengen

Vereinigungs-Menge, Schnitt-Menge und die Differenz zweier Mengen genügen Gesetzmäßigkeiten, die in den folgenden Gleichungen zusammengefaßt sind. Bevor wir diese Gesetze angeben, vereinbaren wir noch, die leere Menge  $\{\}$  mit  $\emptyset$  zu bezeichnen. Damit haben wir die folgenden Gesetze:

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

Wir beweisen exemplarisch die Gleichung  $K \setminus (M \cup N) = (K \setminus M) \cap (K \setminus N)$ . Um die Gleichheit zweier Mengen zu zeigen ist nachzuweisen, dass beide Mengen die selben Elemente enthalten. Wir haben die folgende Kette von Äquivalenzen:

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

Die übrigen Gleichungen können nach dem selben Schema hergeleitet werden. Ist  $M$  eine Menge und  $x$  ein Objekt, so vereinbaren wir  $x \notin M$  als Kurzschreibweise für die Formel  $\neg x \in M$ , formal:

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

## 1.4 Binäre Relationen

Binäre Relationen treten in der Informatik häufig auf. Es ist daher notwendig, sich mit diesen näher zu beschäftigen. Im nächsten Unterabschnitt werden wir zunächst das Verhältnis von binären Relationen und Funktionen beleuchten. Wir werden sehen, dass wir Funktionen als spezielle binäre Relationen auffassen können. Anschließend betrachten wir noch einige Spezialfälle binärer Relationen, die in der Praxis von Bedeutung sind.

### 1.4.1 Binäre Relationen und Funktionen

Ist eine Menge  $R$  als Teilmenge des kartesischen Produkts zweier Mengen  $M$  und  $N$  gegeben, gilt also

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

so bezeichnen wir  $R$  auch als *binäre Relation*. In diesem Fall definieren wir den *Definitions-Bereich* von  $R$  als

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

Entsprechend wird der *Werte-Bereich* von  $R$  als

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

definiert.



**Beispiel:** Es sei  $R = \{\langle 1, 1 \rangle, \langle 2, 4 \rangle, \langle 3, 9 \rangle\}$ . Dann gilt  
 $\text{dom}(R) = \{1, 2, 3\}$  und  $\text{rng}(R) = \{1, 4, 9\}$ .

**Links- und Rechts-Eindeutige Relationen** Wir nennen eine Relation  $R \subseteq M \times N$  *rechts-eindeutig*, wenn folgendes gilt:

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

Bei einer rechts-eindeutigen Relation  $R \subseteq M \times N$  gibt es also zu jedem  $x \in M$  höchstens ein  $y \in N$  so, dass  $\langle x, y \rangle \in R$  gilt. Entsprechend nennen wir eine Relation  $R \subseteq M \times N$  *links-eindeutig*, wenn gilt:

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

Bei einer links-eindeutigen Relation  $R \subseteq M \times N$  gibt es also zu jedem  $y \in N$  höchstens ein  $x \in M$  so, dass  $\langle x, y \rangle \in R$  gilt.

**Beispiele:** Es sei  $M = \{1, 2, 3\}$  und  $N = \{4, 5, 6\}$ .

1. Die Relation  $R_1$  sei definiert durch

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

Diese Relation ist nicht rechts-eindeutig, denn  $4 \neq 6$ . Die Relation ist links-eindeutig, denn die rechten Seiten aller in  $R_1$  auftretenden Tupel sind verschieden.

2. Die Relation  $R_2$  sei definiert durch

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

Diese Relation ist rechts-eindeutig, denn die linken Seiten aller in  $R_2$  auftretenden Tupel sind verschieden. Sie ist auch links-eindeutig, denn die rechten Seiten aller in  $R_2$  auftretenden Tupel sind verschieden.

3. Die Relation  $R_3$  sei definiert durch

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

Diese Relation ist rechts-eindeutig, denn die linken Seiten aller in  $R_3$  auftretenden Tupel sind verschieden. Sie ist nicht links-eindeutig, denn  $2 \neq 3$ .

**Totale Relationen** Eine binäre Relation  $R \subseteq M \times N$  heißt *links-total auf M*, wenn

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

gilt. Dann gibt es für alle  $x$  aus der Menge  $M$  ein  $y$  aus der Menge  $N$ , so dass  $\langle x, y \rangle$  in der Menge  $R$  liegt. Die Relation  $R_3$  aus dem obigen Beispiel ist links-total, denn jedem Element aus  $M$  wird durch  $R_3$  ein Element aus  $N$  zugeordnet.

Analog nennen wir eine binäre Relation  $R \subseteq M \times N$  *rechts-total auf N*, wenn

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

gilt. Dann gibt es für alle  $y$  aus der Menge  $N$  ein  $x$  aus der Menge  $M$ , so dass  $\langle x, y \rangle$  in der Menge  $R$  liegt. Die Relation  $R_3$  aus dem obigen Beispiel ist nicht rechts-total, denn dem Element 5 aus  $N$  wird durch  $R_3$  kein Element aus  $M$  zugeordnet, denn für alle  $\langle x, y \rangle \in R_3$  gilt  $y \neq 5$ .

**Funktionale Relationen** Eine Relation  $R \subseteq M \times N$ , die sowohl links-total auf  $M$  als auch rechts-eindeutig ist, nennen wir eine *funktionale* Relation auf  $M$ . Ist  $R \subseteq M \times N$  eine funktionale Relation, so können wir eine Funktion  $f_R: M \rightarrow N$  wie folgt definieren:

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

Diese Definition funktioniert, denn aus der Links-Totalität von  $R$  folgt, dass es für jedes  $x \in M$  auch ein  $y \in N$  gibt, so dass  $\langle x, y \rangle \in R$  ist. Aus der Rechts-Eindeutigkeit von  $R$  folgt dann, dass dieses  $y$  eindeutig bestimmt ist. Ist umgekehrt eine Funktion  $f: M \rightarrow N$  gegeben, so können wir dieser Funktion eine Relation  $\text{graph}(f) \subseteq M \times N$  zuordnen, die links-total und rechts-eindeutig ist, indem wir definieren:

$$\text{graph}(f) := \{z \mid \exists x \in M: \exists y \in N: z = \langle x, y \rangle \wedge y = f(x)\}.$$

Wir werden daher im folgenden alle Funktionen als spezielle binäre Relationen auffassen. Für die Menge aller Funktionen von  $M$  nach  $N$  schreiben wir auch  $N^M$ . Diese Schreibweise erklärt sich wie folgt: Sind  $M$  und  $N$  endliche Mengen mit  $m$  bzw.  $n$  Elementen, so gibt es genau  $n^m$  verschiedene Funktionen von  $M$  nach  $N$ . Wir werden daher funktionale Relationen und die entsprechenden Funktionen identifizieren. Damit ist dann für eine funktionale Relation  $R \subseteq M \times N$  und ein  $x \in M$  auch die Schreibweise  $R(x)$  zulässig: Mit  $R(x)$  bezeichnen wir das eindeutig bestimmte  $y \in N$ , für das  $\langle x, y \rangle \in R$  gilt.

### Beispiele:

1. Wir setzen  $M = \{1, 2, 3\}$ ,  $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$  und definieren

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

Dann ist  $R$  eine funktionale Relation auf  $M$ . Diese Relation berechnet gerade die Quadrat-Zahlen auf der Menge  $M$ .

2. Diesmal setzen wir  $M = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$  und  $N = \{1, 2, 3\}$  und definieren

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

Dann ist  $R$  keine funktionale Relation auf  $M$ , denn  $R$  ist nicht links-total auf  $M$ . Beispielsweise wird das Element 2 von der Relation  $R$  auf kein Element aus  $N$  abgebildet.

3. Wir setzen diesmal  $M = \{1, 2, 3\}$ ,  $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$  und definieren

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

Dann ist  $R$  keine funktionale Relation auf  $M$ , denn  $R$  ist nicht rechts-eindeutig auf  $M$ . Beispielsweise wird das Element 2 von der Relation  $R$  sowohl auf 3 als auch auf 4 abgebildet.

Ist  $R \subseteq M \times N$  eine binäre Relation und ist weiter  $X \subseteq M$ , so definieren wir das *Bild von  $X$  unter  $R$*  als

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

Besteht die Menge  $X$  nur aus einem Element, gilt also  $X = \{x\}$ , so schreiben wir  $R\{x\}$  anstelle von  $R(\{x\})$ , es gilt also

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

**Inverse Relation** Zu einer Relation  $R \subseteq M \times N$  definieren wir die *inverse* Relation  $R^{-1} \subseteq N \times M$  wie folgt:

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

Aus dieser Definition folgt sofort, dass  $R^{-1}$  rechts-eindeutig ist genau dann, wenn  $R$  links-eindeutig ist. Außerdem ist  $R^{-1}$  links-total genau dann, wenn  $R$  rechts-total ist. Ist eine Relation sowohl links-eindeutig als auch rechts-eindeutig und außerdem sowohl links-total als auch rechts-total, so nennen wir sie auch *bijektiv*. In diesem Fall läßt sich neben der Funktion  $f_R$  auch eine Funktion  $f_{R^{-1}}$  definieren. Die Definition der letzten Funktion lautet ausgeschrieben:

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

Diese Funktion ist dann aber genau die Umkehr-Funktion von  $f_R$ , es gilt

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

Dieser Umstand rechtfertigt im nachhinein die Schreibweise  $R^{-1}$ .

**Komposition von Relationen** Ähnlich wie wir Funktionen verknüpfen können, können auch Relationen verknüpft werden. Wir betrachten zunächst Mengen  $L$ ,  $M$  und  $N$ . Sind dort zwei Relationen  $R \subseteq L \times M$  und  $Q \subseteq M \times N$  definiert, so ist das *relationale Produkt*  $Q \circ R$  wie folgt definiert:

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

Das relationale Produkt von  $Q$  und  $R$  wird gelegentlich auch als die *Komposition* von  $Q$  und  $R$  bezeichnet.

**Beispiel:** Es sei  $L = \{1, 2, 3\}$ ,  $M = \{4, 5, 6\}$  und  $N = \{7, 8, 9\}$ . Weiter seien die Relationen  $Q$  und  $R$  wie folgt gegeben:

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

Dann gilt

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

**Eigenschaften des relationalen Produkts** Ist die Relation  $R \subseteq L \times M$  funktional auf  $L$  und ist die Relation  $Q \subseteq M \times N$  funktional auf  $M$ , so ist die Relation  $R \circ Q$  funktional auf  $L$  und es gilt

$$\forall x \in L: f_{R \circ Q}(x) = f_Q(f_R(x)).$$

Außerdem ist die Komposition von Relationen *assoziativ*: Sind

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

binäre Relationen, so gilt

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

Eine weitere wichtige Eigenschaft des relationalen Produkts ist die folgende: Sind zwei Relationen  $R \subseteq L \times M$  und  $Q \subseteq M \times N$  gegeben, so gilt

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

Beachten Sie, dass sich die Reihenfolgen von  $Q$  und  $R$  hier vertauschen. Zum Beweis ist zu zeigen, dass für alle Paare  $\langle z, x \rangle \in N \times L$  die Äquivalenz

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

gilt. Den Nachweis erbringen wir durch die folgende Kette von Äquivalenz-Umformungen:

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

Wir bemerken noch, dass das folgende Distributivgesetz gilt: Sind  $R_1$  und  $R_2$  Relationen auf  $L \times M$  und ist  $Q$  eine Relation auf  $M \times N$ , so gilt

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

Analog gilt ebenfalls

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

falls  $R$  eine Relation auf  $L \times M$  und  $Q_1$  und  $Q_2$  Relationen auf  $M \times N$  sind. Interessant ist noch zu bemerken, dass für den Schnitt von Relationen kein analoges Distributivgesetz gilt. Seien etwa die Relationen  $R_1$ ,  $R_2$  und  $Q$  wie folgt definiert:

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

Dann gilt

$$Q \circ R_1 = \{\langle 1, 4 \rangle\}, \quad Q \circ R_2 = \{\langle 1, 4 \rangle\}, \quad \text{also } (Q \circ R_1) \cap (Q \circ R_2) = \{\langle 1, 4 \rangle\},$$

aber andererseits haben wir

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

**Identische Relation** Ist  $M$  eine Menge, so definieren wir die *identische Relation*  $\text{id}_M \subseteq M \times M$  wie folgt:

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

**Beispiel:** Es sei  $M = \{1, 2, 3\}$ . Dann gilt

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

Aus der Definition folgt sofort

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

Sei weiterhin  $R \subseteq M \times N$  und eine binäre Relation, so gilt

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

### 1.4.2 Spezielle Relationen

Wir betrachten im folgenden den Spezialfall von Relationen  $R \subseteq M \times N$ , für den  $M = N$  gilt. Wir definieren: Eine Relation  $R \subseteq M \times M$  heißt eine Relation *auf* der Menge  $M$ . Im Rest dieses Abschnittes betrachten wir nur noch solche Relationen. Statt  $M \times M$  schreiben wir auch  $M^2$ .

Ist  $R$  eine Relation auf  $M$  und sind  $x, y \in M$ , so schreiben wir statt  $\langle x, y \rangle \in R$  gelegentlich auch  $x R y$ . Beispielsweise läßt sich die Relation  $\leq$  auf  $\mathbb{N}$  wie folgt definieren:

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

Statt  $\langle x, y \rangle \in \leq$  hat sich die Schreibweise  $x \leq y$  eingebürgert.

**Reflexive Relationen** Eine Relation  $R \subseteq M \times M$  ist *reflexiv* falls gilt:

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

Äquivalent dazu ist die Definition dass  $R$  genau dann reflexiv ist, wenn  $\text{id}_M \subseteq R$  gilt.

**Symmetrische Relationen** Eine Relation  $R \subseteq M \times M$  ist *symmetrisch* falls gilt:

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

Äquivalent dazu ist die Definition dass  $R$  genau dann symmetrisch ist, wenn  $R^{-1} \subseteq R$  gilt.

**Anti-Symmetrische Relationen** Eine Relation  $R \subseteq M \times M$  ist *anti-symmetrisch* falls gilt:

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

Äquivalent dazu ist die Definition dass  $R$  genau dann anti-symmetrisch ist, wenn  $R \cap R^{-1} = \text{id}_M$  gilt.

**Transitive Relation** Eine Relation  $R \subseteq M \times M$  ist *transitiv* falls gilt:

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

Äquivalent dazu ist die Definition dass  $R$  genau dann transitiv ist, wenn  $R \circ R \subseteq R$  ist.

**Beispiele:** In den ersten beiden Beispielen sei  $M = \{1, 2, 3\}$ .

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

$R_1$  ist reflexiv, symmetrisch, anti-symmetrisch und transitiv.

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

$R_2$  ist nicht reflexiv, da  $\langle 1, 1 \rangle \notin R_2$ .  $R_2$  ist symmetrisch.  $R_2$  ist nicht anti-symmetrisch, denn aus  $\langle 1, 2 \rangle \in R_2$  und  $\langle 2, 1 \rangle \in R_2$  müßte  $2 = 1$  folgen. Schließlich ist  $R_2$  auch nicht transitiv, denn aus  $\langle 1, 2 \rangle \in R_2$  und  $\langle 2, 1 \rangle \in R_2$  müßte  $\langle 1, 1 \rangle \in R_2$  folgen.

In allen folgenden Beispielen sei  $M = \mathbb{N}$ .

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

$R_3$  ist reflexiv, denn für alle natürlichen Zahlen  $n \in \mathbb{N}$  gilt  $n \leq n$ .  $R_3$  ist nicht symmetrisch, denn beispielsweise gilt  $1 \leq 2$ , aber es gilt nicht  $2 \leq 1$ . Allerdings ist  $R_3$  anti-symmetrisch,

denn wenn  $n \leq m$  und  $m \leq n$  gilt, so muß schon  $m = n$  gelten. Schließlich ist  $R_3$  auch transitiv, denn aus  $k \leq m$  und  $m \leq n$  folgt natürlich  $k \leq n$ .

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

Für zwei positive Zahlen  $m$  und  $n$  gilt  $\langle m, n \rangle \in R_4$  genau dann, wenn  $m$  ein Teiler von  $n$  ist. Damit ist klar, dass  $R_4$  reflexiv ist, denn jede Zahl teilt sich selbst. Natürlich ist  $R_4$  nicht symmetrisch, denn 1 ist ein Teiler von 2 aber nicht umgekehrt. Dafür ist  $R_4$  aber antisymmetrisch, denn wenn sowohl  $m$  ein Teiler von  $n$  ist und auch  $n$  ein Teiler von  $m$ , so muß  $m = n$  gelten. Schließlich ist  $R_4$  auch transitiv: Ist  $m$  ein Teiler von  $n$  und  $n$  ein Teiler von  $o$ , so ist natürlich  $m$  ebenfalls ein Teiler von  $o$ .

Ist  $R$  eine Relation auf  $M$ , die nicht transitiv ist, so können wir  $R$  zu einer transitiven Relation erweitern. Dazu definieren wir für alle  $n \in \mathbb{N}$  die Potenzen  $R^n$  durch Induktion über  $n$ .

1. Induktions-Anfang:  $n = 0$ . Wir setzen

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

2. Induktions-Schritt:  $n \rightarrow n+1$ . Nach Induktions-Voraussetzung ist  $R^n$  bereits definiert. Daher können wir  $R^{n+1}$  definieren als

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

Nun definieren wir den *transitiven Abschluß* von  $R$  als die Menge

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

Dabei ist für eine Folge  $(A_n)_n$  von Mengen der Ausdruck  $\bigcup_{i=1}^{\infty} A_n$  formal wie folgt definiert:

$$\bigcup_{i=1}^{\infty} A_n := \bigcup \{ A_n \mid n \in \mathbb{N} \wedge n \geq 1 \},$$

wobei wir an dieser Stelle daran erinnern, dass nach Definition

$$\bigcup \{ A_n \mid n \in \mathbb{N} \wedge n \geq 1 \} = \{ x \mid \exists n \in \mathbb{N} : x \in A_n \wedge n \geq 1 \}$$

gilt. Anschaulich, aber etwas weniger exakt können wir auch schreiben,

$$\bigcup_{i=1}^{\infty} A_n = A_1 \cup A_2 \cup A_3 \cup \dots$$

Der so definierte transitive Abschluß  $R^+$  einer Relation  $R$  auf  $M$  hat folgende Eigenschaften:

1.  $R^+$  ist transitiv.
2.  $R^+$  ist die bezüglich  $\subseteq$  kleinste Relation  $T$  auf  $M$ , die folgende Eigenschaften hat:

(a)  $T$  ist transitiv.

(b)  $R \subseteq T$ .

Anders ausgedrückt: Ist  $T$  eine transitive Relation auf  $M$  mit  $R \subseteq T$ , so muß  $R^+ \subseteq T$  gelten.

**Äquivalenz-Relation** Eine Relation  $R \subseteq M \times M$  ist eine *Äquivalenz-Relation* auf  $M$  genau dann, wenn  $R$

1. reflexiv,
2. symmetrisch und
3. transitiv ist.

Ein triviales Beispiel für eine Äquivalenz-Relation auf  $M$  ist die Relation  $\text{id}_M$ . Als nicht-triviales Beispiel betrachten wir die Menge  $\mathbb{Z}$  der ganzen Zahlen zusammen mit der Relation  $\approx_n$ , die wir für natürliche Zahlen  $n \neq 0$  wie folgt definieren:

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

Für zwei Zahlen  $x, y \in \mathbb{Z}$  gilt also  $x \approx_n y$  genau dann, wenn  $x$  und  $y$  beim Teilen durch  $n$  den gleichen Rest ergeben. Es läßt sich zeigen, dass die Relation  $\approx_n$  für  $n \neq 0$  eine Äquivalenz-Relation auf  $\mathbb{Z}$  definiert.

Ist  $R$  eine Äquivalenz-Relation auf  $M$  so definieren wir für alle  $x \in M$  Mengen  $[x]_R$  durch

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

Die Mengen  $[x]_R$  werden auch als die *Äquivalenz-Klassen* der Äquivalenz-Relation  $R$  bezeichnet. Die Äquivalenz-Klassen haben folgende Eigenschaften:

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

Die letzten beiden Eigenschaften zeigen, dass zwei Äquivalenz-Klassen entweder gleich, oder aber disjunkt sind.

Ist  $\mathcal{P} \subseteq 2^M$  eine Menge von Teilmengen von  $M$ , so sagen wir, dass  $\mathcal{P}$  eine *Partition* von  $M$  ist, wenn folgende Eigenschaften gelten:

1.  $\forall x \in M: \exists K \in \mathcal{P}: x \in K$ ,  
jedes Element aus  $M$  findet sich also in einem Element aus  $\mathcal{P}$  wieder.
2.  $\forall K \in \mathcal{P}: \forall L \in \mathcal{P}: K \cap L = \emptyset \vee K = L$ ,  
zwei Elemente aus  $\mathcal{P}$  sind also entweder disjunkt oder identisch.

Ist  $\mathcal{P}$  eine Partition von  $M$ , so folgt aus ersten Eigenschaft, dass es für jedes Element  $x$  aus  $M$  eine Menge  $K$  aus  $\mathcal{P}$  gibt, so dass  $x \in K$  liegt. Aus der zweiten Eigenschaft folgt überdies, dass diese Menge eindeutig bestimmt ist, denn nehmen wir einmal an, dass  $x \in K$  und gleichzeitig  $x \in L$  mit  $K \in \mathcal{P}$  und  $L \in \mathcal{P}$  gilt. Wegen  $x \in K$  und  $x \in L$  folgt  $x \in K \cap L$ . Also kann der Schnitt  $K \cap L$  nicht leer sein. Mit der zweiten Eigenschaft folgt daraus, dass schon  $K = L$  gelten muss, es gibt also genau eine Menge  $K \in \mathcal{P}$  mit  $x \in K$ . Wir werden diese Menge aus  $\mathcal{P}$  weiter unten mit  $\text{gen}_{\mathcal{P}}(x)$  bezeichnen.

Nach den oben festgestellten Eigenschaften der Äquivalenz-Klassen ist klar, dass für jede Äquivalenz-Relation  $R$  auf  $M$  die Menge

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

eine Partition bilden. Ist umgekehrt eine Partition  $\mathcal{P}$  auf einer Menge  $M$  gegeben, so läßt sich mittels der Definition

$$R := \{ \langle x, y \rangle \in M \times M \mid \text{gen}_{\mathcal{P}}(x) \cap \text{gen}_{\mathcal{P}}(y) \neq \emptyset \}$$

eine Relation auf  $M$  definieren, von der sich zeigen läßt, dass sie eine Äquivalenz-Relation ist.

**Partielle Ordnung, Totale Ordnung** Eine Relation  $R \subseteq M \times M$  ist eine *partielle Ordnung* (im Sinne von  $\leq$ ) auf  $M$  genau dann, wenn  $R$

1. reflexiv,
2. anti-symmetrisch und
3. transitiv ist.

Die Relation ist darüber hinaus eine *totale Ordnung auf  $M$* , wenn gilt:

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

Auf der Menge der ganzen Zahlen  $\mathbb{Z}$  ist die Relation  $\leq$  eine totale Ordnung. Die Relation  $\text{div}$ , die wir auf der Menge der positiven natürlichen Zahlen durch

$$\text{div} := \{ \langle x, y \rangle \in \mathbb{N} \times \mathbb{N} \mid \exists k \in \mathbb{N}: k * x = y \}$$

definieren, ist eine partielle Ordnung auf  $\mathbb{N}$ . Es gilt  $x \text{ div } y$  genau dann, wenn  $x$  ein Teiler von  $y$  ist. Die Relation  $\text{div}$  ist aber keine totale Ordnung, denn beispielsweise gilt weder  $2 \text{ div } 3$  noch  $3 \text{ div } 2$ .

Partielle Ordnungen sind Verallgemeinerungen der Relation  $\leq$ . Wir geben ein weiteres Beispiel: Dazu betrachten wir die Menge  $2^{\mathbb{N}}$  der Menge aller Teilmengen von  $\mathbb{N}$ . Die Relation  $\subseteq$  ist auf  $2^{\mathbb{N}}$  zwar eine partielle, aber keine totale Ordnung.

Wir schließen damit den theoretischen Teil unseres Ausflugs in die Mengenlehre und verweisen für weitere Details auf die Literatur [4].

## Kapitel 2

# Die Programmier-Sprache SETL2

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 stellen wir daher nun eine Programmier-Sprache vor, die mit diesen Begriffen arbeitet. Dies ist die Sprache SETL2. In dieser Vorlesung können wir aus Zeitgründen nur eine vereinfachte Darstellung der Sprache SETL2 geben. Für einen vollständigen Überblick über die Sprache verweisen wir auf die Literatur. Der Artikel [3] gibt eine gute Einführung, die Arbeiten [7] und [8] beschreiben den vollen Sprachumfang. Sie finden den Artikel [3] auch im Internet unter der Adresse

`ftp://cs.nyu.edu/pub/languages/setl2/doc/2.2/intro.ps.`

### 2.1 Einführende Beispiele

Wir wollen in diesem Abschnitt die Sprache SETL2 an Hand einiger einfacher Beispiele vorstellen, bevor wir dann in den folgenden Abschnitten auf die Details eingehen. Wir beginnen mit dem Programm `hello.stl`, das in Abbildung 2.1 auf Seite 23 abgebildet ist. Die Zeilen-Nummern in dieser und den folgenden Abbildungen von SETL2-Programmen sind nicht Bestandteil der Programme sondern wurden hinzugefügt um besser auf einzelne Zeilen Bezug nehmen zu können.

---

```
1  -- Mein erstes SETL2-Programm
2  program main;
3      print("Hallo, da bin ich.");
4  end main;
```

---

Abbildung 2.1: Ein einfaches SETL2-Programm.

Wenn wir dieses Programm in einer Datei mit dem Namen `hallo.stl` abspeichern, so sind zum Übersetzen und Ausführen dieses Programms die folgenden drei Befehle notwendig:

1. `stll -c setl2.lib`

Dieser Befehl erzeugt eine Bibliothek, die zunächst leer ist und in die wir neue Programme hinzufügen können. Dieser Befehl ist daher nur beim Übersetzen des ersten Programmes erforderlich.

2. `stlc hallo.stl`

Dieses Kommando übersetzt das SETL2-Programm in der Datei "`hallo.stl`" und fügt die erzeugte Übersetzung in die im ersten Schritt erstellte Bibliothek ein.



### 3. stlx main

Hier wird nun das Programm, das in der Bibliothek “setl2.lib” unter dem Namen “main” abgelegt ist, ausgeführt. Als Resultat dieser Ausführung erscheint dann am Bildschirm der folgende Text:

Hallo, da bin ich.

Wir diskutieren nun das Programm in Abbildung 2.1 auf Seite 23 Zeile für Zeile.

1. Die erste Zeile enthält einen Kommentar. In *Setl2* werden Kommentare durch den String “--” eingeleitet. Aller Text zwischen diesem String und dem Ende der Zeile wird von dem *Setl2*-Compiler ignoriert.
2. Die zweite Zeile ist die sogenannte *Programm-Deklaration*, die den Namen des Programms festlegt. Eine Programm-Deklaration beginnt mit dem Schlüsselwort “**program**”, gefolgt von dem Namen des Programms. Im Prinzip ist als Name jede Folge von Buchstaben, Ziffern und dem Zeichen “\_” zugelassen, die mit einem Buchstaben beginnt. Aus Gründen der Bequemlichkeit werden wir immer den Namen “main” verwenden.  
Die Zeile wird mit dem Zeichen “;” abgeschlossen. In *Setl2* werden alle Deklarationen und Kommandos mit einem Semikolon abgeschlossen.
3. Die dritte Zeile enthält das einzige Kommando in diesem Beispiel. Es ist der Aufruf der vordefinierten Prozedur “**print**”. Diese Prozedur kann mit einer beliebigen Anzahl von Argumenten aufgerufen werden, die durch Kommata getrennt werden. Diese Argumente werden dann am Bildschirm ausgegeben. Zusätzlich wird noch eine Zeilen-Umbruch ausgegeben.  
Diese Zeile zeigt auch den ersten Datentyp der Sprache *Setl2*, die *Strings*. Diese werden in *SETL2* in doppelte Anführungszeichen eingeschlossen.
4. Die letzte Zeile beendet das Programm. Wichtig ist, dass hinter “**end**” der selbe Name verwendet wird, der in der Programm-Deklaration zu Beginn des Programms benutzt wurde, in diesem Fall also der Name “main”. Auch diese Zeile muß mit einem Semikolon “;” abgeschlossen werden.

Als nächstes betrachten wir ein Programm, das eine Zahl  $n$  von der Tastatur liest, dann eine Prozedur **sum** mit  $n$  als Argument aufruft, und schließlich das Ergebnis, das von der Prozedur **sum** berechnet worden ist, am Bildschirm ausgibt. Das Programm ist in Abbildung 2.2 auf Seite 25 gezeigt.

Dieses Programm zeigt eine ganze Reihe von Eigenschaften der Sprache SETL2.

1. Zunächst wird in Zeile 2 ein Wert eingelesen. Beachten Sie, dass die Variable **n**, in die der Wert eingelesen worden ist, nicht deklariert worden ist. Im Gegensatz zu Sprachen wie *C* oder *Pascal* ist SETL2 eine ungetypte Sprache, das heißt das eine Variable beliebige Werte annehmen kann.
2. Dann wird in Zeile 3 für den eben eingelesenen Wert von **n** die Prozedur **sum** aufgerufen. Diese Prozedur wird weiter unten in Zeile 7 definiert. Im Gegensatz zu der Sprache *C* ist es nicht notwendig, dass Prozeduren bei ihrem Aufruf bereits definiert sind.  
Zusätzlich enthält Zeile 3 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. Wie auch in *Pascal*, hat in SETL2 der Zuweisungs-Operator den Wert “:=”. Im Gegensatz dazu hat der Zuweisungs-Operator in der Sprache *C* den Wert “=”.
3. Zeile 7 bis Zeile 13 enthalten die Definition der Prozedur **sum**. Die Definition einer Prozedur wird in SETL2 durch das Schlüsselwort “**procedure**” eingeleitet. Hinter diesem Schlüsselwort steht der Name der Prozedur, gefolgt von einer öffnenden Klammer “(”, einer Liste von Argumenten, einer schließenden Klammer “)” und einem Semikolon “;”. Die Definition der

---

```

1  program main;
2      read(n);
3      total := sum(n);
4      print("Summe 0 + 1 + 2 + ... + ", n, " = ", total);
5
6      -- Berechnung der Summe  $\sum_{i=0}^n i$ 
7      procedure sum(n);
8          if n = 0 then
9              return 0;
10         else
11             return n + sum(n-1);
12         end if;
13     end sum;
14 end main;

```

---

Abbildung 2.2: Programm zur Berechnung der Summe  $\sum_{i=0}^n i$

Prozedur wird beendet durch das Schlüsselwort **end** gefolgt von dem Namen der Prozedur und einem Semikolon.

Zwischen diesen Zeilen steht der Rumpf der Prozedur. Im allgemeinen ist dies eine 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:

```

    if test then
        body1
    else
        body2
    end if;

```

Hier sind alle Schlüsselwörter, die wörtlich so im Programm stehen müssen, fett gesetzt. Im einzelnen sind dies die Strings “**if**”, “**then**”, “**else**” und “**end if**”. Die Semantik der Fallunterscheidung ist wie folgt:

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

Die Prozedur **sum** in dem obigen Beispiel ist *rekursiv*, d.h. sie ruft sich selber auf. Damit dies funktioniert muss es einen Fall geben, in dem die Prozedur sich nicht mehr selber aufruft, denn sonst würde die Prozedur in einer Endlos-Schleife stecken bleiben. Dies ist der Fall, wenn die Variable **n** den Wert 0 annimmt. In diesem Fall gibt die Prozedur mit Hilfe des Kommandos “**return 0**” den Wert 0 zurück.

Die generelle Form eines SETL2-Programms ist in Abbildung 2.3 auf Seite 26 angegeben. Hierbei ist

```

    statement 1;
    :
    statement n;

```

die Liste der Statements, die von dem Programm abgearbeitet werden. Diese Liste wird gefolgt von einer Liste von Prozedur-Definitionen. Ähnlich wie in *C* haben diese Prozeduren eine sogenannte *Call-by-Value* Semantik. Damit ist folgendes gemeint: wird einer Prozedur eine Variable  $x$  übergeben und ändert die Prozedur den Wert dieser Variablen ab, so bleibt diese Änderung außerhalb der Prozedur unsichtbar. Würde die Prozedur `sum` beispielsweise den Wert der Variablen `n` abändern, so würde trotzdem in Zeile 4 noch der ursprüngliche Wert von `n` ausgegeben.

```

program prog-name;
  statement1;
  :
  statementn;

  procedure function-name1(args);
    body
  end function-name1;
  :
  procedure function-namem(args);
    body
  end function-namem
end prog-name;

```

Abbildung 2.3: Allgemeine Form eines SETL2 Programms

## 2.2 Darstellung von Mengen

Bisher haben wir noch keine ungewöhnlichen Eigenschaften der Sprache SETL2 sehen können. Alles, was wir bis hierhin präsentiert haben, sieht ganz ähnlich aus wie vergleichbare Konstrukte in den Sprachen *Pascal* oder *C*, nur die Syntax ist etwas anders. Wir präsentieren nun den dritten Datentyp der Sprache SETL2, die Mengen. Wir beginnen mit einem einfachen Programm, das Vereinigung, Schnitt und Differenz zweier Mengen berechnet.

---

```

1  program main;
2    a := { 1, 2, 3 };
3    b := { 2, 3, 4 };
4    -- Berechnung der Vereinigungs-Menge  $a \cup b$ 
5    c := a + b;
6    print(a, " + ", b, " = ", c);
7    -- Berechnung der Schnitt-Menge  $a \cap b$ 
8    c := a * b;
9    print(a, " * ", b, " = ", c);
10   -- Berechnung der Mengen-Differenz  $a \setminus b$ 
11   c := a - b;
12   print(a, " - ", b, " = ", c);
13   -- Berechnung der Potenz-Menge  $2^a$ 
14   c := pow a;
15   print("pow ", a, " = ", c);
16 end main;

```

---

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

In Zeile 2 und 3 sehen wir, dass wir Mengen ganz einfach durch explizite Aufzählung ihrer Argumente angeben können. In den Zeilen 5, 8, 11 und 14 berechnen wir dann nacheinander Vereinigung, Schnitt, Differenz und Potenz-Menge dieser Mengen. Hier ist zu beachten, dass dafür in SETL2 die Operatoren “+”, “\*”, “-” und “pow” verwendet werden. Führen wir dieses Programm aus, so erhalten wir die folgende Ausgabe:

```
{1, 2, 3} + {4, 2, 3} = {4, 1, 2, 3}
{1, 2, 3} * {4, 2, 3} = {2, 3}
{1, 2, 3} - {4, 2, 3} = {1}
pow {1, 2, 3} = {{1, 2, 3}, {}, {2, 3}, {1}, {1, 3}, {2}, {3}, {1, 2}}
```

Hier fällt auf, dass die Reihenfolge der Elemente der Menge **b** bei der Ausgabe anders ist als bei der Definition dieser Menge. Das ist auch in Ordnung, denn bei einer Menge spielt die Reihenfolge der Elemente ja keine Rolle. Daher ist bei der Ausgabe von Mengen die Reihenfolge, mit der die Elemente ausgegeben werden, auch nicht vorhersehbar.

Das obige Beispiel zeigt zwar den Gebrauch von Mengen, ist aber letztlich kaum eindrucksvoller als das Programm `hallo.stl` aus Abbildung 2.1. Um interessantere Programme zeigen zu können, stellen wir jetzt weitere Möglichkeiten vor, um in SETL2 Mengen zu definieren.

**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 natürlichen Zahlen von *start* bis *stop* zugewiesen, formal gilt

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

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:

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

**Definition von Mengen durch Iteratoren** Eine weitere Möglichkeit, Mengen zu definieren, ist mit der 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 Produkte, deren Faktoren  $\leq 10$  sind, in der Schreibweise aus dem letzten Kapitel gilt also

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

Wie ausdrückstark Iteratoren sind, läßt sich an dem folgenden Beispiel erkennen, das in Abbildung 2.5 auf Seite 28 gezeigt ist. Das Programm berechnet die Menge der Primzahlen bis zu einer

vorgegebenen Größe  $n$  und ist so kurz wie eindurcksvoll. 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  $\{x \in \mathbb{N} \mid 2 \leq x \wedge x \leq n\}$  die Menge aller Produkte ab. Genau dies passiert in Zeile 3 des Programms.

---

```

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

---

Abbildung 2.5: Programm zur Berechnung der Primzahlen bis  $n$

Eine allgemeine Form der Definition eine Menge mit Iteratoren ist wie folgt:

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

Hierbei ist  $\text{expr}(x_1, \dots, x_n)$  ein Ausdruck, der die Variablen  $x_1$  bis  $x_n$  enthält und aus diesen Variablen einen neuen Wert berechnet. 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 (man sagt auch *iterieren*). Die mathematische Interpretation der obigen Menge ist dann einfach durch

$$\{ \text{expr}(x_1, \dots, x_n) \mid x_1 \in S_1 \wedge \dots \wedge x_n \in S_n \}$$

gegeben. Die Definition einer Menge über Iteratoren entspricht also der Bildung einer Bild-Menge aus dem letzten Kapitel.

Es ist in SETL2 auch möglich, das Auswahl-Prinzip zu verwenden. Dazu können wir Iteratoren mit einer Bedingung verknüpfen. Die allgemeine Syntax dafür ist folgende:

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

Hierbei haben die Ausdrücke  $\text{expr}(x_1, \dots, x_n)$  und  $S_i$  die selbe Bedeutung wie oben und  $\text{cond}(x_1, \dots, x_n)$  ist ein Ausdruck, der entweder **true** oder **false** ergibt. Die mathematische Interpretation der obigen Menge ist dann

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

Wir geben ein konkretes Beispiel:

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

Nach dieser Zuweisung 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 SETL2 die Funktion **mod** verwenden: Der Ausdruck  $p \bmod x$  berechnet den Rest, der übrig bleibt, wenn Sie die Zahl  $p$  durch  $x$  teilen. (In *C* lautet der entsprechende Operator “%”.) Eine Zahl  $p$  ist also genau dann durch eine andere Zahl  $x$  teilbar, wenn der Rest 0 ist, wenn also  $p \bmod x = 0$  ist. Damit liefert

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

also 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 in Abbildung 2.6 auf Seite 29 benutzt diese Methode zur Berechnung der Primzahlen.

In Zeile 3 dieses Programms haben wir zur Berechnung der Primzahlen eine etwas andere Syntax benutzt, als oben angegeben. Wir haben dabei die folgende Abkürzungs-Möglichkeit der Sprache SETL2 verwendet: Eine Menge der Form

$$\{ x : x \text{ in } \text{expr}(x) \mid \text{cond}(x) \}$$

---

```

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

```

---

Abbildung 2.6: Alternatives Berechnung der Primzahlen

kann kürzer auch in der Form

$$\{x \text{ in } \text{expr}(x) \mid \text{cond}(x)\}$$

geschrieben werden.

## 2.3 Paare und Funktionen

Das Paar  $\langle x, y \rangle$  wird in SETL2 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 SETL2 der Ausdruck  $R(x)$  das eindeutig bestimmte Element  $y$ , für das  $\langle x, y \rangle \in R$  gilt. Das Programm in Abbildung 2.7 auf Seite 29 zeigt dies konkret. Zusätzlich zeigt das Programm noch, dass in SETL2 bei einer binären Relation  $\text{dom}(R)$  als  $\text{domain}(R)$  und  $\text{rng}(R)$  als  $\text{range}(R)$  geschrieben wird.

---

```

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

```

---

Abbildung 2.7: Rechnen mit binären Relationen

Das Programm berechnet zunächst 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. Dann wird diese Relation an der Stelle  $x = 3$  ausgewertet und anschließend werden noch  $\text{dom}(Q)$  und  $\text{rng}(Q)$  berechnet. Das Programm liefert die folgende Ausgabe:

```

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

```

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 in Abbildung 2.8 auf Seite 30 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 SETL2 als “<om>” ausgegeben. Der Versuch, einen undefinierten Wert in eine Menge  $M$  einzufügen, ändert diese Menge nicht, es gibt aber auch keine Fehlermeldung. Deswegen wird in Zeile 5 für die Menge  $\{R(1), R(2)\}$  einfach die leere Menge ausgegeben.

Will man das Auftreten von undefinierten Werten beim Auswerten einer binären Relation vermeiden, so gibt es in SETL2 die Möglichkeit,  $R\{x\}$  statt  $R(x)$  zu schreiben. Dieses ist für eine

---

```

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

```

---

Abbildung 2.8: Rechnen mit nicht-funktionalen binären Relationen

binäre Relation 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} = {4, 1}
R{2} = {}

```

## 2.4 Allgemeine Tupel

Auch beliebige  $n$ -Tupel lassen sich in SETL2 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. 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 29, es hat darüber hinaus aber den Vorteil, dass die Primzahlen sortiert ausgegeben werden.

---

```

program main;
2  n := 100;
3  -- primes is the set of prime numbers that are less or equal than n.
4  primes := [ p in [2..n] | [ x in [1..p] | p mod x = 0 ] = [1, p] ];
5  print( primes );
end main;

```

---

Abbildung 2.9: Berechnung der Primzahlen mit Tupeln

Das Programm in Abbildung 2.10 auf Seite 31 zeigt ein einfaches Verfahren, um eine Menge natürlicher Zahlen in ein sortiertes Tupel zu transformieren. Es benutzt allerdings einige Eigenschaften von SETL2 die wir noch nicht diskutiert haben. Es gibt in SETL2 eine ganze Menge binäre Operatoren, die auf Mengen oder auf Zahlen operieren. Neben den üblichen arithmetischen Operatoren “+”, “-”, “\*”, “/”, “\*\*” die die Summe, die Differenz, das Produkt, den Quotienten und die Potenz berechnen, gibt es in SETL2 noch die beiden Operatoren “min” und “max”, die das Minimum und das Maximum zweier Zahlen berechnen, der Ausdruck “2 min 4” liefert also den Wert 2. Zusätzlich gibt es für jeden dieser Operatoren noch eine Variante, die man durch Anhängen des Zeichens “/” an den Namen des Operators erhält. Der so erzeugte Operator ist dann wieder ein binärer Operator, der allerdings als zweites Argument eine Menge verarbeitet.

Wie das funktioniert, sehen wir am besten an dem folgenden Beispiel. Der Ausdruck

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

wird in SETL2 ausgewertet zu

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

Allgemein wird bei der Notation

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

der binäre Operator “**op**” zwischen  $x$  und alle Elemente aus  $S$  gesetzt. Falls  $S$  leer ist, wird einfach  $x$  zurück gegeben. Für  $x$  wird dabei dann sinnvollerweise das neutrale Element der zugrunde liegenden Operation “**op**” verwendet. Für  $S$  kann hier entweder ein Tupel oder aber eine Menge verwendet werden. Ist der Operator  $op$  nicht kommutativ oder nicht assoziativ, so macht allerdings die Verwendung einer Menge kaum Sinn, denn das Ergebnis hängt in so einem Fall von der Reihenfolge ab, in der die Elemente der Menge intern von SETL2 gespeichert werden.

---

```

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

---

Abbildung 2.10: Sortieren einer Menge

Ein zusammengesetzter Operator kann auch als unärer Operator verwendet werden. Beispielsweise können wir durch

$$\text{sum} := +/ S$$

alle Elemente der Menge  $S$  aufsummieren. Dies geht allerdings nur, wenn die Menge  $S$  nicht leer ist, sonst bricht das Programm zur Laufzeit mit einer Fehlermeldung ab. Da die Summe der Elemente einer leeren Menge intuitiv ganz einfach 0 ist, ist in der Regel die Verwendung von “+” als binärer Operator vorzuziehen. Das obige Beispiel würde sich dann als

$$\text{sum} := 0 +/ S$$

schreiben. Es gibt allerdings auch Operatoren, wo dies so nicht möglich ist, weil der zugrunde liegende Operator auf den natürlichen Zahlen kein neutrales Element besitzt. Beispielsweise kann das Minimum einer Menge nur durch

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

berechnet werden, denn das neutrale Element des Operators “**min**” ist  $\infty$  und dieses Objekt können wir in SETL2 nicht darstellen.

**Spezielle Funktionen für Tupel und Mengen** Es gibt eine Reihe vordefinierter Funktionen für Mengen und Tupel, von denen wir jetzt einige behandeln werden. Da ist zunächst einmal der Operator “**from**” mit der wir ein (nicht näher spezifiziertes) Element aus einer Menge auswählen können. Die Syntax ist:

$$x \text{ from } S;$$

Hierbei  $S$  eine Menge und  $x$  der Name einer Variablen. 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 in Abbildung 2.11 auf Seite 32 nutzt diese Anweisung um eine Menge elementweise auszugeben. Jedes Element wird dabei in einer eigenen Zeile ausgedruckt.



---

```

1  program main;
2      S := { 13, 5, 7, 2, 4 };
3      printSet(S);
4
5      -- print the set S one element per line
6      procedure printSet(S);
7          if S = {} then
8              return;
9          end if;
10         x from S;
11         print(x);
12         printSet(S);
13     end printSet;
14 end main;

```

---

Abbildung 2.11: Menge elementweise ausdrucken

Neben dem binären Operator “**from**” gibt es noch den unären Operator “**arb**”, der ein beliebiges Element aus einer Menge auswählt, die Menge selbst aber unverändert läßt. Nach den Statements

```

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

```

würde also für  $x$  entweder der Wert “1” oder “2” ausgedruckt, während für  $S$  auf jeden Fall der Wert “{1,2}” ausgedruckt wird.

Als Analogon zu dem Operator “**from**” sind für Tupel die Operatoren “**fromb**” und “**frome**” definiert, die das erste bzw. letzte Element eines Tupels entfernen. Weiterhin steht für Tupel der Operator “**+**” zur Verfügung, mit dem zwei Tupel aneinander gehängt werden können. Außerdem gibt es noch den unären Operator “**#**”, der für Mengen und Tupel die Anzahl der Elemente berechnet. Schließlich kann man Elemente von Tupeln mit der Schreibweise

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

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

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

wird das Tupel  $t$  so abgeändert, dass das  $n$ -te Element danach den Wert  $x$  hat.

Das Programm in Abbildung 2.12 auf Seite 33 demonstriert die eben vorgestellten Operatoren. Es produziert die 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
x = 1
a = [2, 3]
x = 6
b = [2, 3, 4, 5]
[2, 3, 4, 5](3) = 4
b = [2, 3, 42, 5]

```

---

```

1  program main;
2      a := [ 1, 2, 3 ];
3      b := [ 2, 3, 4, 5, 6 ];
4      c := { 5, 6, 7 };
5      -- Aneinanderhängen der Tupel mit +
6      print(a, " + ", b, " = ", a + b);
7      -- Berechnung der Anzahl der Elemente einer Menge
8      print("# ", c, " = ", # c);
9      -- Berechnung der Länge eines Tupels
10     print("# ", a, " = ", # a);
11     -- Entfernen des ersten Elements des Tupels a
12     x fromb a;
13     print("x = ", x);
14     print("a = ", a);
15     -- Entfernen des letzten Elements des Tupels b
16     x frome b;
17     print("x = ", x);
18     print("b = ", b);
19     -- Auswahl des dritten Elements von b
20     print(b, "(3) = ", b(3) );
21     -- Ändern des dritten Elements von b
22     b(3) := 42;
23     print("b = ", b);
24 end main;

```

---

Abbildung 2.12: Weitere Operatoren auf Tupeln und Mengen

## 2.5 Kontroll-Strukturen

Die Sprache SETL2 stellt alle Kontroll-Strukturen zur Verfügung, die heutzutage in imperativen Sprachen üblich sind. Wir haben “if”-Abfragen bereits mehrfach gesehen. In der allgemeinsten Form hat eine Fallunterscheidung die in Abbildung 2.13 auf Seite 33 gezeigte Struktur.

```

if test0 then
    body0
elseif test1 then
    body1
:
elseif testn then
    bodyn
else
    bodyn+1
end if;

```

Abbildung 2.13: Struktur der Fallunterscheidung

Hierbei steht *test<sub>i</sub>* für einen Test, der “true” oder “false” liefert. Liefert der Test “true”, so wird der zugehörigen Anweisungen in *body<sub>i</sub>* ausgeführt, andernfalls wird der nächste Test *test<sub>i+1</sub>* versucht.

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:

$x \text{ in } S$

ist genau dann **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 “**and**”, “**or**” und “**not**” komplexere Tests aufgebaut werden.

### 2.5.1 Schleifen

Es gibt in SETL2 **while**-Schleifen, **for**-Schleifen, **until**-Schleifen, sowie Schleifen ohne Abbruch.

**while-Schleifen** Die allgemeine Syntax der **while**-Schleife ist in Abbildung 2.14 auf Seite 34 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 loop
    body
end loop;
```

Abbildung 2.14: Struktur der **while**-Schleife

Abbildung 2.15 auf Seite 34 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  program main;
2      n := 100;
3      primes := {};
4      p := 2;
5      while p <= n loop
6          if { t in primes | p mod t = 0 } = {} then
7              print(p);
8              primes := primes + { p };
9          end if;
10         p := p + 1;
11     end loop;
12 end main;
```

---

Abbildung 2.15: Prozedurale Berechnung der Primzahlen

**for-Schleifen** Die allgemeine Syntax der **for**-Schleife ist in Abbildung 2.16 auf Seite 35 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 jeweils mit dem jeweiligen Wert von *x* der Schleifenrumpf *body* ausgeführt. Anstelle einer Menge kann *S* auch ein Tupel sein.

```

for x in S loop
    body
end loop;

```

Abbildung 2.16: Struktur der **for**-Schleife

Abbildung 2.17 auf Seite 35 zeigt eine Berechnung von Primzahlen mit Hilfe einer **for**-Schleife. Der dabei verwendete Algorithmus ist als das *Sieb des Erasthenes* bekannt. Das funktioniert wie folgt: Sollen alle Primzahlen kleiner oder gleich  $n$  berechnet werden, so wird zunächst ein Tupel der Länge  $n$  gebildet, dessen sämtliche Elemente den Wert 1 haben. Das passiert in der Schleifen in den Zeilen 4 bis 6. Anschließend werden in der **for**-Schleife in den Zeilen 7 bis 13 alle Elemente des Tupels **primes** auf den Wert 0 gesetzt, die sich als Produkt zweier Zahlen  $i$  und  $j$  darstellen lassen. 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.

---

```

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

```

---

Abbildung 2.17: Berechnung der Primzahlen nach Erasthenes

Der Algorithmus aus Abbildung 2.17 kann durch die folgende Beobachtung noch verbessert werden: Falls in der Schleife von Zeile 7 bis 13 die Zahl  $i$  schon als zusammengesetzte Zahl erkannt ist, so bringt es nichts mehr, die **while**-Schleife in den Zeilen 9 bis 12 zu durchlaufen, denn alle Indizes, für die dort **primes**( $i * j$ ) 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 2.18 auf Seite 36 zeigt den resultierenden Algorithmus. Um den Durchlauf der inneren **while** Schleife in dem Fall, dass **primes**( $i$ ) = 0 ist, zu überspringen, haben wir das **continue** Statement 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 8 fort. Das **continue** verhält sich also genauso, wie der Befehl “**continue**” in der Sprache C.

**loop-Schleifen** Die allgemeine Syntax der **loop**-Schleife ist in Abbildung 2.19 auf Seite 36 gezeigt. Eine solche Schleife wird so lange durchlaufen, bis sie mit einem **exit**- oder **return**-Kommando verlassen wird. Solche Schleifen sind dann nützlich, wenn die zum Abbruch zu testende

---

```

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

```

---

Abbildung 2.18: Effizientere Berechnung der Primzahlen nach Eratosthenes

Bedingung am Anfang der Schleife noch nicht berechnet werden kann.

```

loop
    body
end loop;

```

Abbildung 2.19: Struktur der `loop`-Schleife

Wir geben ein Beispiel das aufzeigt, in welchen Fällen so etwas nützlich ist. 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 2.20 auf Seite 37 angegebene Algorithmus zum Ziel.

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 `exit`-Kommando. Dieses bricht die Schleife ab. In der Sprache *C* gibt es ein Kommando mit der selben Semantik. Dieses Kommando heißt dort `break`.

Ganz nebenbei zeigt das obige Beispiel auch, dass Sie in SETL2 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 SETL2 stehen unter anderem die folgenden reellen Funktionen zur Verfügung:

1. Die trigonometrischen Funktionen `sin()`, `cos()` und `tan()`.

---

```

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

```

---

Abbildung 2.20: Lösung der Gleichung  $x = \cos(x)$  durch Iteration.

- Die Umkehr-Funktionen der trigonometrischen Funktionen sind `asin()`, `acos()` und `atan()`. Zusätzlich berechnet `atan2(y, x)` den Arcus-Tangens von  $\frac{y}{x}$ .
- Die Exponential-Funktion `exp()` und der natürliche Logarithmus `log()`.
- Die Funktion `abs()` berechnet den Absolut-Betrag des übergebenen Arguments.
- Die Funktion `fix()` schneidet die Nachkomma-Stellen einer reellen Zahl ab.

## 2.6 Fallstudie: Berechnung des kürzesten Wegs

Wir wollen dieses Kapitel mit einer praktisch relevanten Anwendung der Sprache SETL2 abschließen. Dazu betrachten wir das Problem, kürzeste Wege in einem *Graphen* zu berechnen, wobei wir unter einem Graphen zunächst einfach eine Ansammlung von Informationen über bestimmte Wegstrecken verstehen wollen. Der Graph enthält also die Information, zwischen welchen Punkten es überhaupt Verbindungen gibt und welche Länge diese Verbindungen haben. Zur Vereinfachung wollen wir annehmen, dass die einzelnen Punkte durch Zahlen identifiziert werden. Dann können wir die Wegstrecken als Paare von Zahlen darstellen. Den Graphen selber stellen wir als eine Menge von Paaren dar: Die erste Komponente dieser Paare ist dann eine Wegstrecke, also selber wieder ein Paar, und die zweite Komponente spezifiziert die Länge der Wegstrecke. Wir betrachten ein Beispiel. Sei  $G$  durch

$$G := \{ [ [1, 2], 2 ], [ [2, 3], 1 ], [ [1, 3], 4 ], [ [2, 4], 7 ], [ [4, 5], 1 ] \};$$

definiert. In diesem Graphen haben wir die Punkte 1, 2, 3, 4 und 5. Zwischen den Punkten 1 und 2 gibt es eine Verbindung der Länge 2, zwischen den Punkten 2 und 3 gibt es eine Verbindung der Länge 1 und so weiter. 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. 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 der Länge 4. Es gibt aber auch noch einen Weg von 1 nach 3, der über den Punkt 2 geht. Die Gesamt-Länge dieses Weges ist 3, dieser Weg ist also kürzer. Wenn Sie sich diese Situation anschaulich vorstellen wollen, dann nehmen Sie einfach an, dass auf dem direkten Weg zwischen 1 und 3 ein hoher Berg liegt und dass der Weg, der über diesen Berg führt, aus sehr verschlungenen Serpentinaen besteht.

Nachdem wir jetzt das Problem in der Sprache der Mengenlehre mathematisch formuliert haben, versuchen wir es zu lösen. Da das Problem zu komplex ist, um in einem Schlag gelöst werden zu können, versuchen wir, das Problem zu vereinfachen und lösen zunächst nur dieses einfachere Problem. Unsere Hoffnung besteht dann darin, dass es uns gelingt, die Lösung des vereinfachten Problems zu einer Lösung des ursprünglichen Problems auszubauen.

### 2.6.1 Berechnung des transitiven Abschlusses einer Relation

Wie sollen wir nun das Problem vereinfachen? Ein Problem kann dadurch vereinfacht werden dass von bestimmten Informationen abstrahiert wird. In unserem Fall können wir von der Entfernung abstrahieren und zunächst einmal nur untersuchen, zwischen welchen Punkten es überhaupt eine Verbindung gibt. Wir transformieren also den Graphen  $G$  in die binäre Relation  $R$ :

$$R := \{ [1,2], [2,3], [1,3], [2,4], [4,5] \};$$

Um feststellen zu können, ob es zwischen zwei Punkten einen Weg gibt, müssen wir den transitiven Abschluß  $R^+$  der Relation  $R$  bilden. Wir erinnern daran, dass wir im ersten Kapitel festgestellt 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 Wege, 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 Wege enthalten, die aus zwei einfachen Wegen zusammengesetzt sind. Allgemein läßt sich durch Induktion sehen, dass  $R^n$  alle die Wege enthält, die aus  $n$  einfachen Wegen 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  $k - 1$  einfachen Wegen 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. Wir berechnen einfach die Menge

$$\bigcup_{i=1}^{\infty} R^i$$

indem wir so lange  $R^i$  zu der schon berechneten Vereinigung hinzu addieren, wie sich dabei neue Wege ergeben. Das Programm in Abbildung 2.21 auf Seite 39 zeigt eine mögliche Implementierung des zugrunde liegenden Gedankens.

Lassen wir dieses Programm laufen, so erhalten wir als Ausgabe:

$$\begin{aligned} R &= \{ [2, 3], [4, 5], [1, 3], [2, 4], [1, 2] \} \\ R^+ &= \{ [1, 5], [2, 3], [4, 5], [1, 4], [1, 3], [2, 4], [1, 2], [2, 5] \} \end{aligned}$$

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 endliche Folge der Form

$$\langle \langle x_0, y_0 \rangle, \langle x_1, y_1 \rangle, \dots, \langle x_n, y_n \rangle \rangle,$$

für die folgendes gilt:

1.  $x = x_0$ .

Der Pfad beginnt also an dem Punkt  $x$ .

2.  $y_{i-1} = x_i$  für alle  $i = 1, \dots, n$ .

Der  $i$ -te Wegstück fängt da an, wo das  $(i-1)$ -te Wegstück aufgehört hat.

3.  $y_n = y$ .

Der Pfad endet in dem Punkt  $y$ .

---

```

1  program main;
2
3      R := { [1,2], [2,3], [1,3], [2,4], [4,5] };
4      print( "R = ", R );
5      print( "computing transitive closure of R" );
6      T := closure(R);
7      print( "R+ = ", T );
8
9      -- The procedure call closure(R) computes the transitive closure
10     -- of the binary relation R.
11     procedure closure(R);
12         T := R;
13         loop
14             Old_T := T;
15             T := R + product(R, T);
16             if T = Old_T then
17                 return T;
18             end if;
19         end loop;
20     end closure;
21
22     -- The procedure call product(R1, R2) computes the relational
23     -- product R1 o R2.
24     procedure product(R1, R2);
25         return { [x,z] : [x,y1] in R1, [y2,z] in R2 | y1 = y2 };
26     end product;
27
28 end main;

```

---

Abbildung 2.21: Berechnung des transitiven Abschlusses

## 2.6.2 Berechnung des kürzesten Weges

Wir versuchen nun, die Berechnung des transitiven Abschlusses zu einer Berechnung des kürzesten Weges zu verallgemeinern. Zunächst betrachten wir noch mal unsere Eingabe-Daten  $G$ :

$$G := \{ [[1,2], 2], [[2,3], 1], [[1,3], 4], [[2,4], 7], [[4,5], 1] \};$$

Wir stellen fest, dass  $G$  nicht anderes ist als eine Funktion, deren Definitions-Bereich durch die Relation  $R$  mit

$$R := \{ [1,2], [2,3], [1,3], [2,4], [4,5] \};$$

gegeben ist. Der zentrale Punkt bei der Berechnung des transitiven Abschlusses war die Berechnung der Komposition zweier binärer Relationen. Diese hatten wir durch folgenden Ausdruck berechnet:

$$\{ [x,z] : [x,y1] \text{ in } R1, [y2,z] \text{ in } R2 \mid y1 = y2 \};$$

Diesen Ausdruck müssen wir nun in zwei Aspekten erweitern:

1. Wir gehen nicht mehr von binären Relationen  $R1$  und  $R2$  aus, sondern verwenden statt dessen Abstands-Funktionen  $D1$  und  $D2$ , deren Definitions-Bereiche  $R1$  und  $R2$  binäre Relationen sind.
2. Für zwei Punkte  $x$  und  $z$  geht es nun nicht mehr nur um die Frage, ob es ein  $y$  mit  $\langle x, y \rangle \in R1$  und  $\langle y, z \rangle \in R2$  gibt, sondern wir müssen zusätzlich noch die Abstände der beteiligten Paare  $\langle x, y \rangle$  und  $\langle y, z \rangle$  aufsummieren.



Diese führt zu den folgenden Definitionen.

**Definition 1 (Abstands-Funktion)** Eine *Abstands-Funktion*  $D$  ist eine Funktion

$$D : R \rightarrow \mathbb{N}$$

die auf einer endlichen Menge  $R$  von Paaren von Punkten definiert ist.

Die Natur der Punkte, die die Paare bilden, auf denen eine Abstands-Funktion definiert ist, ist für unsere Belange unwichtig. In unseren Programmen werden wir natürliche Zahlen als Punkte wählen.

**Definition 2 (Komposition von Abstands-Funktionen)** Sind zwei Abstands-Funktionen  $D_1$  und  $D_2$  auf den Paar-Mengen  $R_1$  und  $R_2$  definiert, so definieren wir die *Komposition* von  $D_1 \circ D_2$  dieser Funktionen wie folgt:

$$D_1 \circ D_2 := \left\{ \langle \langle x, z \rangle, D_1(x, y) + D_2(y, z) \rangle \mid \exists y : \langle x, y \rangle \in R_1 \wedge \langle y, z \rangle \in R_2 \right\}.$$

Es läßt sich leicht zeigen, dass  $\text{dom}(D_1 \circ D_2) = \text{dom}(D_1) \circ \text{dom}(D_2)$  ist. Dieser Umstand rechtfertigt die Schreibweise  $D_1 \circ D_2$ .

Beachten Sie, dass  $D_1 \circ D_2$  im allgemeinen keine Funktion ist, denn es kann vorkommen, dass zu einem Paar  $\langle x, z \rangle$  verschiedene Punkte  $y_1, y_2$  mit  $\langle x, y_i \rangle \in R_1$  und  $\langle y_i, z \rangle \in R_2$  existieren. Falls dann  $D_1(x, y_1) + D_2(y_1, z) \neq D_1(x, y_2) + D_2(y_2, z)$  gilt, ist die Relation  $D_1 \circ D_2$  nicht mehr rechts-eindeutig und also keine Funktion. Eine solche Relation wollen wir als *Abstands-Relation* bezeichnen.

Es ist relativ leicht, eine solche Abstands-Relation zu einer Funktion zu machen: Da es uns um die Berechnung kürzester Pfade geht, können wir das Minimum aller dieser Werte nehmen. Wir definieren eine entsprechende Funktion *make\_func* die auf diese Weise eine Abstands-Relation  $D$  in eine Abstands-Funktion umwandelt:

$$\text{make\_func}(D) := \left\{ \langle x, \min(\{y \mid \langle x, y \rangle \in D\}) \rangle \mid x \in \text{dom}(D) \right\}.$$

Wir erinnern daran, dass wir eine Menge der Form

$$\{y \mid \langle x, y \rangle \in D\}$$

in SETL2 sehr einfach berechnen können, denn diese Menge wird in SETL2 durch den Ausdruck  $D\{x\}$  berechnet. Zur Berechnung des Minimums können wir dann den Operator “`min/`” verwenden, denn wir wissen ja, dass die Menge  $D\{x\}$  nicht leer ist. Das ganze führt dann zu dem in Abbildung 2.22 auf Seite 41 abgebildeten Programm.

Wir müssen die Funktion *make\_func* zweimal aufrufen: Einmal transformieren wir das Ergebnis von *compose* in Zeile 16. Aber das reicht noch nicht, denn wenn wir in Zeile 17 die Vereinigung von  $T$  mit  $D_n$  bilden, ist die resultierende Relation unter Umständen nicht mehr rechts-eindeutig. Daher müssen wir die Funktion *make\_func* auch in Zeile 17 aufrufen.

Vergleichen wir die Programme in Abbildung 2.22 und Abbildung 2.21, so erkennen wir, dass der Fixpunkt-Algorithmus in beiden Fällen der selbe ist. Es gibt nur zwei Unterschiede:

1. Wir berechnen jetzt nicht mehr die Komposition von binären Relationen, sondern die Komposition von Abstands-Funktionen.
2. Wir müssen die Vereinigung noch in eine Funktion transformieren.

Lassen wir das Programm laufen, so erhalten wir die folgende Ausgabe:

```
D = {[[2, 3], 1], [[4, 5], 1], [[1, 2], 2], [[2, 4], 7], [[1, 3], 4]}
D+ = {[[2, 3], 1], [[4, 5], 1], [[1, 4], 9], [[1, 3], 3], [[2, 4], 7],
      [[1, 2], 2], [[1, 5], 10], [[2, 5], 8]}
```

Die läßt in so fern noch zu wünschen übrig, dass wir jetzt zwar wissen, welche Länge der kürzeste Weg zwischen zwei Punkten hat. Wir wissen aber nicht, wo dieser Weg her geht. Eine entsprechende Erweiterung des Programms empfehle ich Ihnen zur Übung.

---

```

1  program main;
2      -- G gives the distance between two points
3      G := { [[1,2],2], [[2,3],1], [[1,3],4], [[2,4],7], [[4,5],1] };
4
5      print( "G = ", G );
6      T := closure(G);
7      print( "G+ = ", T );
8
9      -- The procedure call closure(D) computes the transitive closure
10     -- of the distance function D.
11     procedure closure(D);
12         T := D;
13         D_n := D;
14         loop
15             Old_T := T;
16             D_n := make_func( composition(D, D_n));
17             T := make_func(T + D_n);
18             if T = Old_T then
19                 return T;
20             end if;
21         end loop;
22     end closure;
23
24     -- The procedure call composition(D1, D2) computes the composition of
25     -- the distance functions D1 and D2.
26     procedure composition(D1, D2);
27         R1 := domain(D1);
28         R2 := domain(D2);
29         return { [ [x,z], D1([x,y1]) + D2([y2,z]) ]
30                 : [x,y1] in R1, [y2,z] in R2 | y1 = y2 };
31     end composition;
32
33     -- Turn the relation D into a function by computing the minimum.
34     procedure make_func(D);
35         return { [ x, min/ D{x} ] : x in domain(D) };
36     end make_func;
37 end main;

```

---

Abbildung 2.22: Berechnung des kürzesten Weges

### 2.6.3 Ausblick

Wir können aus Zeitgründen nur einen Teil der Funktionalität von SETL2 diskutieren. Insbesondere haben wir die folgenden Aspekte nicht behandelt:

1. Funktionen zur String-Verarbeitung.
2. Funktionen zum Lesen und Schreiben von Dateien.
3. Funktionen zur Kommunikation mit dem Betriebs-System.
4. SETL2 ist objekt-orientiert. Ähnlich wie in  $C^{++}$  oder *Java* können Sie mit Klassen und Objekten arbeiten.

5. Über sogenannte *packages* können Sie komplexe Software-Pakete erstellen, die nur über eine wohldefinierte Schnittstelle angesprochen werden können. Das Konzept ist ähnlich wie bei *Java*.
6. Darüber hinaus gibt es eine Schnittstelle zu der Sprache *C*. Sie können also Funktionen in *C* implementieren und diese dann in SETL2 aufrufen.

Für eine detailliertere Darstellung dieser Aspekte verweise ich auf die Artikel [7] und [8]. Außerdem gibt es im Internet unter

<http://www.settheory.com> eine Vorabversion eines Buches über SETL2.

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.

# 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”. Aussagen sind dabei einfache Sätze, die einen Tatbestand ausdrücken. Beispiel für 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. Hätten wir zusätzlich die Aussagen

“Die Sonne scheint.”    und  
“Es regnet.”

gegeben, so könnten wir daraus die Aussage

“Am Himmel ist ein Regenbogen.”

folgern. Die Aussagenlogik beschäftigt sich nun mit der Frage, wann solche Schlußfolgerungen korrekt sind. Dazu abstrahiert die Aussagenlogik von dem Wahrheitswert der einzelnen Aussagen und untersucht zunächst die Frage, 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 folgern können.

Um die Struktur komplexerer Aussagen übersichtlich werden zu lassen, führen wir in der Aussagenlogik zunächst sogenannte *Aussage-Variablen* ein. Wir zeigen das an dem obigen Beispiel. Wir würden dort beispielsweise die folgenden *Aussage-Variablen* einführen:

1. SonneScheint
2. EsRegnet
3. Regenbogen

Zusätzlich führen wir für die Junktoren “nicht”, “und”, “oder”, “nicht”, “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*

Diese Abkürzungen ermöglichen uns eine übersichtlichere Notation. Die Aussage (1) können wir jetzt kürzer als

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

schreiben. Das *Beweis-Prinzip*, das wir oben verwendet haben, ist dabei wie folgt: Aus den Aussagen

1. **SonneScheint**
2. **EsRegnet**
3. **SonneScheint  $\wedge$  EsRegnet  $\rightarrow$  Regenbogen**

folgt *logisch* die Aussage

**Regenbogen.**

Um Beweis-Prinzipien übersichtlicher angeben zu können, führen wir die folgende Notation ein:

$$\frac{\text{SonneScheint} \quad \text{EsRegnet} \quad \text{SonneScheint} \wedge \text{EsRegnet} \rightarrow \text{Regenbogen}}{\text{Regenbogen}}$$

Die Aussagen über dem Bruchstrich bezeichnen wir als *Prämissen*, die Aussage unter dem Bruchstrich ist die *Konklusion*. Statt Beweis-Prinzip sagen wir oft auch *Schluß-Regel*.

Wir stellen fest, dass die obige Schluß-Regel unabhängig von dem Wahrheitswert der Aussagen in dem folgenden Sinne gültig ist: Wenn alle Prämissen gültig sind, dann folgt aus logischen Gründen auch die Gültigkeit der Konklusion. Um dieses weiter formalisieren zu können, ersetzen wir die Aussage-Variablen **SonneScheint**, **EsRegnet** und **Regenbogen** durch die *Meta-Variablen*  $p$ ,  $q$  und  $r$ , die für beliebige aussagenlogische Formeln stehen. Die obige Schluß-Regel ist dann eine Instanz der folgenden allgemeinen Schluß-Regel:

$$\frac{p \quad q \quad p \wedge q \rightarrow r}{r}$$

**Aufgabe:** Formalisieren Sie die Schluß-Regel, die in dem folgenden Argument verwendet wird.

*Wenn es regnet, ist die Straße naß. Es regnet nicht.  
Also ist die Straße nicht naß.*

**Lösung:** Es wird die folgende Schluß-Regel verwendet:

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

Diese Schluß-Regel ist nicht korrekt. Wenn Sie das nicht einsehen, sollten Sie bei strahlendem Sonnenschein einen Eimer Wasser auf die Straße kippen.  $\square$

Dadurch, dass wir ausgehend von Beobachtungen und als wahr erkannten Tatsachen und Zusammenhängen mehrere *logische Schlüsse* aneinander fügen, erhalten wir einen *Beweis*. Die als wahr erkannten Tatsachen und Beobachtungen bezeichnet wir dabei als *Axiome*. Als Notation für Beweise vereinbaren wir die folgende Schreibweise:

$$M \vdash r.$$

Hierbei ist  $M$  eine Menge von Aussagen und  $r$  ist eine einzige Aussage. Die obige Schreibweise

wäre dann als

“Aus den Aussagen der Menge  $M$  kann die Aussage  $r$  hergeleitet werden”

zu lesen. Damit ist gemeint, dass wir ausgehend von den Axiomen in  $M$  durch sukzessives Anwenden verschiedener Schluß-Regeln die Aussage  $r$  beweisen können. Das Zeichen  $\vdash$  symbolisiert dabei den Herleitungs-Begriff, den wir gelegentlich auch als *Beweis-Begriff* bezeichnen. Wir werden in einem späteren Abschnitt den Herleitungs-Begriff formal definieren. Parallel zu dem Herleitungs-begriff, der seiner Natur nach syntaktisch ist, gibt es auch einen *semantischen*, also inhaltlichen *Folgerungs-Begriff*. Wir schreiben

$$M \models r,$$

wenn die Aussage  $r$  logisch aus den Aussagen  $M$  folgt. Das können wir anders auch so formulieren: Immer wenn alle Aussagen aus  $M$  wahr sind, dann ist auch die Aussage  $r$  wahr. Wir können den Begriff der *logischen Folgerung* aber erst dann präzise definieren, wenn wir die Semantik der Junktoren mathematisch festgelegt haben.

Ziel der Aussagenlogik ist es, einen Herleitungsbegriff zu finden, der die folgenden beiden Bedingungen erfüllt:

1. Der Herleitungsbegriff sollte **korrekt** sein, es sollte also nicht möglich sein, Unsinn zu beweisen. Es sollte also gelten

$$\text{Aus } M \vdash r \text{ folgt } M \models r.$$

2. Der Herleitungsbegriff sollte **vollständig** sein, d.h. wenn eine Aussage  $r$  aus einer Menge von anderen Aussagen  $M$  logisch folgt, dann sollte sie auch aus  $M$  beweisbar sein:

$$\text{Aus } M \models r \text{ folgt } M \vdash r.$$

## 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 Gattern. Ein Gatter ist dabei, aus logischer Sicht betrachtet, ein atomarer Baustein. Die Komplexität solcher Schaltungen wäre ohne den Einsatz rechnergestützter Verfahren zur Verifikation nicht mehr beherrschbar. Die dabei eingesetzten Verfahren sind Anwendungen der Aussagenlogik.

Software-Werkzeuge, die für die Verifikation digitaler Schaltungen eingesetzt werden, kosten heutzutage etwa 100.000 \$. Diese zeigt die wirtschaftliche Bedeutung der Aussagenlogik.

2. Erstellung von Stundenplänen.

Allgemein lassen sich viele diskrete Optimierungs-Probleme durch aussagenlogische Formeln beschreiben und dann mit Algorithmen der Aussagenlogik lösen.

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. Solche Fahrstraßen dürfen sich nicht kreuzen. Das Erstellen von Fahrstraßen kann durch aussagenlogische Formeln modelliert werden.

4. Eine Reihe kombinatorischer Puzzles lassen sich als aussagenlogische Formeln interpretieren und dann mit Hilfe aussagenlogischer Methoden lösen. Als ein Beispiel möchte ich hier das 8-Damen-Problem nennen. 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 betrachten eine Menge  $\mathcal{P}$  von *Aussage-Variablen* als gegeben. Die Menge der aussagenlogischen Formeln  $\mathcal{F}$  definieren wir ausgehend von  $\mathcal{P}$  wie folgt:

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  $p \in \mathcal{F}$ .
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  $(f_1 \vee f_2) \in \mathcal{F}$ .
5. Sind  $f_1, f_2 \in \mathcal{F}$ , so gilt auch  $(f_1 \wedge f_2) \in \mathcal{F}$ .
6. Sind  $f_1, f_2 \in \mathcal{F}$ , so gilt auch  $(f_1 \rightarrow f_2) \in \mathcal{F}$ .
7. Sind  $f_1, f_2 \in \mathcal{F}$ , so gilt auch  $(f_1 \leftrightarrow f_2) \in \mathcal{F}$ .

Um Klammern zu sparen vereinbaren wir folgendes:

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

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

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

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

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

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

Um den aussagenlogischen Formeln eine Semantik zu geben, 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 3 (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.  $\oslash : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
3.  $\odot : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
4.  $\ominus : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
5.  $\ominus : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

Die Werte dieser Funktionen werden durch die Tabelle 3.1 auf Seite 47 definiert: Nun können

$p$	$q$	$\neg(p)$	$\odot(p, q)$	$\oslash(p, q)$	$\ominus(p, q)$	$\ominus(p, q)$
true	true	false	true	true	true	true
true	false	false	true	false	false	false
false	true	true	true	false	true	false
false	false	true	false	false	true	true

Tabelle 3.1: Interpretation der Junktoren.

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  $\hat{\mathcal{I}}(f)$  bezeichnen.

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

Um die Schreibweise nicht übermäßig kompliziert werden zu lassen, unterscheiden wir in Zukunft nicht mehr zwischen  $\hat{\mathcal{I}}$  und  $\mathcal{I}$ , wir werden das Hüttchen 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:



$$\begin{aligned}
\mathcal{I}\left((p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q\right) &= \ominus\left(\mathcal{I}((p \rightarrow q)), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\right) \\
&= \ominus\left(\ominus(\mathcal{I}(p), \mathcal{I}(q)), \ominus(\mathcal{I}(\neg p \rightarrow q), \mathcal{I}(q))\right) \\
&= \ominus\left(\ominus(\mathbf{true}, \mathbf{false}), \ominus(\mathcal{I}(\neg p \rightarrow q), \mathbf{false})\right) \\
&= \ominus\left(\mathbf{false}, \ominus(\mathcal{I}(\neg p \rightarrow q), \mathbf{false})\right) \\
&= \ominus\left(\mathbf{false}, \ominus(\ominus(\mathcal{I}(\neg p), \mathcal{I}(q)), \mathbf{false})\right) \\
&= \ominus\left(\mathbf{false}, \ominus(\ominus(\ominus(\mathcal{I}(p))), \mathbf{false}), \mathbf{false})\right) \\
&= \ominus\left(\mathbf{false}, \ominus(\ominus(\ominus(\mathbf{true})), \mathbf{false}), \mathbf{false})\right) \\
&= \ominus\left(\mathbf{false}, \ominus(\ominus(\mathbf{false}, \mathbf{false}), \mathbf{false}), \mathbf{false})\right) \\
&= \ominus(\mathbf{false}, \ominus(\mathbf{true}, \mathbf{false})) \\
&= \ominus(\mathbf{false}, \mathbf{false}) \\
&= \mathbf{true}
\end{aligned}$$

Für die Praxis ist die eben durchgeführte Rechnung viel zu umständlich. Stattdessen wird der Wert einer Formel direkt mit Hilfe der Tabelle 3.1 auf Seite 47 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 eine Tabelle auf, die für jede in der Formel auftretende Teilformel eine Spalte enthält. Tabelle 3.2 auf Seite 48 zeigt die entstehende Tabelle.

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

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

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

Damit erhalten wir folgende Rechnung:

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

### 3.3.1 Implementierung in Setl2

Um die bisher eingeführten Begriffe nicht zu abstrakt werden zu lassen, entwickeln wir in SETL2 ein Programm, mit dessen Hilfe sich Formeln auswerten lassen. Dazu müssen wir uns zunächst überlegen, wie wir eine aussagenlogische Formel in SETL2 repräsentieren können. Zusammengesetzte Daten-Strukturen können in SETL2 am einfachsten als Listen dargestellt werden und das ist auch der Weg, den wir für aussagenlogische Formeln beschreiten werden. Wir definieren die Repräsentation von aussagenlogischen Formeln durch Induktion:

1.  $\top$  wird repräsentiert durch die Zahl 1.
2.  $\perp$  wird repräsentiert durch die Zahl 0.
3. Eine aussagenlogische Variable  $p \in \mathcal{P}$  repräsentieren wir durch einen String, der den Namen der Variablen angibt.
4. Ist  $f$  eine aussagenlogische Formel, die durch die Liste  $L$  repräsentiert wird, so repräsentieren wir  $\neg f$  als zwei-elementige Liste, die als erstes Element den String “-” enthält:

$$[ "-", L ].$$

5. Sind  $f_1$  und  $f_2$  aussagenlogische Formel, die durch die Listen  $L_1$  und  $L_2$  repräsentiert werden, so repräsentieren wir  $f_1 \vee f_2$  als drei-elementige Liste, die als erstes Element den String “+” enthält:

$$[ "+", L_1, L_2 ].$$

6. Sind  $f_1$  und  $f_2$  aussagenlogische Formel, die durch die Listen  $L_1$  und  $L_2$  repräsentiert werden, so repräsentieren wir  $f_1 \wedge f_2$  als drei-elementige Liste, die als erstes Element den String “\*” enthält:

$$[ "*", L_1, L_2 ].$$

7. Sind  $f_1$  und  $f_2$  aussagenlogische Formel, die durch die Listen  $L_1$  und  $L_2$  repräsentiert werden, so repräsentieren wir  $f_1 \rightarrow f_2$  als drei-elementige Liste, die als erstes Element den String “->” enthält:

$$[ "->", L_1, L_2 ].$$

8. Sind  $f_1$  und  $f_2$  aussagenlogische Formel, die durch die Listen  $L_1$  und  $L_2$  repräsentiert werden, so repräsentieren wir  $f_1 \leftrightarrow f_2$  als drei-elementige Liste, die als erstes Element den String “<->” enthält:

$$[ "<->", L_1, L_2 ].$$

Als nächstes geben wir an, wie wir eine aussagenlogische Interpretation in SETL2 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  $b \in \mathbb{B}$ . 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 50 gezeigt.

Das Programm verwendet eine Kontroll-Struktur, die wir noch nicht besprochen haben, den **case**-Block. Ein solcher Block hat die in Abbildung 3.2 auf Seite 50 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

---

```

1  procedure eval(f, I);
2      case
3          when f = 1      => return TRUE;
4          when f = 0      => return FALSE;
5          when is_string(f) => return I(f);
6          when f(1) = "-" => return not eval(f(2), I);
7          when f(1) = "*" => return eval(f(2), I) and eval(f(3), I);
8          when f(1) = "+" => return eval(f(2), I) or eval(f(3), I);
9          when f(1) = ">" => return not eval(f(2), I) or eval(f(3), I);
10         when f(1) = "<-" => return eval(f(2), I) = eval(f(3), I);
11         otherwise => print("eval: Syntax-Fehler: ", f);
12     end case;
13 end eval;

```

---

Abbildung 3.1: Auswertung einer aussagenlogischen Formel.

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

```

---

Abbildung 3.2: Struktur eines Case-Blocks

In der Sprache C gibt es eine zum **case**-Block analoge Konstruktion. Dort heißt das Schlüsselwort, was dem “**case**” in SETL2 entspricht, aber “**switch**” und das Schlüsselwort, was dem “**when**” in SETL2 entspricht, heißt “**case**”!

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

1. Falls das Argument  $f$  die Formel  $\top$  repräsentiert, so ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation  $I$  immer **true**.
2. Falls das Argument  $f$  die Formel  $\perp$  repräsentiert, so ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation  $I$  immer **false**.
3. In Zeile 5 betrachten wir den Fall, dass das Argument  $f$  eine aussagenlogische Variable repräsentiert. Dies erkennen wir mit Hilfe der Bibliotheks-Funktion `is_string()`, 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  $[ \text{"*"}, g_1, g_2 ]$  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 **and**.
6. In Zeile 8 betrachten wir den Fall, dass  $f$  die Form  $[ \text{"+"}, g_1, g_2 ]$  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 **or**.
7. In Zeile 9 betrachten wir den Fall, dass  $f$  die Form  $[ \text{"->"}, g_1, g_2 ]$  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 nützen dann die folgende Eigenschaft der Funktion  $\mathcal{I}$  aus:

$$\mathcal{I}(p \rightarrow q) = \neg \mathcal{I}(p) \vee \mathcal{I}(q).$$

8. In Zeile 10 führen wir die Auswertung einer Formel  $g_1 \leftrightarrow g_2$  zurück auf die Auswertungen der Formeln  $g_1 \rightarrow g_2$  und  $g_2 \rightarrow g_1$ .
9. Wenn keiner der vorhergehenden Fälle greift, liegt ein Syntax-Fehler vor, auf den wir in Zeile 12 hinweisen.

### 3.3.2 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 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 vierten 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 3.3 zeigt ein solches Programm.

---

```

1  program main;
2      f1 := [ "+", [ "+", "a", "b" ], "c" ];
3      f2 := [ "->", "a", [ "+", "b", "c" ] ];
4      f3 := [ "->", "a", [ "-", [ "*", "b", "c" ] ] ];
5      f4 := [ "->", [ "-", "b" ], [ "-", "c" ] ];
6      f5 := [ "-", [ "*", [ "*", "a", "b" ], [ "-", "c" ] ] ];
7      f6 := [ "->", [ "-", "c" ], "a" ];
8
9      F := { f1, f2, f3, f4, f5, f6 };
10
11     A := { "a", "b", "c" };
12     P := pow A;
13     B := { createBelegung(M, A) : M in P };
14
15     S := { I in B | evalSet(F, I) = true };
16     print("S = ", S);
17
18     -- Diese Prozedur erzeugt aus einer Teilmenge M von A eine Belegung,
19     -- die genau für die Elemente x aus A true liefert, für die x ein
20     -- Element von M ist.
21     procedure createBelegung(M, A);
22         return { [ x, x in M ] : x in A };
23     end createBelegung;
24
25     -- F ist eine Menge von Formeln und I ist eine Belegung. Die
26     -- Funktion liefert genau dann wahr, wenn eval(f, I) für alle
27     -- Formeln f aus F true liefert.
28     procedure evalSet(F, I);
29         return { eval(f, I) : f in F } = { true };
30     end evalSet;
31
32     :
33 end main;

```

---

Abbildung 3.3: Programm zur Aufklärung des Einbruchs.

Wir diskutieren diese Programm nun Zeile für Zeile.

1. In den Zeilen 2 – 7 definieren wir die Formeln  $f_1, \dots, f_6$ . Wir müssen hier die Formeln in die SETL2-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  $\{a, b, c\}$ .

Wir berechnen daher in Zeile 12 zunächst die Menge aller dieser Teilmengen.

- Wir brauchen jetzt eine Möglichkeit, eine Teilmenge in eine Belegung umzuformen. In den Zeilen 21 – 23 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

$$M = \{a, c\}$$

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

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

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  $\mathcal{I}$ . Diese beiden Fälle können wir 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 22.

- In Zeile 13 sammeln wir in der Menge  $B$  alle möglichen Belegungen auf.
- In Zeile 28 – 30 definieren wir eine Funktion, die für eine Menge von Formeln  $F$  und eine gegebene Belegungen  $I$  entscheidet, ob die Belegungen  $I$  alle Formeln aus  $F$  wahr macht. Zu diesem Zweck wertet die Funktion alle Formeln aus  $F$  mit der gegebenen Belegung  $I$  aus und prüft, ob die Menge der dabei entstehenden Wahrheitswerte nur aus dem Wert **true** besteht.
- In Zeile 15 sammeln wir schließlich die Belegungen auf, die alle Formeln aus  $F$  wahr machen und geben die Menge dieser Belegungen in Zeile 16 aus.
- Bei der Implementierung der Funktion `evalSet` benutzen wir die Funktion `eval`, die wir bereits früher implementiert haben. Diese Implementierung ist durch die Pünktchen in Zeile 32 angedeutet.

Lassen wir das Programm laufen, so erhalten wir

$$S = \{ \{["a", \text{FALSE}], ["c", \text{TRUE}], ["b", \text{TRUE}]\} \}$$

Wir sehen, dass die Menge der Belegungen nur aus einem Element besteht. 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 4 (Tautologie)** Ist  $f$  eine aussagenlogische Formel und gilt

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

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:**

- $\models p \vee \neg p$
- $\models p \rightarrow p$

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

Die letzten beiden Beispiele geben Anlaß zu einer neuen Definition.

**Definition 5 (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 Setl2

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 SETL2-Programm entwickeln, mit dessen Hilfe wir die obige Frage automatisch lösen können. Die in Abbildung 3.4 auf Seite 55 gezeigte Prozedur `tautology` testet, ob für eine Formel  $f$  eine Tautologie ist. Diese Prozedur verwendet die Prozedur `eval` aus dem in Abbildung 3.1 auf Seite 50 gezeigten Programm. Wir diskutieren dieses Programm Zeile für Zeile:

---

```

1  -- Test, ob Formel f eine Tautologie ist.
2  procedure tautology(f);
3      V := collectVars(f);
4      A := { { [x, x in M] : x in V } : M in pow V };
5      return { eval(f, I) : I in A } = { true };
6  end tautology;
7
8  -- Diese Funktion sammelt alle Variablen in f auf.
9  procedure collectVars(f);
10     case
11         when f = 1      => return {};
12         when f = 0      => return {};
13         when is_string(f) => return { f };
14         when #f = 2      => return collectVars( f(2) );
15         when #f = 3      => return collectVars( f(2) ) + collectVars( f(3) );
16         otherwise       => print("collectVars: Syntax-Fehler: ", f);
17     end case;
18 end collectVars;

```

---

Abbildung 3.4: Überprüfung der Allgemeingültigkeit einer aussagenlogischen Formel.

1. In Zeile 3 sammeln wir alle aussagenlogischen Variablen auf, die in der zu überprüfenden Formel auftreten. Die dazu benötigte Prozedur `collectVars` ist in den Zeilen 9 – 18 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

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

beschränken.

2. In Zeile 4 berechnen wir die Menge aller aussagenlogischen Interpretationen über der Menge  $V$  der Variablen. Die Idee ist hierbei, dass die Menge aller aussagenlogischen Interpretationen isomorph zu der Potenz-Menge  $2^V$  von  $V$  ist: Haben wir eine Menge  $M \subseteq V$  gegeben, so können wir daraus eine aussagenlogische Interpretation  $\mathcal{I}_M$  gewinnen, indem wir definieren:

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

Diese Formel können wir noch vereinfachen zu

$$\mathcal{I}_M := \{ \langle x, x \in M \rangle \mid x \in V \}.$$

Umgekehrt können wir jeder aussagenlogischen Interpretation  $\mathcal{I}$  eine Teilmenge  $M_{\mathcal{I}} \subseteq V$  zuordnen:

$$M_{\mathcal{I}} := \{ x \in V \mid \mathcal{I}(x) = \text{true} \}.$$

Daher gibt es genau so viele Teilmengen von  $V$  wie es aussagenlogische Interpretationen auf der Menge  $V$  gibt und die Menge  $A$  in Zeile 4 enthält tatsächlich alle für die Auswertung der Formel relevanten aussagenlogischen Interpretationen.

Betrachten wir zur Verdeutlichung als Beispiel die Formel

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

Die Menge  $V$  der aussagenlogischen Variablen, die in dieser Formel auftreten, ist

$$V = \{p, q\}.$$

Die Potenz-Menge der Menge  $V$  ist



$$2^V = \{\{\}, \{p\}, \{q\}, \{p, q\}\}.$$

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 aber nun alle möglichen Interpretationen gewonnen.

3. In Zeile 5 berechnen wir zunächst die Menge

$$\{\text{eval}(f, I) : I \in A\}.$$

Hier ist  $A$  die Menge aller möglichen aussagenlogischen Belegungen, die wir in der vorhergehenden Zeile ausgerechnet haben. Wenn sich für jede der Belegungen  $I$  aus  $A$  bei der Auswertung von  $f$  der Wert **true** ergibt, dann ist  $f$  eine Tautologie. Dies ist aber genau dann der Fall, wenn die obige Menge nur aus dem Element **true** besteht.

### 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 der Distributivität 8 Zeilen. 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 6 (Literal)** Eine 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  $\mathcal{L}$ . □

**Definition 7 (Klausel)** Eine Formel  $k$  ist eine Klausel wenn  $k$  die Form

$$k = L_1 \vee \cdots \vee L_r$$

hat, wobei  $L_i$  für alle  $i = 1, \dots, r$  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 des Auftretens der Literale. Dies ist möglich aufgrund der Assoziativität, Kommutativität und Idempotenz des Junktors “ $\vee$ ”. Für die Klausel  $L_1 \vee \cdots \vee L_r$  schreiben wir also in Zukunft auch

$$\{L_1, \dots, L_r\}.$$

**Definition 8** 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  $\neg p \in k$ . □

**Satz 9** Eine Klausel ist genau dann eine Tautologie, wenn sie trivial ist.

**Beweis:** Unter Ausnutzung von Assoziativität und Kommutativität des Junktors “ $\vee$ ” und der Tatsache, dass “ $\perp$ ” ein neutrales Element bezüglich “ $\vee$ ” ist, kann jede nicht-triviale Klausel  $k$  in der Form

$$k \leftrightarrow \neg p_1 \vee \cdots \neg p_m \vee q_1 \vee \cdots \vee q_n$$

geschrieben werden, wobei zusätzlich  $p_i \neq q_j$  für alle  $i = 1, \dots, m$  und  $j = 1, \dots, n$  gilt. Definieren wir eine Interpretation  $\mathcal{I}$  durch

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

so gilt offenbar  $\mathcal{I}(k) = \mathbf{false}$  und damit kann  $k$  keine Tautologie sein. Da andererseits jede triviale Klausel offensichtlich eine Tautologie ist, ist der Beweis abgeschlossen.  $\square$

**Definition 10 (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.  $\square$

Aus der Definition der KNF folgt sofort das folgende.

**Korollar 11** Ist  $f = k_1 \wedge \cdots \wedge k_n$  in konjunktiver Normalform, so gilt

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

Ist eine Formel  $f = k_1 \wedge \cdots \wedge k_n$  in konjunktiver Normalform, so repräsentieren wir diese Formel durch die Menge ihrer Klauseln und schreiben

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

Ist  $f = k_1 \wedge \cdots \wedge k_n$  eine Formel in KNF, so ist  $f$  nach Satz 9 und dem Korollar 11 genau dann eine Tautologie, wenn alle Klauseln  $k_i$  trivial sind. Wir stellen nun ein Verfahren vor, mit dem sich jede Formel in KNF transformieren läßt. Nach dem eben Gesagten können wir dann sofort entscheiden, ob  $f$  eine Tautologie ist.

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

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

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

$$\models (p \rightarrow q) \leftrightarrow \neg p \vee q$$

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

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

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

$$(c) \models \neg \neg p \leftrightarrow p$$

$$(d) \models \neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$$

$$(e) \models \neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$$

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 des Distributiv-Gesetzes

$$\models p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$$

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 in diesem letzten Schritt stark anwachsen kann. Das liegt daran, dass die Formel  $p$  auf der rechten Seite der Äquivalenz  $p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$  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 Set12

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 zur Elimination des Junktors “ $\leftrightarrow$ ”. Dazu stellen wir zunächst rekursive Gleichungen auf, die das Verhalten der Funktion `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:

$$(a) \text{ elimGdw}(\top) = \top.$$

$$(b) \text{ elimGdw}(\perp) = \perp.$$

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

2. Hat  $f$  die Form  $f = \neg g$ , so eliminieren wir den Junktor “ $\leftrightarrow$ ” 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 “ $\leftrightarrow$ ” 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 “ $\leftrightarrow$ ” 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 3.5 auf Seite 60 zeigt die Implementierung der Prozedur `elimGdw`.

---

```

1  procedure elimGdw(f);
2      case
3          when f = 1          => return 1;
4          when f = 0          => return 0;
5          when is_string(f) => return f;
6          when f(1) = "-"    => return [ "-", elimGdw( f(2) ) ];
7          when f(1) = "*"    => return [ "*", elimGdw( f(2) ), elimGdw( f(3) ) ];
8          when f(1) = "+"    => return [ "+", elimGdw( f(2) ), elimGdw( f(3) ) ];
9          when f(1) = "->"  => return [ "->", elimGdw( f(2) ), elimGdw( f(3) ) ];
10         when f(1) = "<->" => return
11             elimGdw( [ "*", [ "->", f(2), f(3) ], [ "->", f(3), f(2) ] ] );
12         otherwise          => print("Fehler in elimGdw( ", f, " )");
13     end case;
14 end elimGdw;

```

---

Abbildung 3.5: Elimination von  $\leftrightarrow$ .

Als nächstes betrachten wir die Prozedur zur Elimination des Junktors “ $\rightarrow$ ”. Abbildung 3.6 auf Seite 61 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 Junktor “ $\leftrightarrow$ ” bereits vorher eliminiert wurde. Dadurch entfällt ein Fall.

Als nächstes zeigen wir die Routinen zur Berechnung der Negations-Normalform. Abbildung 3.7 auf Seite 62 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. 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.  $\text{nnf}(f_1 \wedge f_2) = \text{nnf}(f_1) \wedge \text{nnf}(f_2)$ .

---

```

1  procedure elimFolgt(f);
2      case
3          when f = 1      => return 1;
4          when f = 0      => return 0;
5          when is_string(f) => return f;
6          when f(1) = "-" => return [ "-", elimFolgt(f(2)) ];
7          when f(1) = "*" => return [ "*", elimFolgt(f(2)), elimFolgt(f(3)) ];
8          when f(1) = "+" => return [ "+", elimFolgt(f(2)), elimFolgt(f(3)) ];
9          when f(1) = "->" => return elimFolgt( [ "+", [ "-", f(2) ], f(3) ] );
10         otherwise      => print("Fehler in elimFolgt( ", f, ")" );
11     end case;
12 end elimFolgt;

```

---

Abbildung 3.6: Elimination von  $\rightarrow$ .

$$5. \text{nnf}(f_1 \vee f_2) = \text{nnf}(f_1) \vee \text{nnf}(f_2).$$

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

1.  $\text{neg}(\top) = \perp$
2.  $\text{neg}(\perp) = \top$
3.  $\text{neg}(p) = \neg p$  für alle Aussage-Variablen  $p$ .
4.  $\text{neg}(\neg f) = \text{nnf}(f)$ .
5.  $\text{neg}(f_1 \wedge f_2) = \text{neg}(f_1) \vee \text{neg}(f_2)$ .
6.  $\text{neg}(f_1 \vee f_2) = \text{neg}(f_1) \wedge \text{neg}(f_2)$ .

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 3.8 auf Seite 62 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  $\perp$ .

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 Konjunktion ist, haben wir die folgende Äquivalenz:

---

```

1  procedure nnf(f);
2      case
3          when f = 1      => return 1;
4          when f = 0      => return 0;
5          when is_string(f) => return f;
6          when f(1) = "-" => return neg( f(2) );
7          when f(1) = "*" => return [ "*", nnf( f(2) ), nnf( f(3) ) ];
8          when f(1) = "+" => return [ "+", nnf( f(2) ), nnf( f(3) ) ];
9          otherwise      => print("Fehler in nnf( ", f, ")" );
10     end case;
11 end nnf;
12
13 procedure neg(f);
14     case
15         when f = 1      => return 0;
16         when f = 0      => return 1;
17         when is_string(f) => return [ "-", f ];
18         when f(1) = "-" => return nnf( f(2) );
19         when f(1) = "*" => return [ "+", neg( f(2) ), neg( f(3) ) ];
20         when f(1) = "+" => return [ "-", neg( f(2) ), neg( f(3) ) ];
21         otherwise      => print("Fehler in neg( ", f, ")" );
22     end case;
23 end neg;

```

---

Abbildung 3.7: Berechnung der Negations-Normalform.

---

```

1  procedure knf(f);
2      case
3          when f = 1      => return { };
4          when f = 0      => return { {} };
5          when is_string(f) => return { { f } };
6          when f(1) = "-" => return { { f } };
7          when f(1) = "*" => return knf( f(2) ) + knf( f(3) );
8          when f(1) = "+" => return { k1+k2: k1 in knf(f(2)), k2 in knf(f(3)) };
9          otherwise      => print("Fehler in knf( ", f, ")" );
10     end case;
11 end knf;

```

---

Abbildung 3.8: Berechnung der konjunktiven Normalform.

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

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 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**.

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

$$\mathbf{k}\mathbf{n}\mathbf{f}(f_1) = \{h_1, \dots, h_m\} \quad \text{und} \quad \mathbf{k}\mathbf{n}\mathbf{f}(f_2) = \{k_1, \dots, k_n\}.$$

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

$$\text{knf}(f_1 \wedge f_2) = \text{knf}(f_1) \cup \text{knf}(f_2).$$

Das liefert Zeile 7 der Implementierung.

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

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

Dabei sind die  $h_i$  und die  $k_j$  Klauseln. Um nun die KNF von  $f_1 \vee f_2$  zu bilden, rechnen wir wie folgt:

$$\begin{aligned}
& f_1 \vee f_2 \\
\leftrightarrow & (h_1 \wedge \cdots \wedge h_m) \vee (k_1 \wedge \cdots \wedge k_n) \\
\leftrightarrow & (h_1 \vee k_1) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_1) \quad \wedge \\
& \qquad \vdots \qquad \qquad \qquad \qquad \qquad \vdots \\
& (h_1 \vee k_n) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_n) \\
\leftrightarrow & \{h_i \vee k_j : i \in \{1, \dots, m\}, j \in \{1, \dots, n\}\}
\end{aligned}$$

Berücksichtigen wir noch, dass Klauseln in der Mengen-Schreibweise als Mengen von Literalen aufgefaßt werden die implizit disjunktiv verknüpft werden, so können wir für  $h_i \vee k_j$  auch  $h_i \cup k_j$  schreiben. Insgesamt erhalten wir damit

$$\mathbf{k}\mathbf{n}\mathbf{f}(f_1 \vee f_2) = \{h \cup k : h \in \mathbf{k}\mathbf{n}\mathbf{f}(f_1), k \in \mathbf{k}\mathbf{n}\mathbf{f}(f_2)\}.$$

Das liefert die Zeile 8 der Implementierung der Prozedur `knf`.

Zum Abschluß zeigen wir in Abbildung 3.9 auf Seite 63 wie die einzelnen Funktionen zusammenspielen.

```

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

```

Abbildung 3.9: 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  $g$  auf  $f_1, \dots, f_n$  folgt, ob also gilt

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

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

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

Wir 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 12 (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:

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

**Beispiele für Schluß-Regeln:**

1. “*Modus Ponens*”:

$$\frac{p \quad p \rightarrow q}{q}$$

2. “*Modus Tollens*”:

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

3. “*Trug-Schluß*”:

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

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 13 (Korrekte Schluß-Regel)** Eine Schluß-Regel

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

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

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 14 (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*:

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

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*.

**Satz 15** 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:

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

Wir haben jetzt alles Material zusammen, um den Beweis-Begriff formalisieren zu können.

**Definition 16 ( $\vdash$ )** Es sei  $M$  eine Menge von Klauseln und  $f$  sei eine einzelne Klausel. Die Formel 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:  
Falls  $f \in M$ , dann  $M \vdash f$ .

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

Falls  $M \vdash k_1 \cup \{p\}$  und  $M \vdash \{\neg p\} \cup k_2$ , dann  $M \vdash k_1 \cup k_2$ . □

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

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

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  $\{\neg p, \neg p\} = \{\neg p\}$ . Wir schreiben 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  $p \rightarrow r$  folgt. Dazu überführen wir zunächst alle Formeln in Klauseln:

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

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, die wir nun formulieren werden.

**Satz 17 (Korrektheit)** Ist  $\{k_1, \dots, k_n\}$  eine Menge von Klauseln und  $k$  eine einzelne Klausel, so haben wir:

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

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

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

Haben wir also eine Menge von Klauseln  $M = \{k_1, \dots, k_n\}$  gegeben und wollen zeigen, dass eine Formel  $f$  aus  $M$  folgt, so wird es im allgemeinen nicht gelingen,  $f$  direkt aus  $M$  herzuleiten. Wir können uns aber mit einem Trick behelfen.: Es gilt

$$\models k_1 \wedge \dots \wedge k_n \rightarrow f \quad \text{g.d.w.} \quad \models k_1 \wedge \dots \wedge k_n \wedge \neg f \rightarrow \perp.$$

Anstatt also  $f$  herzuleiten, negieren wir  $f$ , überführen  $\neg f$  in konjunktive Normalform und fügen die Klauseln der Menge  $\mathbf{knf}(\neg f)$  den ursprünglichen Annahmen in der Menge  $M$  hinzu. Wenn sich nun aus  $M \cup \{\mathbf{knf}(\neg f)\}$  das  $\perp$  herleiten lässt, dann folgt  $f$  aus  $M$ :

$$\models k_1 \wedge \dots \wedge k_n \rightarrow f \quad \text{g.d.w.} \quad \{k_1, \dots, k_n\} \cup \mathbf{knf}(\neg f) \vdash \{\}.$$

Wir erläutern das Verfahren mit einem Beispiel. Wir wollen zeigen, dass aus

$$u \rightarrow s \vee t, \quad s \rightarrow p, \quad \text{und} \quad t \rightarrow p,$$

die Formel

$$u \rightarrow p$$

folgt. Die Umwandlung der Annahmen in Klauseln ist trivial, wir erhalten

$$M = \{\{\neg u, s, t\}, \{\neg s, p\}, \{\neg t, p\}\}.$$

Jetzt negieren wir die Formel  $\neg(u \rightarrow p)$  und wandeln die Negation in konjunktive Normalform um. Es gilt

$$\mathbf{knf}(\neg(u \rightarrow p)) = \{\{u\}, \{\neg p\}\}.$$

Anschließend müssen wir die beiden Klauseln, die wir bei der Berechnung der konjunktiven Normalform gefundenen haben, zu der Menge  $M$  hinzufügen. Wir bilden also die Menge

$$N := M \cup \mathbf{knf}(\neg(u \rightarrow p)) = \{\{\neg u, s, t\}, \{\neg s, p\}, \{\neg t, p\}, \{u\}, \{\neg p\}\}.$$

Schließlich müssen wir aus der Menge  $N$  mit Hilfe der Schnitt-Regel die leere Klausel, die ja der Formel  $\perp$  entspricht, herleiten:

1.  $\{u\}, \{\neg u, s, t\} \vdash \{s, t\}$
2.  $\{s, t\}, \{\neg s, p\} \vdash \{t, p\}$
3.  $\{t, p\}, \{\neg p\} \vdash \{t\}$
4.  $\{t\}, \{\neg t, p\} \vdash \{p\}$
5.  $\{p\}, \{\neg p\} \vdash \{\}$

### 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

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

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 [1, 2] zurück. Verfeinerungen dieses Verfahrens werden beispielsweise eingesetzt, um die logische Korrektheit von digitalen elektronischen Schaltungen zu analysieren.

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

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

Offenbar ist  $K_1$  lösbar und die Belegung

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

ist eine Lösung. Betrachten wir ein weiteres Beispiel:

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

Das  $K_2$  die leere Klausel enthält und da die leere Klausel äquivalent zu  $\perp$  ist, ist  $K_2$  offenbar unlösbar. Wir nehmen diese Beobachtungen zum Anlaß für zwei Definitionen.

**Definition 19 (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 20 (Triviale Klausel-Mengen)** Eine Klausel-Menge  $K$  heißt *trivial* wenn einer der beiden folgenden Fälle vorliegt.

1.  $K$  enthält die leere Klausel.

In diesem Fall ist  $K$  offensichtlich unlösbar.

2.  $K$  enthält nur Unit-Klausel mit verschiedenen Aussage-Variablen.

Dann ist

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

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  $n + 1$  und als  $m + 1$ . Die Klauseln wachsen nur dann 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*.

### 3.6.2 Vereinfachung durch Subsumption

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

$$K_1 = \{ \{p, q, \neg r\}, \{p\} \} \cup M.$$

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  $k$ , die mindestens zwei Literale enthält, 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. Wir schreiben

$$K \rightsquigarrow K - \{k\}.$$

Falls eine Klausel-Menge eine Unit-Klausel  $u$  enthält, so können wir alle die Klauseln  $k \in K$  löschen, für die gilt

$$u \subseteq k \quad \text{und} \quad \#k \geq 2.$$

### 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 21** Ist  $K$  eine Menge von Klauseln und ist  $p$  eine aussagenlogische Variable, die in einer Klausel aus  $K$  vorkommt, so ist  $K$  genau dann erfüllbar, wenn  $K \cup \{\{p\}\}$  oder  $K \cup \{\{\neg p\}\}$  erfüllbar ist.

Beweis: 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. Ist andererseits  $K$  erfüllbar durch eine Belegung  $\mathcal{I}$ , so muss die Belegung  $\mathcal{I}$  auch der aussagenlogischen Variable  $p$  einen Wahrheitswert zuweisen, denn diese Variable tritt ja in  $K$  auf. Falls  $\mathcal{I}(p) = \text{true}$  ist, ist damit auch die Menge  $K \cup \{\{p\}\}$  erfüllbar, andernfalls ist  $K \cup \{\{\neg p\}\}$  erfüllbar.  $\square$

Wir können nun eine Menge  $K$  von Klauseln dadurch vereinfachen, dass wir eine aussagenlogische Variable  $p$  suchen, 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 von Davis und Putnam

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:

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

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  $K$ .

Um das Verfahren von Davis und Putnam durchzuführen ist es zweckmäßig, zwei Hilfsfunktionen zu definieren. Wir bezeichnen im folgenden die Menge der Klauseln mit  $\mathcal{K}$  und die Menge der Literale mit  $\mathcal{L}$ . Zunächst definieren wir die Funktion

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

die als Argumente eine Klausel-Menge  $K$  und ein Literal  $l$  erhält. Der Funktions-Aufruf

$$\text{unitCut}(K, l)$$

führt dann alle Unit-Schnitte, die mit der Unit-Klausel  $\{l\}$  und Klauseln aus  $K$  möglich sind, durch und liefert als Ergebnis die entsprechend veränderte Klausel-Menge  $K$  zurück. Formal definieren wir

$$\text{unitCut}(K, l) = \{k - \{\neg l\} \mid k \in K \wedge (\neg l) \in k\} \cup \{k \mid k \in K \wedge (\neg l) \notin k\}.$$

Die Menge  $\text{unitCut}(K, l)$  besteht also aus zwei Teilen:

1. Der erste Teil entsteht dadurch, dass wir Unit-Schnitte mit der Klausel  $\{l\}$  durchführen. Dazu wählen wir die Klauseln  $k$  aus  $K$  aus, die das Literal  $\neg l$  enthalten.<sup>1</sup> Solche Klauseln haben die Form  $k = k' \cup \{\neg l\}$ , es gilt also  $k' = k - \{\neg l\}$ . Für diese Klauseln hat die Schnitt-Regel die Form

$$\frac{k' \cup \{\neg l\} \quad \{l\}}{k'},$$

von diesen Klauseln bleibt also nur  $k' = k - \{\neg l\}$  über.

2. Der zweite Teil der Menge enthält die restlichen Klauseln aus  $K$ , die unverändert bleiben. Die restlichen Klauseln sind natürlich gerade die Klauseln, die das Literal  $\neg l$  nicht enthalten.

Als nächstes definieren wir die Funktion

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

Für eine Klausel-Menge  $K$  und ein Literal  $l$  berechnet der Aufruf

$$\text{unitSubsumption}(K, l)$$

gerade die Vereinfachung der Menge  $K$ , die sich durch Unit-Subsumption mit der Unit-Klausel  $\{l\}$  ergibt. Formal gilt:

$$\text{unitSubsumption}(K, l) = \{k \mid k \in K \wedge l \notin k\}.$$

Für die Implementierung ist es zweckmäßig, die beiden Funktionen  $\text{unitCut}$  und  $\text{unitSubsumption}$  zusammen zu fassen. Wir definieren

$$\begin{aligned} \text{reduce}(K, l) &= \text{unitSubsumption}(\text{unitCut}(K, l), l) \\ &= \{k - \{\neg l\} \mid k \in K \wedge (\neg l) \in k\} \cup \{k \mid k \in K \wedge (\neg l) \notin k \wedge l \notin k\}. \end{aligned}$$

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  $(\neg l) \in k$  gilt.

### 3.6.5 Ein Beispiel

Zur Veranschaulichung demonstrieren wir das Verfahren von Davis und Putnam an einem Beispiel. Die Menge  $M$  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

---

<sup>1</sup>Streng genommen ist das nicht ganz sauber. Ist  $l$  das Literal  $\neg p$ , so gilt eigentlich  $\neg l = \neg\neg p$  und das ist kein Literal. Da aber  $\neg\neg p$  zu  $p$  äquivalent ist, identifizieren wir  $\neg\neg p$  mit  $p$ .

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

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

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

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

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

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

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

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

2. Also bilden wir jetzt die Menge

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

und versuchen, diese Menge zu lösen. Dazu bilden wir

$$K_1 = \text{reduce}(K, \{\neg p\}) = \left\{ \{q, s\}, \{r, s\}, \{\neg s\}, \{\neg q, s\}, \{\neg r, \neg s\}, \{\neg p\} \right\}.$$

Die Menge  $K_1$  enthält die Unit-Klausel  $\{\neg s\}$ . Wir bilden daher

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

Die Menge  $K_2$  enthält die neue Unit-Klausel  $\{q\}$ , mit der wir als nächstes reduzieren:

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

Da  $K_3$  die leere Klausel enthält, ist  $K_3$  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 63 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 **Result**, so dass jede Belegung  $\mathcal{I}$ , die alle Unit-Klauseln aus **Result** 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 den Wert **false** zurück.

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 bilden wir solange wie möglich Unit-Schnitte mit den Klauseln aus der Menge **Clauses** 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 **false** zurück.
3. Dann testen wir in Zeile 6, ob bereits alle Klauseln  $k$  aus der Menge **Clauses** Unit-Klauseln sind. Wenn dies so ist, dann ist **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

$$\text{Clauses} \cup \{\{\text{literal}\}\}$$



lösbar ist. Dann gibt es zwei Fälle:

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

$\text{Clauses} \cup \{\{\neg \text{literal}\}\}$   
lösbar ist.

---

```

1  procedure DavisPutnam( Clauses, Literals );
2      Clauses := saturate(Clauses);
3      if {} in Clauses then
4          return false;
5      end if;
6      if { k in Clauses | #k = 1 } = Clauses then
7          return Clauses;
8      end if;
9      literal := selectLiteral(Clauses, Literals);
10     Result := DavisPutnam(Clauses + {{literal}}, Literals + { literal });
11     if Result /= false then
12         return Result;
13     end if;
14     notLiteral := negateLiteral(literal);
15     return DavisPutnam(Clauses + {{notLiteral}}, Literals + { notLiteral });
16 end DavisPutnam;
```

---

Abbildung 3.10: 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 3.11 auf Seite 72 gezeigt.

---

```

1  procedure saturate(S);
2      Units := { k in S | #k = 1 };
3      Used := {};
4      while Units /= {} loop
5          unit := arb Units;
6          Used := Used + { unit };
7          literal := arb unit;
8          S := reduce(S, literal);
9          Units := { k in S | #k = 1 } - Used;
10     end loop;
11     return S;
12 end saturate;
```

---

Abbildung 3.11: 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 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 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 neue Unit-Klauseln finden.

Die dabei verwendete Prozedur **reduce** ist in Abbildung 3.12 gezeigt. Die Implementierung setzt die im vorigen Abschnitt gegebene Definition der Funktion **reduce** unmittelbar um.

---

```

1  procedure reduce( S, l );
2      notL := negateLiteral(l);
3      return { k - { notL } : k in S | notL in k } +
4              { k : k in S | not notL in k and (not l in k or k = {l}) };
5  end reduce;
```

---

Abbildung 3.12: Die Prozedur **reduce**.

Die Implementierung von **DavisPutnam** benutzt außer den bisher diskutierten Prozeduren noch zwei weitere Hilfsprozeduren, deren Implementierung in Abbildung 3.13 auf Seite 73 gezeigt wird.

1. Die Prozedur **selectLiteral** wählt ein beliebiges Literal aus einer gegebenen Menge  $S$  von Klauseln aus, das außerdem nicht in der Menge  $F$  von Literalen vorkommen darf, die bereits benutzt worden sind.
2. Die Prozedur **negateLiteral** bildet die Negation  $\neg l$  eines gegebenen Literals  $l$ . Falls das Literal  $l$  die Form  $\neg p$  hat, wir aber statt der Formel  $\neg\neg p$  das Literal  $p$  zurückgeben.

---

```

1  procedure selectLiteral( S, F );
2      return arb { l: k in S, l in k | not l in F };
3  end selectLiteral;
4
5  procedure negateLiteral(l);
6      if l(1) = "-" then
7          return l(2);
8      else
9          return [ "-", l ];
10     end if;
11 end negateLiteral;
```

---

Abbildung 3.13: 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 den folgenden Artikel verwiesen:

### 3.7 Das 8-Damen-Problem

In diesem Abschnitt zeigen wir, wie bestimmte kombinatorische Problem in aussagenlogische Probleme umformuliert werden können. Diese können anschließend mit dem Algorithmus von Davis und Putnam gelöst werden. Als konkretes Beispiel betrachten wir das 8-Damen-Problem. Dabei geht es darum, 8 Damen so auf einem Schach-Brett aufzustellen, dass keine Dame eine andere Dame schlagen kann. Beim Schach-Spiel kann eine Dame dann eine andere Figur schlagen falls diese Figur entweder

- in der selben Reihe,
- in der selben Spalte, oder
- in der selben Diagonale

steht. Abbildung 3.14 auf Seite 74 zeigt ein Schachbrett, in dem sich in der dritten Reihe 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.

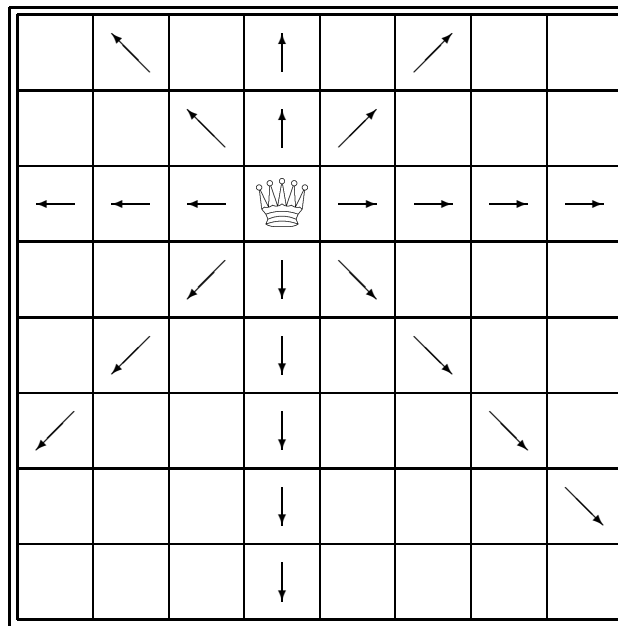


Abbildung 3.14: Das 8-Damen-Problem.

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 Reihe bezeichnet, erhält den Namen

$$p_{ij} \quad \text{mit } i, j \in \{1, \dots, 8\}.$$

Wir numerieren die Reihen dabei von oben beginnend von 1 bis 8 durch, während die Spalten von links nach rechts numeriert werden. Abbildung 3.15 auf Seite 75 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 3.15: Zuordnung der Variablen.

Um die obigen Überlegungen in SETL2 umzusetzen, implementieren wir die Prozedur `createBoard`, die das oben gezeigte Schach-Brett als Liste von Listen von Variablen berechnet. Abbildung 3.16 auf Seite 75 zeigt die Implementierung.

---

```

1  procedure createBoard(n);
2      return [ createRow(n, i) : i in [1..n] ];
3  end createBoard;
4
5  procedure createRow(n, i);
6      return [ "p" + i + j : j in [1..n] ];
7  end createRow;
```

---

Abbildung 3.16: 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. Diese Formel können wir unmittelbar in eine Klausel umformen. Wir erhalten:

$$\{\neg x_i, \neg x_j\}.$$

Die Formel, die für eine Variablen-Menge  $V$  ausdrückt, dass keine zwei verschiedenen Variablen gleichzeitig gesetzt sind, kann jetzt als Klausel-Menge wie folgt geschrieben werden:

$$\{\{\neg p, \neg q\} \mid p \in V \wedge q \in V \wedge p \neq q\}.$$

Da wir die Lösung des 8-Damen-Problems in SETL2 implementieren wollen, setzen wir die obigen Überlegungen sofort in eine Prozedur um. Die in Abbildung 3.17 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  procedure atMostOne(V);
2      return { { [ "-", p ], [ "-", q ] } : p in V, q in V | p /= q };
3  end atMostOne;
```

---

Abbildung 3.17: 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 Reihe *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 3.18 zeigt die Implementierung: Wir sammeln alle Variablen der gegebenen Reihe in der Menge  $S$  auf und rufen dann die Hilfs-Prozedur **atMostOne(S)** auf, die das Ergebnis als Menge von Klauseln liefert.

---

```

1  procedure atMostOneInRow(board, row);
2      S := { p : p in board(row) };
3      return atMostOne(S);
4  end atMostOneInRow;
```

---

Abbildung 3.18: Die Prozedur **atMostOneInRow**.

In analoger Weise können wir auch die Prozedur **atMostOneInColumn** implementieren. Der Aufruf

**atMostOneInColumn(board, column)**

berechnet für ein gegebenes Brett *board* und eine Zahl *column* eine Formel die ausdrückt, dass in der Spalte *column* höchstens eine Dame steht. Der Ausdruck zur Berechnung der Menge  $S$  ist hier etwas komplizierter, weil wir aus jeder Zeile von *board* das Element mit der Nummer *column* extrahieren müssen. Abbildung 3.19 zeigt die Implementierung.

---

```

1  procedure atMostOneInColumn(board, column);
2      n := #board;
3      S := { board(row)(column) : row in [1..n] };
4      return atMostOne(S);
5  end atMostOneInColumn;

```

---

Abbildung 3.19: Die Prozedur `atMostOneInColumn`.

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

p81, p72, p63, p54, p45, p36, p28, p81.

Die Indizes dieser Variablen  $i$  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  $\{2, \dots, 16\}$  annimmt. Diesen Wert  $k$  geben wir nun als Argument bei der Prozedur `atMostOneInUpperDiagonal` mit. Diese Prozedur ist in Abbildung 3.20 gezeigt.

---

```

1  procedure atMostOneInUpperDiagonal(board, k);
2      n := #board;
3      S := { board(r)(c) : r in [1..n], c in [1..n] | r + c = k };
4      return atMostOne(S);
5  end atMostOneInUpperDiagonal;

```

---

Abbildung 3.20: Die Prozedur `atMostOneInUpperDiagonal`.

Um zu sehen, wie die Variablen einer fallenden Diagonale charakterisiert werden können, betrachten wir die fallende Hauptdiagonale, die aus den Variablen

p11, p22, p33, p44, p55, p66, p77, p88

besteht. Die Indizes dieser Variablen  $i$  und  $j$  dieser Variablen  $p_{ij}$  erfüllen offenbar die Gleichung

$$i - j = 0.$$

Allgemein erfüllen die Indizes der Variablen einer absteigenden Diagonale die Gleichung

$$i - j = k,$$

wobei  $k$  einen Wert aus der Menge  $\{-7, \dots, 7\}$  annimmt. Diesen Wert  $k$  geben wir nun als Argument bei der Prozedur `atMostOneInLowerDiagonal` mit. Diese Prozedur ist in Abbildung 3.21 gezeigt.

Wir brauchen noch eine Formel die aussagt, dass mindestens eine Dame in einer gegebenen Reihe steht. Eine solche Formel erhalten wir als Disjunktion aller Variablen der Zeile. Schreiben wir diese Formel als Klausel in Mengenschreibweise, so erhalten wir einfach die Menge aller Variablen der Zeile. Abbildung 3.22 zeigt die Implementierung. Da wir das Ergebnis später als Menge von Klauseln auffassen wollen, geben wir eine Menge zurück, die als einziges Element die Klausel enthält, die aus allen Variablen der Zeile besteht.

An dieser Stelle erwarten Sie vielleicht, dass wir als nächstes eine Formel angeben die ausdrückt, dass in einer gegebenen Spalte mindestens eine Dame steht. Eine solche Formel ist aber unnötig,

---

```

1  procedure atMostOneInLowerDiagonal(board, k);
2      n := #board;
3      S := { board(r)(c) : r in [1..n], c in [1..n] | r - c = k };
4      return atMostOne(S);
5  end atMostOneInLowerDiagonal;

```

---

Abbildung 3.21: Die Prozedur `atMostOneInLowerDiagonal`.

---

```

1  procedure oneInRow(board, row);
2      return { { p : p in board(row) } };
3  end oneInRow;

```

---

Abbildung 3.22: Die Prozedur `oneInRow`.

denn wenn wir wissen, dass in jeder Reihe mindestens eine Dame steht, so wissen wir bereits, dass auf dem Brett 8 Damen stehen. Wenn wir nun zusätzlich wissen, dass in jeder Spalte höchstens eine Dame steht, so ist automatisch klar, dass in jeder Spalte genau eine Dame stehen muß.

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 3.23 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 4),
2. in jeder Spalte höchstens eine Dame steht (Zeile 5),
3. in jeder absteigenden Diagonale höchstens eine Dame steht (Zeile 6),
4. in jeder aufsteigenden Diagonale höchstens eine Dame steht (Zeile 7) und
5. in jeder Zeile mindestens eine Dame steht.

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.

---

```

1  procedure allClauses(board);
2      n := #board;
3      Clauses := {};
4      Clauses += { atMostOneInRow(board, row) : row in [1..n] };
5      Clauses += { atMostOneInColumn(board, column) : column in [1..n] };
6      Clauses += { atMostOneInLowerDiagonal(board, k) : k in [-(n-1) .. n-1] };
7      Clauses += { atMostOneInUpperDiagonal(board, k) : k in [2 .. 2*n] };
8      Clauses += { oneInRow(board, row) : row in [1 .. n] };
9      return Clauses;
10 end allClauses;

```

---

Abbildung 3.23: Die Prozedur `allClauses`.

Mit den bisher gezeigten Prozeduren und unserer Implementierung des Algorithmus von Davis und Putnam können wir nun das 8-Damen-Problem durch die folgenden Aufrufe lösen:

```
board    := createBoard(8);
Clauses  := allClauses(board);
I        := DavisPutnam(Clauses,{});
printBoard(I);
```

Hier benutzen wir noch die Prozedur `printBoard`, die als Argument eine triviale Menge von Unit-Klauseln  $I$  erhält und ein Schach-Brett `board` und die Lösung als sparsame ASCII-Graphik ausgibt. Abbildung 3.24 zeigt die Implementierung dieser Prozedur. In Zeile 7 überprüfen wir, ob die Variable für die Zeile `row` und die Spalte `col` gesetzt ist, denn die 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). Abbildung 3.25 zeigt die von dieser Prozedur erzeugte Ausgabe. Eine graphische Darstellung des selben Schach-Bretts sehen Sie in Abbildung 3.26.

---

```
1  procedure printBoard(I, board);
2      n := #board;
3      print( "          " + ((2*n+1) * "-" ) );
4      for row in [1..n] loop
5          line := "          |";
6          for col in [1..n] loop
7              if { board(row)(col) } in I then
8                  line += "Q|";
9              else
10                 line += " |";
11             end if;
12         end loop;
13         print(line);
14         print( "          " + ((2*n+1) * "-" ) );
15     end loop;
16 end printBoard;
```

---

Abbildung 3.24: Die Prozedur `printBoard`.

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 eine Sekunde<sup>2</sup>!

In der Praxis gibt es Probleme, die sich in ganz ähnlicher Weise auf die Lösung einer Menge von Klauseln zurückführen lassen. Dazu gehört zum Beispiel die Erstellung eines Stundenplans oder ganz allgemein *Scheduling-Probleme* aus dem Projekt-Management.

---

<sup>2</sup>Das System, auf dem der Test lief, war mit einem auf 2,4 Ghz getakteten Pentium-IV-Prozessor und 1 GB Hauptspeicher bestückt.



---

1	-----
2	Q
3	-----
4	Q
5	-----
6	Q
7	-----
8	Q
9	-----
10	Q
11	-----
12	Q
13	-----
14	Q
15	-----
16	Q
17	-----

---

Abbildung 3.25: Ausgabe der Prozedur `printBoard`.

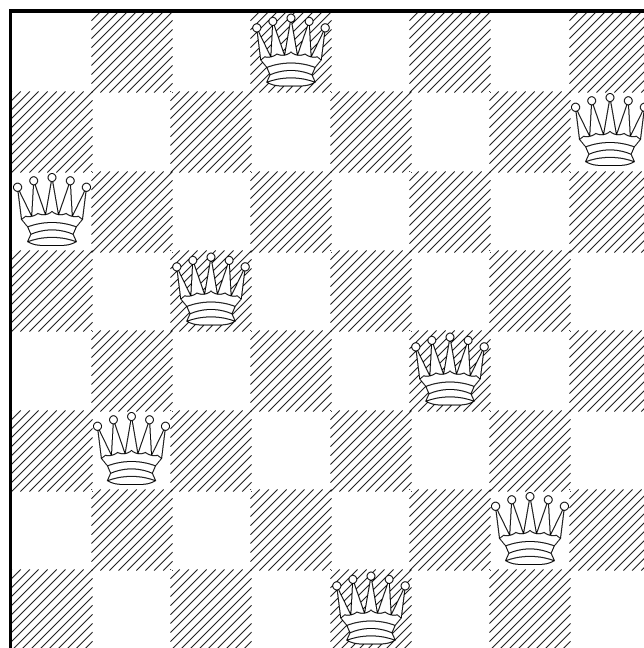


Abbildung 3.26: Eine Lösung des 8-Damen-Problems.

# 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.
3. Verschiedene Objekte werden durch *Prädikats-Zeichen* in Relation gesetzt.
4. Schließlich werden *Quantoren* eingeführt, um Aussagen über Mengen von Objekten formulieren zu können.

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 Mengen von Variablen, Funktions- und Prädikats-Zeichen, sowie deren Stelligkeit.

**Definition 22 (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  $f$   $n$ -stellig ist falls  $\text{arity}(f) = n$ .

5. Die Mengen  $\mathcal{V}$ ,  $\mathcal{F}$  und  $\mathcal{P}$  sind paarweise disjunkt:

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

Als Bezeichner für Objekte verwenden wir *Terme*. Formal werden Terme wie folgt definiert.

**Definition 23 (Terme)** Ist  $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$  eine Signatur, so definieren wir die Menge der  $\Sigma$ -Terme  $\mathcal{T}_\Sigma$  induktiv:

1. Für jede Variable  $x \in \mathcal{V}$  gilt  $x \in \mathcal{T}_\Sigma$ .
2. Ist  $f \in \mathcal{F}$  ein  $n$ -stelliges Funktions-Zeichen und sind  $t_1, \dots, t_n \in \mathcal{T}_\Sigma$ , 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  $\mathcal{F} = \{0, 1, +, -, *\}$ , und die Menge der Prädikats-Zeichen  $\mathcal{P} = \{=, \leq\}$  gegeben. Ferner sei die Funktion  $\text{arity}$  als die Relation

$$\text{arity} = \{ \langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle +, 2 \rangle, \langle -, 2 \rangle, \langle *, 2 \rangle \}$$

definiert. Schließlich sei die Signatur  $\Sigma_{\text{arith}}$  durch das 4-Tupel  $\langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$  gegeben.

Dann können wir wie folgt  $\Sigma_{\text{arith}}$ -Terme konstruieren:

1.  $x, y, z \in \mathcal{T}_{\Sigma_{\text{arith}}}$ ,  
denn alle Variablen sind auch  $\Sigma_{\text{arith}}$ -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 24 (Atomare Formeln)** Gegeben sei eine Signatur  $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ . Die Menge der atomaren  $\Sigma$ -Formeln  $\mathcal{A}_\Sigma$  wird wie folgt definiert: Ist  $p \in \mathcal{P}$  ein  $n$ -stelliges Prädikats-Zeichen und sind  $n$   $\Sigma$ -Terme  $t_1, \dots, t_n$  gegeben, so ist  $p(t_1, \dots, t_n)$  eine atomare  $\Sigma$ -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  $\Sigma$ -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:

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

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

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

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

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

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

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

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

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

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

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

und diese Formel ist nicht mehr gültig. Um das Problem näher zu analysieren, definieren wir für einen  $\Sigma$ -Term  $t$  die Menge der in  $t$  enthaltenen Variablen.

**Definition 25 ( $\text{Var}(t)$ )** Ist  $t$  ein  $\Sigma$ -Term, so definieren wir die Menge  $\text{Var}(t)$  der Variablen, die in  $t$  auftreten, durch Induktion nach dem Aufbau des Terms:

1. Ist  $t = x$  mit  $x \in \mathcal{V}$ , so gilt

$$\text{Var}(t) := \{x\}.$$

2. Ist  $t = f(t_1, \dots, t_n)$  so definieren wir

$$\text{Var}(t) := \text{Var}(t_1) \cup \dots \cup \text{Var}(t_n)$$

□

**Definition 26 ( $\Sigma$ -Formel, gebundene und freie Variablen)**

Es sei  $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$  eine Signatur. Die Menge der  $\Sigma$ -Formeln bezeichnen wir mit  $\mathbb{F}_\Sigma$ . Wir definieren diese Menge induktiv. Gleichzeitig definieren wir für jede Formel  $F \in \mathbb{F}_\Sigma$  die Menge  $BV(F)$  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

$$FV(\perp) := FV(\top) := BV(\perp) := BV(\top) := \emptyset$$

2. Ist  $F = p(t_1, \dots, t_n)$  eine atomare  $\Sigma$ -Formel, so gilt  $F \in \mathbb{F}_\Sigma$ . Weiter definieren wir:

$$(a) \quad FV(p(t_1, \dots, t_n)) := \text{Var}(t_1) \cup \dots \cup \text{Var}(t_n).$$

$$(b) \quad BV(p(t_1, \dots, t_n)) := \emptyset.$$

3. Ist  $F \in \mathbb{F}_\Sigma$ , so gilt  $\neg F \in \mathbb{F}_\Sigma$ . Weiter definieren wir:

$$(a) \quad FV(\neg F) := FV(F).$$

$$(b) \quad BV(\neg F) := BV(F).$$

4. Sind  $F, G \in \mathbb{F}_\Sigma$  und gilt außerdem

$$FV(F) \cap BV(G) = \emptyset \quad \text{und} \quad FV(G) \cap BV(F) = \emptyset,$$

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

**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

$$\begin{aligned} \forall x, y: p(x, y) & \quad \text{statt} \\ \forall x: (\forall y: p(x, y)). \end{aligned}$$

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 obige Formel  $(\exists x: \leq(+ (y, x), y))$  wird dann lesbarer als

$$\exists x: y + x = y$$

geschrieben.

## 4.2 Semantik der Prädikatenlogik

**Definition 27 (Struktur)** Es sei eine Signatur

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

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

1.  $\mathcal{U}$  ist eine nicht-leere Menge.

2.  $\mathcal{J}$  ist 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.

- (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. Hier bezeichnet  $\mathbb{B}$  die Menge der Wahrheitswerte, d.h.  $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ .

- (c) Ist das Zeichen “=” eine Element der Menge der Prädikats-Zeichen  $\mathcal{P}$ , so soll gelten

$$=^{\mathcal{J}}(u, v) = \mathbf{true} \quad \text{g.d.w.} \quad u = v,$$

das Gleichheits-Zeichen wird also durch die identische Relation  $\text{id}_{\mathcal{U}}$  interpretiert.

**Beispiel:** Wir geben ein Beispiel für eine  $\Sigma_{\text{arith}}$ -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  $0, 1, +, -, *$  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  $\Sigma_G$  der Gruppen-Theorie sei definiert als

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

1.  $\mathcal{V} := \{x_i \mid i \in \mathbb{N}\}$
2.  $\mathcal{F} := \{1, *\}$
3.  $\mathcal{P} := \{=\}$
4.  $\text{arity} = \{\langle 1, 0 \rangle, \langle *, 2 \rangle, \langle =, 2 \rangle\}$

Dann können wir eine  $\Sigma_G$  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!

**Definition 28 (Variablen-Interpretation)** 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  $\Sigma$ -Struktur. Dann bezeichnen wir eine Abbildung

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

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

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

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

□

**Definition 29 (Semantik der Terme)** Ist  $\mathcal{S}$  eine  $\Sigma$ -Struktur und  $\mathcal{I}$  eine  $\mathcal{S}$ -Variablen-Interpretation, 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  $\Sigma$ -Terme der Form  $f(t_1, \dots, t_n)$  definieren wir

$$\mathcal{S}(\mathcal{I}, f(t_1, \dots, t_n)) := f^{\mathcal{J}}(\mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n)).$$

□

**Beispiel:** Mit der oben definierten  $\Sigma_{\text{arith}}$ -Struktur  $\mathcal{S}_{\text{arith}}$  definieren wir eine  $\mathcal{S}_{\text{arith}}$ -Variablen-Interpretation  $\mathcal{I}$  wie folgt:

1.  $\mathcal{I}(x) := 0$ .
2.  $\mathcal{I}(y) := 7$ .
3.  $\mathcal{I}(z) := 42$ .

Dann gilt

$$\mathcal{S}(\mathcal{I}, x + y) = 7.$$

**Definition 30 (Semantik der atomaren  $\Sigma$ -Formeln)** Ist  $\mathcal{S}$  eine  $\Sigma$ -Struktur und  $\mathcal{I}$  eine  $\mathcal{S}$ -Variablen-Interpretation, so definieren wir für jede atomare  $\Sigma$ -Formel  $p(t_1, \dots, t_n)$  den Wert  $\mathcal{S}(\mathcal{I}, p(t_1, \dots, t_n))$  wie folgt:

$$\mathcal{S}(\mathcal{I}, p(t_1, \dots, t_n)) := p^{\mathcal{J}}(\mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n)).$$

□

**Beispiel:** In Fortführung des obigen Beispiels gilt:

$$\mathcal{S}(\mathcal{I}, z \leq x + y) = \text{false}.$$

Um die Semantik beliebiger  $\Sigma$ -Formeln definieren zu können, nehmen wir an, dass wir die folgenden Funktionen zur Verfügung haben:

1.  $\ominus: \mathbb{B} \rightarrow \mathbb{B}$ ,
2.  $\odot: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ ,
3.  $\oslash: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ ,
4.  $\ominus: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ ,
5.  $\ominus: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ .

Die Semantik dieser Funktionen wird durch die Tabelle in Abbildung 3.1 auf Seite 47 gegeben.

**Definition 31 (Semantik der  $\Sigma$ -Formeln)** Ist  $\mathcal{S}$  eine  $\Sigma$ -Struktur und  $\mathcal{I}$  eine  $\mathcal{S}$ -Variablen-Interpretation, so definieren wir für jede  $\Sigma$ -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.  $\mathcal{S}(\mathcal{I}, \neg F) := \ominus(\mathcal{S}(\mathcal{I}, F))$ .
3.  $\mathcal{S}(\mathcal{I}, F \wedge G) := \oslash(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$ .
4.  $\mathcal{S}(\mathcal{I}, F \vee G) := \odot(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$ .
5.  $\mathcal{S}(\mathcal{I}, F \rightarrow G) := \ominus(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$ .
6.  $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) := \ominus(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$ .
7.  $\mathcal{S}(\mathcal{I}, \forall x: F) := \begin{cases} \text{true} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \text{true} \text{ für alle } c \in \mathcal{U} \text{ gilt;} \\ \text{false} & \end{cases}$

$$8. \mathcal{S}(\mathcal{I}, \exists x: F) := \begin{cases} \mathbf{true} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \mathbf{true} \text{ für ein } c \in \mathcal{U} \text{ gilt;} \\ \mathbf{false} & \end{cases} \quad \square$$

**Beispiel:** In Fortführung des obigen Beispiels gilt

$$\mathcal{S}(\mathcal{I}, \forall x: x * 0 < 1) = \mathbf{true}.$$

**Definition 32 (Allgemeingültig)** Ist  $F$  eine  $\Sigma$ -Formel, so dass für jede  $\Sigma$ -Struktur  $\mathcal{S}$  und für jede  $\mathcal{S}$ -Variablen-Interpretation  $\mathcal{I}$

$$\mathcal{S}(\mathcal{I}, F) = \mathbf{true}$$

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

$$\models F. \quad \square$$

Ist  $F$  eine Formel für die  $FV(F) = \emptyset$  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  $\mathcal{S}(F) = \mathbf{true}$ , 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  $\Sigma$  als gegeben an. Dadurch können wir in den folgenden Definitionen von Termen, Formeln, Strukturen, etc. sprechen und meinen damit  $\Sigma$ -Terme,  $\Sigma$ -Formeln und  $\Sigma$ -Strukturen.

**Definition 33 (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 34 (Erfüllbar)** Eine Menge  $M \subseteq \mathbb{F}_\Sigma$  ist genau dann *erfüllbar*, wenn es eine Struktur  $\mathcal{S}$  und eine Variablen-Interpretation  $\mathcal{I}$  gibt, so dass

$$\forall m \in M : \mathcal{S}(\mathcal{I}, m) = \mathbf{true}$$

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* 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 mal durchprobieren. Die Situation ist ähnlich der, wie wenn Sie für ein auf der Menge  $\mathbb{N}$  der natürlichen Zahlen definierte Funktion

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

überprüfen sollen, ob die Formel



$$\exists x: f(x) = 0$$

gültig ist, ob die Funktion also eine Nullstelle hat. Wenn Sie alle natürlichen Zahlen der Reihe nach ausprobieren, dann werden Sie eine vorhandene Nullstelle irgendwann auch finden. Wenn die Funktion aber keine Nullstelle hat, werden Sie das auf diese Art nie feststellen können.

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.

### 4.3 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 35** 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  $x \notin FV(f) \cup BV(f)$  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 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

$$(\forall x: f) \rightarrow (\exists x: f)$$

allgemeingültig ist:

$$\begin{aligned} & (\forall x: f) \rightarrow (\exists x: f) \\ \leftrightarrow & \neg(\forall x: f) \vee (\exists x: f) \\ \leftrightarrow & (\exists x: \neg f) \vee (\exists x: f) \\ \leftrightarrow & \exists x: \neg f \vee f \\ \leftrightarrow & \exists x: \top \\ \leftrightarrow & \top \end{aligned}$$

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. Trotzdem sagen diese Formeln in gewisser Weise das Gleiche aus. Die Formel  $f_1$  behauptet, dass für alle  $x$  ein  $y$  existiert, so dass  $p(x, y)$  gilt. In der Formel  $f_2$  wird eigentlich das Selbe ausgesagt, nur dass hier dieses  $y$ , was ja in Abhängigkeit von dem  $x$  existieren soll, durch die Funktion  $s$  berechnet wird. Wir verallgemeinern nun dieses Beispiel in der folgenden Definition. Wenn nichts anderes gesagt wird, dann arbeiten wir im Rest dieses Abschnitts nur noch mit geschlossenen Formeln, also mit solchen Formeln, die keine freien Variablen mehr enthaltenen.

**Definition 36 (Skolemisierung)** Es sei  $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$  eine Signatur. Ferner sei  $f$  eine geschlossene  $\Sigma$ -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  $\Sigma$  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  $\Sigma'$ -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 37 (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

$$f \approx_e g. \quad \square$$

**Satz 38** 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 *Literals*. (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 Allquantoren 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 Allquantoren vorkommen, können wir hier die Notation noch vereinfachen indem wir vereinbaren, dass alle Formeln implizit allquantifiziert sind, wir lassen also die Allquantoren 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. 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, mit 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 mir 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 pränexer 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 Allquantoren.

$$\begin{aligned}
& \exists v: \exists x: \forall y: \forall u: \left( p(x, y) \wedge \neg p(u, v) \right) \\
\approx_e & \exists x: \forall y: \forall u: \left( p(x, y) \wedge \neg p(u, s_1) \right) \\
\approx_e & \forall y: \forall u: \left( p(s_2, y) \wedge \neg p(u, s_1) \right)
\end{aligned}$$

Da jetzt nur noch Allquantoren 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 Ersetzen 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

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

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

$$\left\{ \neg \left( (\forall x: \exists y: p(x, y)) \rightarrow (\exists y: \forall x: p(x, y)) \right) \right\}$$

unerfüllbar und folglich gilt

$$\models \left( (\forall x: \exists y: p(x, y)) \rightarrow (\exists y: \forall x: p(x, y)) \right).$$

## 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 39 (Substitution)** Es sei eine Signatur

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

gegeben. Eine  $\Sigma$ -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  $\Sigma$ -Substitution, so schreiben wir

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

Außerdem definieren wir den *Domain* einer Substitution als

$$\text{dom}(\sigma) = \{x_1, \dots, x_n\}.$$

□

Substitutionen werden für uns dadurch interessant, dass wir sie auf Terme *anwenden* können. Ist  $t$  ein Term und  $\sigma$  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 40 (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  $\sigma$  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 Substitutionen auf Terme angewendet werden 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  $\sigma$  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.  $(p(x_2) \wedge q(d, h(x_3, x_1)))\tau = p(f(x_4)) \wedge q(d, h(x_3, h(c, c)))$ .

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

**Definition 41 (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  $\text{dom}(\sigma) \cap \text{dom}(\tau) = \emptyset$ . Dann definieren wir die *Komposition*  $\sigma\tau$  von  $\sigma$  und  $\tau$  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 42** Ist  $t$  ein Term und sind  $\sigma$  und  $\tau$  Substitutionen, so gilt

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

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

**Definition 43 (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 44 (Unifikator)** Eine Substitution  $\sigma$  *löst* eine syntaktische Gleichung  $s \doteq t$  genau dann, wenn  $s\sigma = t\sigma$  ist, wenn also durch die Anwendung von  $\sigma$  auf  $s$  und  $t$  tatsächlich identische Objekte entstehen. Ist  $E$  ein syntaktisches Gleichungs-System, so sagen wir, dass  $\sigma$  ein *Unifikator* von  $E$  ist wenn  $\sigma$  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  $\sigma$  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

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

und definieren die Substitution

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

Die Substitution  $\sigma$  löst die obige syntaktische Gleichung, denn es gilt

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

Als nächstes entwickeln wir ein Verfahren, mit dessen Hilfe wir von einer vorgegebenen Menge  $E$  von syntaktischen Gleichungen entscheiden können, ob es einen Unifikator  $\sigma$  für  $E$  gibt. Wir überlegen uns zunächst, in welchen Fällen wir eine syntaktische 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  $\sigma$  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  $g(t_1, \dots, t_n)\sigma$  verschiedene 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  $\tau$  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  $\langle \emptyset, \sigma \rangle$  erreicht wird. In diesem Fall ist  $\sigma$  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  $\sigma$  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.
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. 
$$\langle E \cup \{x \doteq x\}, \sigma \rangle \rightsquigarrow \langle E, \sigma \rangle.$$

Triviale syntaktische Gleichungen von Variablen können wir einfach weglassen.

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. Falls die Funktions-Zeichen  $f$  und  $g$  verschieden sind, so hat das Gleichungs-System

$$E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}$$

keine Lösung.

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 hat das Gleichungs-System  $E$  keine Lösung.
2. Das Paar  $\langle E, [] \rangle$  wird reduziert zu einem Paar  $\langle \emptyset, \sigma \rangle$ . Dann ist  $\sigma$  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  $\text{mgu}$  steht hier für "*most general unifier*".

**Beispiel:** Wir wenden das oben dargestellte Verfahren an, um die syntaktische Gleichung

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

zu lösen. Wir haben die folgenden Reduktions-Schritte:

$$\begin{aligned} & \langle \{p(x_1, q(x_4)) \doteq p(x_2, x_3)\}, [] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq x_2, q(x_4) \doteq x_3\}, [] \rangle \\ \rightsquigarrow & \langle \{q(x_4) \doteq x_3\}, [x_1 \mapsto x_2] \rangle \\ \rightsquigarrow & \langle \{x_3 \doteq q(x_4)\}, [x_1 \mapsto x_2] \rangle \\ \rightsquigarrow & \langle \{\}, [x_1 \mapsto x_2, x_3 \mapsto q(x_4)] \rangle \end{aligned}$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$[x_1 \mapsto x_2, x_3 \mapsto q(x_4)]$$

als Lösung der oben gegebenen syntaktischen Gleichung.

Wir wollen noch ein weiteres Beispiel angeben 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:

$$\begin{aligned}
& \left\langle \{p(h(x_1, c)) \doteq p(x_2), q(x_2, d) \doteq q(h(d, c), x_4)\}, \square \right\rangle \\
\rightsquigarrow & \left\langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c), d \doteq x_4\}, \square \right\rangle \\
\rightsquigarrow & \left\langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c), x_4 \doteq d\}, \square \right\rangle \\
\rightsquigarrow & \left\langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c)\}, [x_4 \mapsto d] \right\rangle \\
\rightsquigarrow & \left\langle \{p(h(x_1, c)) \doteq p(h(d, c))\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \right\rangle \\
\rightsquigarrow & \left\langle \{h(x_1, c) \doteq h(d, c)\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \right\rangle \\
\rightsquigarrow & \left\langle \{x_1 \doteq d, c \doteq c\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \right\rangle \\
\rightsquigarrow & \left\langle \{x_1 \doteq d\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \right\rangle \\
\rightsquigarrow & \left\langle \{\}, [x_4 \mapsto d, x_2 \mapsto h(d, c), x_1 \mapsto d] \right\rangle
\end{aligned}$$

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 45 (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,
4.  $\mu = mgu(p(s_1, \dots, s_n), p(t_1, \dots, t_n))$ .

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*.  $\square$

Die Resolutions-Regel ist die Verallgemeinerung der Schnitt-Regel auf die Prädikatenlogik. Unter Umständen kann es sein, dass wir die Variablen in Klauseln erst umbenennen müssen, bevor die Resolutions-Regel anwenden können. Betrachten wir dazu ein Beispiel. Die 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 in der zweiten Klausel jedoch umbenennen, erhalten wir zum Beispiel

$$\{\neg p(f(y))\}.$$

Nun können wir die Resolutions-Regel anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(y))$$

hat die Lösung  $[x \mapsto f(y)]$ . Dann sehen wir

$$\{p(x)\}, \quad \{\neg p(f(y))\} \quad \vdash \quad \{\}.$$

Die Resolutions-Regel alleine ist nicht ausreichend, 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. Mit der Resolutions-Regel alleine kann ein solcher Nachweis allerdings nicht gelingen, denn alle Klauseln aus  $M$  bestehen aus zwei Literalen. Wenn wir zwei Klauseln schneiden, die beide aus zwei Literalen bestehen, dann hat die resultierende Klausel auch wieder zwei Literale, es sei denn, dass diese beiden Literale identisch sind. Dies kann aber in dem vorliegenden Fall deswegen nicht passieren, weil die Variablen, die in den einzelnen Literalen auftreten, verschieden sind. Da die Klauseln die wir ableiten können, immer aus zwei Literalen bestehen, können wir nie die leere Klausel ableiten. Wir stellen daher jetzt die Faktorisierungs-Regel vor, mit der wir dann später zeigen werden, dass  $M$  widersprüchlich ist.

**Definition 46 (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.  $\mu = \text{mgu}(p(s_1, \dots, s_n), p(t_1, \dots, t_n))$ .

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

$$\mu = \text{mgu}(p(f(x), y), p(u, g(v))) = [y \mapsto g(v), u \mapsto f(x)].$$

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

$$\mu = \text{mgu}(\neg p(f(x), y), \neg p(u, g(v))) = [y \mapsto g(v), u \mapsto f(x)].$$

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 47 (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 48 (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 49 (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

$$M \vdash \{\}.$$

$\square$

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:  $\models f$  g.d.w.  $\{\neg f\} \models \perp$  g.d.w.  $\{k_1, \dots, k_n\} \models \perp$ .

3. Nach dem Korrektheits-Satz und dem Satz über die Widerlegungs-Vollständigkeit gilt

$$\{k_1, \dots, k_n\} \models \perp \text{ g.d.w. } \{k_1, \dots, k_n\} \vdash \perp.$$

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 Beispiels. 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.  $\mathcal{F} = \emptyset$ .

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. Wir können uns die Arbeit erleichtern, wenn wir die Formeln  $f_1$ ,  $f_2$ ,  $f_3$  und  $\neg f_4$  getrennt betrachten.

1. Die Formel  $f_1$  kann wie folgt umgeformt werden:

$$\begin{aligned} f_1 &= \forall x : (\forall y : \text{kind}(y, x) \rightarrow \text{fliegt}(y)) \rightarrow \text{glücklich}(x) \\ &\leftrightarrow \forall x : \neg(\forall y : \neg \text{kind}(y, x) \vee \text{fliegt}(y)) \vee \text{glücklich}(x) \\ &\leftrightarrow \forall x : (\exists y : \text{kind}(y, x) \wedge \neg \text{fliegt}(y)) \vee \text{glücklich}(x) \\ &\leftrightarrow \forall x : \exists y : (\text{kind}(y, x) \wedge \neg \text{fliegt}(y)) \vee \text{glücklich}(x) \\ &\approx_e \forall x : (\text{kind}(s(x), x) \wedge \neg \text{fliegt}(s(x))) \vee \text{glücklich}(x) \end{aligned}$$

Im letzten Schritt haben wir dabei die Skolem-Funktion  $s$  mit  $\text{arity}(s) = 1$  eingeführt. 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:

$$\begin{aligned} f_3 &= \forall x : \text{rot}(x) \rightarrow \forall y : \text{kind}(y, x) \rightarrow \text{rot}(y) \\ &\leftrightarrow \forall x : \neg \text{rot}(x) \vee \forall y : \neg \text{kind}(y, x) \vee \text{rot}(y) \\ &\leftrightarrow \forall x : \forall y : \neg \text{rot}(x) \vee \neg \text{kind}(y, x) \vee \text{rot}(y) \end{aligned}$$

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:

$$\begin{aligned} \neg f_4 &= \neg \forall x : \text{rot}(x) \rightarrow \text{glücklich}(x) \\ &\leftrightarrow \neg \forall x : \neg \text{rot}(x) \vee \text{glücklich}(x) \\ &\leftrightarrow \exists x : \text{rot}(x) \wedge \neg \text{glücklich}(x) \\ &\approx_e \text{rot}(d) \wedge \neg \text{glücklich}(d) \end{aligned}$$

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.  $\{ \text{rot}(d) \}, \{ \neg \text{rot}(x) \vee \neg \text{kind}(y, x) \vee \text{rot}(y) \} \vdash \{ \neg \text{kind}(y, d) \vee \text{rot}(y) \}$
2.  $\{ \neg \text{kind}(y, d) \vee \text{rot}(y) \}, \{ \text{kind}(s(x), x) \vee \text{glücklich}(x) \} \vdash \{ \text{glücklich}(d) \vee \text{rot}(s(d)) \}$
3.  $\{ \text{glücklich}(d) \vee \text{rot}(s(d)) \}, \{ \neg \text{glücklich}(d) \} \vdash \{ \text{rot}(s(d)) \}$
4.  $\{ \text{rot}(s(d)) \}, \{ \neg \text{rot}(x) \vee \text{fliegt}(x) \} \vdash \{ \text{fliegt}(s(d)) \}$
5.  $\{ \text{fliegt}(s(d)) \}, \{ \neg \text{fliegt}(s(x)) \vee \text{glücklich}(x) \} \vdash \{ \text{glücklich}(d) \}$
6.  $\{ \text{glücklich}(d) \}, \{ \neg \text{glücklich}(d) \} \vdash \{ \}.$

Damit ist  $M \vdash \perp$  nachgewiesen und wir haben gezeigt, dass alle roten Drachen glücklich sind.

# Kapitel 5

## Prolog

Im diesem Kapitel wollen wir uns mit dem logischen Programmieren und der Sprache *Prolog* beschäftigen. 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.

Wir geben ein einführendes Beispiel. Abbildung 5.1 auf Seite 102 zeigt ein *Prolog*-Programm, das aus einer Ansammlung von *Fakten* und *Regeln* besteht:

1. Ein *Fakt* ist eine einfache Aussage. Die Syntax ist

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

Dabei ist  $p$  ein Prädikats-Zeichen und  $t_1, \dots, t_n$  sind Terme. Ist die Menge der Variablen, die in den Termen  $t_1, \dots, t_n$  vorkommen, durch  $\{x_1, \dots, x_m\}$  gegeben, so wird der obige Fakt als die logische Formel

$$\forall x_1, \dots, x_m: p(t_1, \dots, t_n)$$

interpretiert. Das Programm in Abbildung 5.1 enthält in den Zeilen 1 – 5 Fakten. Umgangssprachlich können wir diese wie folgt lesen:

- (a) Asterix ist ein Gallier.
- (b) Obelix ist ein Gallier.
- (c) Cäsar ist ein Kaiser.
- (d) Cäsar ist ein Römer.

2. Eine *Regel* ist eine bedingte Aussage. Die Syntax ist

$$A :- B_1, \dots, B_n.$$

Dabei sind  $A$  und  $B_1, \dots, B_n$  atomare Formeln, haben also die Gestalt

$$q(s_1, \dots, s_k),$$

wobei  $q$  ein Prädikats-Zeichen ist und  $s_1, \dots, s_k$  Terme sind. Ist die Menge der Variablen, die in den atomaren Formeln  $A, B_1, \dots, B_n$  auftreten, durch  $\{x_1, \dots, x_m\}$  gegeben, so wird die obige Regel als die logische Formel

$$\forall x_1, \dots, x_m: B_1 \wedge \dots \wedge B_n \rightarrow A$$

interpretiert. Das Programm aus Abbildung 5.1 enthält in den Zeilen 7–12 Regeln. Umgangssprachlich können wir diese Regeln wie folgt lesen:

- (a) Alle Gallier sind stark.
- (b) Wer stark ist, ist mächtig.
- (c) Wer Kaiser und außerdem noch Römer ist, der ist mächtig.
- (d) 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 ein, 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.
2. Strings, die mit einem kleinen Buchstaben beginnen und nur aus Buchstaben, Ziffern und dem Unterstrich “\_” bestehen, bezeichnen Prädikats- und Funktions-Zeichen.
3. Die Strings  

$$“+”, “-”, “*”, “/”, “.”$$
bezeichnen Funktions-Zeichen.
4. Die Strings  

$$“<”, “>”, “=”, “=<”, “>=”$$
bezeichnen Prädikats-Zeichen.
5. Zahlen bezeichnen 0-stellige Funktions-Zeichen.

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*-System<sup>1</sup> in einer Shell mit dem Kommando:

`pl`

Wenn das *Prolog*-System installiert ist, begrüßt uns das System wie folgt:

```
Welcome to SWI-Prolog (Multi-threaded, Version 5.2.13)
Copyright (c) 1990-2003 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 und erhalten als Antwort:

```
?- maechtig(X), spinnt(X).

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

```
?- maechtig(X), spinnt(X).

X = caesar ;

No
?-
```

Außer Caesar gibt es also niemanden, der mächtig ist und spinnt. Wenn wir das *Prolog*-System wieder verlassen wollen, dann geben wir den Befehl “halt.” ein.

---

<sup>1</sup>Sie finden dieses Prolog-System im Netz unter [www.swi-prolog.org](http://www.swi-prolog.org).



## 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. Dabei repräsentiert **true** die Formel  $\top$ . 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  $G = Q_1, \dots, Q_n$
- (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 Instanz $G\sigma$ von $G$ so dass $G\sigma$ aus $P$ folgt.

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 setze wir an dieser Stelle einen Auswahl-Punkt (*Choice-Point*), um später hier eine anderer Regel wählen zu können, falls dies notwendig werden sollte.

#### 3. Wir setzen $G := G\mu$ , wobei $\mu$ 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  $G$  zurück.
  - (b) Sonst beantworten wir rekursiv die Anfrage  $B_1\mu, \dots, B_m\mu, Q_2\mu, \dots, Q_n\mu.$
5. 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.

Um den Algorithmus besser zu verstehen, beobachten wir die Abarbeitung der Anfrage

```
maechtig(X), spinnt(X).
```

noch einmal im Debugger des *Prolog*-Systems. Wir geben dazu nach dem Laden des Programms “gallier.pl” das Kommando `guitracer` ein. Wir erhalten die Antwort:

```
?- guitracer.
```

```
% 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), roemer(X).
```


Als Ergebnis wird ein Fenster geöffnet, das Sie in Abbildung 5.2 auf Seite 106 sehen. Unter dem Menü sehen Sie hier eine Werkzeugleiste. Die einzelnen Symbole haben dabei die folgende Bedeutung:

1. Die Schaltfläche  dient dazu, einzelne Unifikationen anzuzeigen. Mit dieser Schaltfläche können wir die meisten Details der Abarbeitung beobachten.

Alternativ hat die Taste “i” die selbe Funktion.

2. Die Schaltfläche  dient dazu, einen einzelnen Schritt bei der Abarbeitung einer Anfrage durchzuführen.


Alternativ hat die Leertaste die selbe Funktion.

3. Die Schaltfläche  dient dazu, die nächste atomare Anfrage in einem Schritt durchzuführen. Drücken wir diese Schaltfläche unmittelbar, nachdem wir das Ziel

```
maechtig(X), roemer(X).
```

einggegeben haben, so würde der Debugger die Anfrage `maechtig(X)` in einem Schritt beantworten. Dabei würde dann die Variable `X` an die Konstante `asterix` gebunden.


Alternativ hat die Taste “s” (*skip*) die selbe Funktion.

4. Die Schaltfläche  dient dazu, die Prozedur, in deren Abarbeitung wir uns befinden, ohne Unterbrechung zu Ende abarbeiten zu lassen. Versuchen wir beispielsweise die atomare Anfrage “`maechtig(X)`” mit der Regel




```
maechtig(X) :- kaiser(X), roemer(X).
```

abzuarbeiten und müssen nun die Anfrage “`kaiser(X), roemer(X).`” beantworten, so würden wir durch Betätigung dieser Schaltfläche sofort die Antwort “`X = caesar`” für diese Anfrage erhalten.

Alternative hat die Taste “f” (*finish*) die selbe Funktion.

5. Die Schaltfläche  dient dazu, eine bereits beantwortete Anfrage noch einmal zu beantworten. Dies kann sinnvoll sein, wenn man beim Tracen einer Anfrage nicht aufgepaßt hat und noch einmal sehen möchte, wie das *Prolog*-System zu seiner Antwort gekommen ist.

Alternative hat die Taste “r” (*retry*) die selbe Funktion.

6. Die Schaltfläche  beantwortet die gestellte Anfrage ohne weitere Unterbrechung.  
Alternative hat die Taste “n” (*nodebug*) die selbe Funktion.  
Von den weiteren Schaltflächen sind fürs erste nur noch zwei interessant.
7. Die Schaltfläche  läßt das Programm bis zum nächsten Haltepunkt laufen.  
Alternative hat die Taste “l” (*continue*) die selbe Funktion.
8. Die Schaltfläche  setzt einen Haltepunkt. Dazu muß vorher der Cursor an die Stelle gebracht werden, an der ein Haltepunkt gesetzt werden soll.  
Alternative hat die Taste “!” die selbe Funktion.

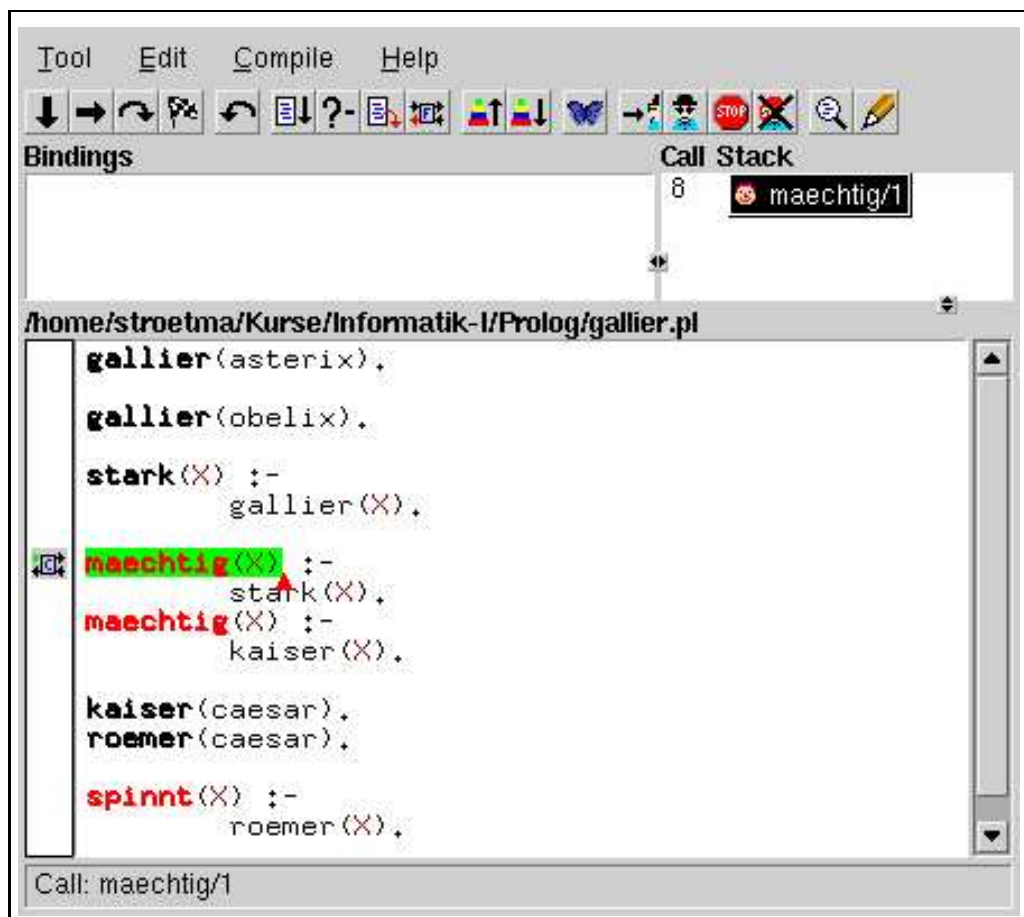


Abbildung 5.2: Der Debugger des SWI-Prolog-Systems.

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.

## 5.2 Ein komplexeres Beispiel

Das obige Beispiel war bewußt einfach gehalten um die Sprache *Prolog* einzuführen. Um die Mächtigkeit des Backtrackings zu demonstrieren, präsentieren wir jetzt ein komplexeres Beispiel. Es handelt sich um das folgende Rätsel:

1. Drei Freunde belegen den ersten, zweiten und dritten Platz bei einem Programmier-Wettbewerb.
2. Jeder der drei hat genau einen Vornamen, genau ein Auto und hat sein Programm in genau einer Programmier-Sprache geschrieben.
3. Michael programmiert in *Setl* und war besser als der Audi-Fahrer.

4. Julia, die einen Ford Mustang fährt, war besser als der Java-Programmierer.
5. Das Prolog-Programm war am besten.
6. Wer fährt Toyota?
7. In welcher Sprache programmiert Thomas?

Um dieses Rätsel zu lösen, überlegen wir uns zunächst, wie wir die einzelnen Daten repräsentieren können, die in dem Rätsel eine Rolle spielen. Zunächst ist dort von Personen die Rede. Jede dieser Personen hat genau einen Vornamen, ein Auto und eine Programmier-Sprache. Wir repräsentieren Personen daher durch Terme der Form

`person(Name, Car, Language).`

Dabei bezeichnen *Name*, *Car* und *Language* Konstanten, die aus den entsprechenden Mengen gewählt werden:

$Name \in \{\text{Julia, Thomas, Michael}\}, \quad Car \in \{\text{Ford, Toyota, Audi}\},$   
 $Language \in \{\text{Java, Prolog, Setl}\}.$

Wenn wir Personen so repräsentieren, können wir sofort Prädikate angeben, die den Vornamen, die Auto-Marke und die Programmier-Sprache aus einem solchen Term extrahieren.

1. Das Prädikat `first_name/2` extrahiert den Vornamen:

`first_name(person(Name, Car, Language), Name).`

2. Das Prädikat `car/2` extrahiert die Auto-Marke:

`car(person(Name, Car, Language), Car).`

3. Das Prädikat `language/2` extrahiert die Programmier-Sprache:

`language(person(Name, Car, Language), Language).`

Um zu verstehen wie diese Prädikate arbeiten, zeigen wir, wie die Anfrage

`car( person(hans, seat, setl), X ).`

von dem *Prolog*-System beantwortet wird. Die einzige Regel, die zur Beantwortung dieser Anfrage herangezogen werden kann, ist die Regel

`car(person(Name, Car, Language), Car) :- true.`

Um diese Regel anwenden zu können, ist die syntaktische Gleichung

`car( person(hans, seat, setl), X )  $\doteq$  car( person(Name, Car, Language), Car )`

zu lösen. Bei der Unifikation findet sich die Lösung

$\mu = [Name \mapsto \text{hans}, Car \mapsto \text{seat}, Language \mapsto \text{setl}, X \mapsto \text{seat}].$

Insbesondere wird also die Variable *X* bei dieser Anfrage an die Konstante *seat* gebunden.

Wie können wir nun die Reihenfolge repräsentieren, in der die drei Personen bei dem Wettbewerb abgeschnitten haben? Wir wählen ein dreistelliges Funktions-Zeichen *sequence* und repräsentieren die Reihenfolge durch den Term

`sequence(First, Second, Third).`

Dabei stehen *First*, *Second* und *Third* für Terme, die von dem Funktions-Zeichen *person/3* erzeugt worden sind und die Personen bezeichnen. Die Reihenfolge kann dann durch das Prädikat

`did_better(Better, Worse, Sequence)`

berechnet werden, dessen Implementierung in den Zeilen 38 – 40 der Abbildung 5.3 auf Seite 111 gezeigt ist. Wir können nun daran gehen, das Rätsel zu lösen. Abbildung 5.3 zeigt die Implementierung. Zeile 1 – 30 enthält die Implementierung einer Regel für das Prädikat *answer/2*, dass die Lösung des Rätsels berechnet. In dieser Regel haben wir das Rätsel als prädikatenlogische Formel codiert. Wir übersetzen diese Regel jetzt zurück in die Umgangssprache und zeigen dadurch, dass das Prädikat *answer/2* das Rätsel korrekt beschreibt. Die Numerierung in der folgenden Aufzählung stimmt jeweils mit der entsprechenden Zeilen-Nummer im Programm überein:

2. Falls **Sequence** eine Reihenfolge von drei Personen beschreibt und
4. **Michael** eine Person aus dieser Reihenfolge ist und
5. der Name der durch **Michael** bezeichneten Person den Wert **michael** hat und
6. **Michael** in SETL programmiert und
8. **Audi** eine Person aus der Reihenfolge **Sequence** ist und
9. **Michael** beim Wettbewerb besser abgeschnitten hat als die durch **Audi** bezeichnete Person
10. die durch **Audi** bezeichnete Person einen Audi fährt und
- ⋮
24. **Toyota** eine Person aus der Reihenfolge **Sequence** ist und
25. die durch **Toyota** bezeichnete Person einen Toyota fährt und
26. **NameToyota** den Vornamen der durch **Toyota** bezeichneten Person angibt und
28. **Thomas** eine Person aus der Reihenfolge **Sequence** ist und
25. die durch **Thomas** bezeichnete Person den Vornamen Thomas hat und
26. **LanguageThomas** die Sprache ist, in der die durch **Thomas** bezeichnete Person programmiert, dann gilt:
  1. **NameToyota** ist der Namen des Toyota-Fahrers und **LanguageThomas** ist die Sprache, in der Thomas programmiert.

Wenn wir die ursprüngliche Aufgabe mit der Implementierung in *Prolog* vergleichen, dann stellen wir fest, dass die in dem Rätsel gemachten Angaben eins-zu-eins in *Prolog* übersetzt werden konnten. Diese Übersetzung beschreibt nur das Rätsel und gibt keinen Hinweis, wie dieses Rätsel zu lösen ist. Für die Lösung ist dann die dem *Prolog*-System zu Grunde liegende Inferenz-Maschine zuständig.

## 5.3 Listen

In Prolog wird viel mit Listen gearbeitet. Listen werden in Prolog mit dem 2-stelligen Funktions-Zeichen “.” konstruiert. Ein Term der Form

.(*s*,*t*)

steht also für eine Liste, die als erstes Element “*s*” enthält. “*t*” bezeichnet den Rest der Liste. Das Funktions-Zeichen “[]” steht für die leere Liste. Eine Liste, die aus den drei Elementen “**a**”, “**b**” und “**c**” besteht, kann also wie folgt dargestellt werden:

.(**a**, .(**b**, .(**c**, [])))

Da dies relativ schwer zu lesen ist, darf diese Liste auch als

[**a**,**b**,**c**]

geschrieben werden. Zusätzlich kann der Term “.(*s*,*t*)” in der Form

[ *s* | *t* ]

geschrieben werden. Um diese Kurzschreibweise zu erläutern, geben wir ein kurzes Prolog-Programm an, dass zwei Listen aneinander hängen kann. Das Programm implementiert das dreistellige Prädikat **concat**<sup>2</sup>. Die Intention ist, dass **concat**(*l*<sub>1</sub>,*l*<sub>2</sub>,*l*<sub>3</sub>) für drei Listen *l*<sub>1</sub>, *l*<sub>2</sub> und *l*<sub>3</sub> genau dann wahr sein soll, wenn die Liste *l*<sub>3</sub> dadurch entsteht, dass die Liste *l*<sub>2</sub> hinten an die Liste *l*<sub>1</sub> angehängt wird. Das Programm besteht aus den folgenden beiden Klauseln:

---

<sup>2</sup>In dem *SWI-Prolog*-System gibt es das vordefinierte Prädikat **append/3**, das genau das selbe leistet wie unsere Implementierung von **concat/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 5.3: Wer fährt Toyota?



```
concat( [], L, L ).
concat( [ X | L1 ], L2, [ X | L3 ] ) :- concat( L1, L2, L3 ).
```

Wir können diese beiden Klauseln folgendermaßen in die Umgangssprache übersetzen:

1. Hängen wir eine Liste  $L$  an die leere Liste an, so ist das Ergebnis  $L$ .
2. Um an eine Liste  $[ X \mid L1 ]$ , 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 gesuchte Ergebnis ist dann die Liste  $[ X \mid L3 ]$ .

Wir testen unser Programm und nehmen dazu an, dass die beiden Programm-Klauseln in der Datei “concat.pl” abgespeichert sind und dass wir diese Datei mit dem Befehl “consult(concat).” geladen haben. Dann stellen wir die Anfrage

```
?- concat( [ 1, 2, 3 ], [ a, b, c ], L ).
```

Wir erhalten die Antwort:

```
L = [1, 2, 3, a, b, c]
```

Die obige Interpretation des gegebenen Prolog-Programms ist *funktional*, dass heißt wir fassen die ersten beiden Argumente des Prädikats `concat` als *Eingaben* auf und interpretieren das letzte *Argument* als Ausgabe. Diese Interpretation ist aber keineswegs die einzig mögliche Interpretation. Um das zu sehen, geben wir als Ziel

```
concat(L1, L2, [1,2,3]).
```

ein und drücken, nachdem das System uns die erste Antwort gegeben hat, nicht die Taste *Return* sondern die Taste “;”. Wir erhalten:

```
?- concat(L1, L2, [1, 2, 3]).
```

```
L1 = []
```

```
L2 = [1, 2, 3] ;
```

```
L1 = [1]
```

```
L2 = [2, 3] ;
```

```
L1 = [1, 2]
```

```
L2 = [3] ;
```

```
L1 = [1, 2, 3]
```

```
L2 = [] ;
```

```
No
```

In diesem Fall hat das Prolog-System durch Backtracking alle Möglichkeiten bestimmt die es gibt, um die Liste “[1, 2, 3]” in zwei Teile zu zerlegen.

### 5.3.1 Sortieren durch Einfügen

Wir entwickeln nun einen einfachen Algorithmus zum Sortieren von Listen von Zahlen. Das Programm besteht 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 5.4 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 5.4: 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 die 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 Schwanz der Liste, in die wir *X* einfügen wollen. Es 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.

### 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. 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  $n * \log_2(n)$ . Wenn es sich bei der zu sortierenden Liste beispielsweise um ein Telefonbuch mit 1 Millionen Einträgen handelt, dann ist der relative Unterschied des Rechenzeit-Verbrauchs durch den Faktor  $\approx 50\,000$  gegeben.

Wir werden den Algorithmus zunächst durch bedingte Gleichungen beschreiben und anschließend die Umsetzung dieser Gleichungen in *Prolog* angeben. Der Algorithmus wird in der Literatur als *Sortieren durch Mischen* bezeichnet (engl. *merge sort*) und besteht aus drei Phasen:

1. In der ersten Phase wird die zu sortierende Liste in zwei etwa gleich große Teile aufgeteilt.
2. In der zweiten Phase werden diese Teile rekursiv sortiert.
3. In der dritten Phase werden die sortierten Teillisten so zusammen gefügt (gemischt), dass die resultierende Liste ebenfalls sortiert ist.

Wir beginnen mit dem Aufteilen einer Liste in zwei Teile. Bei der Aufteilung orientieren wir uns an den Indizes der Elemente. Zur Illustration zunächst ein Beispiel: Wir teilen die Liste

$[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8]$  auf in  $[a_1, a_3, a_5, a_7]$  und  $[a_2, a_4, a_6, a_8]$ .

Elemente, deren Index gerade ist, kommen in die eine Teilliste und die Elemente mit ungeradem Index kommen in die andere Teilliste. Als Namen für die Funktionen, die diese Teillisten berechnen, wählen wir `even` und `odd`:

`odd : List(Number) → List(Number)`,

$\text{even} : \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number})$ .

Diese Funktionen können durch die folgenden Gleichungen spezifiziert werden:

1.  $\text{odd}([]) = []$ .
2.  $\text{odd}([h|t]) = [h|\text{even}(t)]$ .
3.  $\text{even}([]) = []$ .
4.  $\text{even}([h|t]) = \text{odd}(t)$ .

Als nächstes entwickeln wir eine Funktion

$\text{merge} : \text{List}(\text{Number}) \times \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number})$

die zwei Listen so mischt, dass die resultierende Liste aufsteigend sortiert ist. Durch rekursive Gleichungen kann diese Funktion wie folgt spezifiziert werden:

1.  $\text{merge}([], l) = l$ .
2.  $\text{merge}(l, []) = l$ .
3.  $x \leq y \rightarrow \text{merge}([x|s], [y|t]) = [x|\text{merge}(s, [y|t])]$ .  
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.  $x > y \rightarrow \text{merge}([x|s], [y|t]) = [y|\text{merge}([x|s], t)]$ .

Damit können wir jetzt die Funktion

$\text{sort} : \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number})$ ,

die eine Liste von Zahlen sortiert, durch bedingte Gleichungen spezifizieren.

1.  $\text{sort}([]) = []$ .
2.  $\text{sort}([x]) = [x]$ .
3.  $\text{sort}([x, y|t]) = \text{merge}(\text{sort}(\text{odd}([x, y|t])), \text{sort}(\text{even}([x, y|t])))$ .

Die oben angegebenen Gleichungen lassen sich nun unmittelbar in ein *Prolog*-Programm umsetzen. Abbildung 5.5 auf Seite 116 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

Symbolische Rechnungen sind in *Prolog* deswegen einfach, weil die zu manipulierenden Objekte in der Regel unmittelbar als Prolog-Terme dargestellt werden können. Wir zeigen 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 die arithmetischen Ausdrücke können unmittelbar durch Terme dargestellt werden. Wir stellen zunächst bedingte Gleichungen für eine Funktion

$\text{diff} : \text{Expr} \times \text{Var} \rightarrow \text{Expr}$

auf, die einen gegebenen arithmetischen Ausdruck nach einer gegebenen Variable ableitet.

---

```

1  % odd( +List(Number), -List(Number) ).
2
3  odd( [], [] ).
4
5  odd( [ X | Xs ], [ X | L ] ) :-
6      even( Xs, L ).
7
8  % even( +List(Number), -List(Number) ).
9
10 even( [], [] ).
11
12 even( [ _X | Xs ], L ) :-
13     odd( Xs, L ).
14
15 % merge( +List(Number), +List(Number), -List(Number) ).
16 %   Takes 2 sorted lists and returns a list which
17 %   (a) contains the elements of both input lists, and
18 %   (b) is sorted.
19
20 mix( [], Xs, Xs ).
21
22 mix( Xs, [], Xs ).
23
24 mix( [ X | Xs ], [ Y | Ys ], [ X | Rest ] ) :-
25     X <= Y,
26     mix( Xs, [ Y | Ys ], Rest ).
27
28 mix( [ X | Xs ], [ Y | Ys ], [ Y | Rest ] ) :-
29     X > Y,
30     mix( [ X | Xs ], Ys, Rest ).
31
32 % merge_sort( +List(Number), -List(Number) ).
33 %   Takes one list as input and sorts it.
34
35 merge_sort( [], [] ).
36
37 merge_sort( [ X ], [ X] ).
38
39 merge_sort( [ X, Y | Rest ], Sorted ) :-
40     odd( [ X, Y | Rest ], Odd ),
41     even( [ X, Y | Rest ], Even ),
42     merge_sort( Odd, Odd_Sorted ),
43     merge_sort( Even, Even_Sorted ),
44     mix( Odd_Sorted, Even_Sorted, Sorted ).

```

---

Abbildung 5.5: Sortieren durch Mischen.

1. Die Ableitung einer Variablen nach sich selbst ist durch die Formel

$$\frac{dx}{dx} = 1$$

gegeben. Also haben wir

$$\text{diff}(x, x) = 1.$$

2. Die Ableitung einer Konstante  $c$  ergibt 0:

$$\frac{dc}{dx} = 0.$$

Prinzipiell gibt es in einem arithmetischen Ausdruck zwei Arten von Konstanten:

- (a) Jede Zahl ist eine Konstante.
- (b) Strings, die von der gegebenen Variable verschieden sind, sind ebenfalls Konstanten.

Um Überprüfen zu können, ob ein Term eine Konstante darstellt, brauchen wir einige meta-logische Prädikate. Das *SWI-Prolog*-System stellt für unsere Zwecke die folgenden Prädikate zur Verfügung:

- (a) Das Prädikat `number(X)` überprüft ob das Argument  $X$  eine Zahl ist.
- (b) Das Prädikat `atom(X)` überprüft ob das Argument  $X$  ein String ist. Im *Prolog*-Kontext heißen solche Strings auch *Atome*.
- (c) Mit dem Prädikat `\==` können wir überprüfen, ob zwei Terme von einander verschieden sind.

Damit können wir nun zwei bedingte Gleichungen aufstellen, die die Differenzierung von Konstanten beschreiben:

- (a) `number(n) → diff(n, x) = 0.`
- (b) `atom(c) ∧ c \== x → diff(c, x) = 0.`

3. Die Ableitung eines Ausdrucks mit negativen Vorzeichen wird 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

$$\mathbf{diff}(f/g, x) = (\mathbf{diff}(f, x) * g - f * \mathbf{diff}(g, x)) / (g * g).$$

8. Für eine Zahl  $n \in \mathbb{R}$  finden wir die Ableitung der  $n$ -ten Potenz eines Ausdrucks mit Hilfe der Ketten-Regel:

$$\frac{d}{dx} f^n = n * \frac{df}{dx} * f^{n-1}.$$

Das führt auf die Gleichung

$$\mathbf{number}(n) \rightarrow \mathbf{diff}(f^{**n}, x) = n * \mathbf{diff}(f, x) * f^{**(n-1)}.$$

9. Auch 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

$$\mathbf{diff}(\exp(f), x) = \mathbf{diff}(f, x) * \exp(f).$$

10. Die Ableitung des natürlichen Logarithmus finden wir als

$$\frac{d}{dx} \ln(f) = \frac{\frac{df}{dx}}{f}.$$

Das führt auf die Gleichung

$$\mathbf{diff}(\exp(f), x) = \mathbf{diff}(f, x) / f.$$

Die eben ermittelten Gleichungen lassen sich nun unmittelbar in ein *Prolog*-Programm umsetzen. Abbildung 5.6 auf Seite 119 zeigt dieses Programm. Hier gibt es eine Stelle, die noch erklärt werden muß. In Zeile 33 benutzen wir das extralogische Prädikat `is/2` mit dessen Hilfe arithmetische Ausdrücke ausgewertet werden können. Für dieses Prädikat kann die Infix-Notation verwendet werden. Ist *Expr* ein arithmetischer Ausdruck, so berechnet der Aufruf

`X is Expr`

den Wert des Ausdrucks *Expr* und instantiiert die Variable *X* mit dem Wert dieses Ausdrucks. Beispielsweise führt der Aufruf

`X is 2 + 3 * 4`

dazu, dass die Variable *X* an die Zahl 14 gebunden wird.

---

```

1  % diff( +Expr, +Atom, -Expr ).
2
3  diff( X, X, 1 ).
4
5  diff( C, _X, 0 ) :-
6      number(C).
7
8  diff( Y, X, 0 ) :-
9      atom(Y),
10     X \== Y.
11
12  diff( - F, X, - DF ) :-
13     diff( F, X, DF ).
14
15  diff( F + G, X, DF + DG ) :-
16     diff( F, X, DF ),
17     diff( G, X, DG ).
18
19  diff( F - G, X, DF - DG ) :-
20     diff( F, X, DF ),
21     diff( G, X, DG ).
22
23  diff( F * G, X, DF * G + F * DG ) :-
24     diff( F, X, DF ),
25     diff( G, X, DG ).
26
27  diff( F / G, X, (DF * G - F * DG) / (G * G) ) :-
28     diff( F, X, DF ),
29     diff( G, X, DG ).
30
31  diff( F ** N, X, N * DF * F ** N1 ) :-
32     number(N),
33     N1 is N - 1,
34     diff( F, X, DF ).
35
36  diff( exp(F), X, DF * exp(F) ) :-
37     diff( F, X, DF ).
38
39  diff( ln(F), X, DF / F ) :-
40     diff( F, X, DF ).

```

---

Abbildung 5.6: Ein Programm zum symbolischen Differenzieren



## 5.4 Negation

In *Prolog* gibt es den Operator “**not**”, der alternativ auch als “**\+**” geschrieben werden kann. 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. Zur Implementierung der Funktion **difference** benutzen wir die Hilfs-Funktion

$$\text{remove} : \text{Number} \times \text{List}(\text{Number}) \rightarrow \text{List}(\text{Number}).$$

Der Aufruf

$$\text{remove}(n, L)$$

entfernt das Element  $n$  aus der Liste  $L$ . Die beiden Funktionen **difference** und **remove** werden durch die folgenden Gleichungen spezifiziert.

1.  $\text{difference}(l, []) = l.$
2.  $\text{member}(x, l) \rightarrow \text{difference}(l, [x|r]) = \text{difference}(\text{remove}(x, l), r).$
3.  $\neg \text{member}(x, l) \rightarrow \text{difference}(l, [x|r]) = \text{difference}(l, r).$

Die Gleichungen zur Spezifikation von **remove** lauten:

1.  $\text{remove}(x, []) = [].$
2.  $x = y \rightarrow \text{remove}(x, [y|r]) = \text{remove}(x, r).$
3.  $x \neq y \rightarrow \text{remove}(x, [y|r]) = [y|\text{remove}(x, r)].$

Die Umsetzung dieser Gleichungen in ein *Prolog*-Programm führt auf die in Abbildung 5.7 gezeigte Implementierung. Hier haben wir den **not**-Operator an zwei Stellen benutzt: in Zeile 11 um sicherzustellen, dass **X** keine Element der Liste **L** ist und in Zeile 22 um zu fordern, dass **X** von **Y** verschieden ist. Die Beantwortung einer Anfrage

$$\backslash+ A$$

geschieht im *Prolog*-System folgendermaßen:

1. Das *Prolog*-System versucht, 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 instantiiert.
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 instantiiert werden. Eine negierte Anfrage

$$\backslash+ A$$

funktioniert daher nur dann wie erwartet, wenn die Anfrage **A** keine Variablen mehr enthält. Zur Illustration betrachten wir das Programm in Abbildung 5.8. Versuchen wir mit diesem Programm die Anfrage

$$\text{smart1}(X)$$

zu beantworten, so wird diese Anfrage reduziert zu der Anfrage

$$\backslash+ \text{roemer}(X), \text{gallier}(X).$$

---

```

1  % difference( +List(Number), +List(Number), -List(Number) ).
2  difference( L, [], L ).
3
4  difference( L, [ X | Xs ], D ) :-
5      member( X, L ),
6      remove( X, L, R ),
7      difference( R, Xs, D ).
8
9  difference( L, [ X | Xs ], D ) :-
10     \+ member( X, L ),
11     difference( L, Xs, D ).
12
13 % remove( +Number, +List(Number), -List(Number) ).
14 remove( _X, [], [] ).
15
16 remove( X, [ X | Xs ], R ) :-
17     remove( X, Xs, R ).
18
19 remove( X, [ Y | Ys ], [ Y | R ] ) :-
20     X \= Y,                                     % same as \+ X = Y,
21     remove( X, Ys, R ).

```

---

Abbildung 5.7: Berechnung der Differenz zweier Mengen

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 5.8: 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 5.8 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 sollten in *Prolog*-Programmen Formeln nur dann negiert werden wenn sichergestellt ist, dass die Formeln zum Zeitpunkt des Aufrufs bereits vollständig instantiiert sind.

## 5.5 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 unifizieren. Dabei wird  $Z$  mit  $U$  instantiiert und die Anfrage wird reduziert zu

$$p(U).$$

Da es noch eine weitere Klausel für das Prädikat  $q/1$  gibt, die zur Beantwortung der Anfrage  $q(U)$  in Frage kommt, setzen wir Auswahl-Punkt Nr. 1.

2. Jetzt wird versucht  $p(U)$  mit  $p(X)$  zu unifizieren. Dabei wird die Variable  $X$  an  $U$  gebunden und die ursprüngliche Anfrage wird reduziert zu der Anfrage

$$a(U), b(U), !, c(U,Y), d(Y).$$

Außerdem wird an dieser Stelle Auswahl-Punkt Nr. 2 gesetzt, denn die zweite Klausel des Prädikats  $p/1$  kann ja ebenfalls mit der ursprünglichen Anfrage unifiziert werden.

3. Um die Teilanfrage  $a(U)$  zu beantworten, wird  $a(U)$  mit  $a(1)$  unifiziert. Dabei wird  $U$  mit 1 instantiiert und die Anfrage wird reduziert zu

$$b(1), !, c(1,Y), d(Y).$$

Da es für das Prädikat  $a/1$  noch weitere Klauseln gibt, wird Auswahl-Punkt Nr. 3 gesetzt.

4. Jetzt wird versucht, die Anfrage

$$b(1), !, c(1,Y), d(Y).$$

zu lösen. Dieser Versuch scheitert jedoch, da sich die für das Prädikat  $b/1$  vorliegenden Fakten nicht mit  $b(1)$  unifizieren lassen.

5. Also springen wir zurück zum letzten Auswahl-Punkt (das ist Auswahl-Punkt Nr. 3) und machen die Instantiierung  $U \mapsto 1$  rückgängig. Wir haben jetzt also wieder das Ziel

$$a(U), b(U), !, c(U, Y), d(Y).$$

6. Diesmal wählen wir das Fakt  $a(2)$  um es mit  $a(U)$  zu unifizieren. Dabei wird  $U$  mit 2 instantiiert und wir haben die Anfrage

$$b(2), !, c(2, Y), d(Y).$$

Da es noch eine weitere Klausel für das Prädikat  $a/1$  gibt, setzen wir Auswahl-Punkt Nr.4 an dieser Stelle.

7. Jetzt unifizieren wir die Teilanfrage  $b(2)$  mit der ersten Klausel für das Prädikat  $b/1$ . Die verbleibende Anfrage ist

$$!, c(2, Y), d(Y).$$

8. Diese Anfrage wird reduziert zu

$$c(2, Y), d(Y).$$

Außerdem werden bei diesem Schritt die Auswahl-Punkte Nr. 2 und Nr. 4 gelöscht.

9. Um diese Anfrage zu beantworten, unifizieren wir  $c(2, Y)$  mit dem Kopf der ersten Klausel für das Prädikat  $c/2$ , also mit  $c(2, 2)$ . Dabei erhalten wir die Instantiierung  $Y \mapsto 2$ . Die Anfrage ist damit reduziert zu

$$d(2).$$

Außerdem setzen wir an dieser Stelle Auswahl-Punkt Nr. 5, denn das Prädikat  $c/2$  hat ja noch eine weitere Klausel, die in Frage kommt.

10. Die Anfrage “ $d(2)$ ” scheitert. Also springen wir zurück zum Auswahl-Punkt Nr. 5 und machen die Instantiierung  $Y \mapsto 2$  rückgängig. Wir haben also wieder die Anfrage

$$c(2, Y), d(Y).$$

11. Zur Beantwortung dieser Anfrage nehmen wir nun die zweite Klausel der Implementierung von  $c/2$  und erhalten die Instantiierung  $Y \mapsto 4$ . Die verbleibende Anfrage lautet dann

$$d(4).$$

12. Da sich  $d(4)$  und  $d(3)$  nicht unifizieren lassen, scheitert diese Anfrage. Wir springen jetzt zurück zum Auswahl-Punkt Nr. 1 und machen die Instantiierung  $U \mapsto Z$  rückgängig. Die Anfrage lautet also wieder

$$p(U).$$

13. Wählen wir nun die zweite Klausel der Implementierung von  $q/1$ , so müssen wir  $q(U)$  und  $q(1)$  unifizieren. Diese Unifikation ist erfolgreich und wir erhalten die Instantiierung  $U \mapsto 1$ , die die Anfrage beantwortet.

### 5.5.1 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

`ordered` :  $List(Number) \rightarrow \mathbb{B}$ ,

das überprüft, ob eine Liste geordnet ist. Die bedingten Gleichungen zur Spezifikation der Funktion

`bubble_sort` :  $List(Number) \rightarrow List(Number)$

lauten nun:

1. `append( $l_1, [x, y|l_2], l$ )  $\wedge x > y \rightarrow$  bubble_sort( $l$ ) = bubble_sort(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. `ordered( $l$ )  $\rightarrow$  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. `ordered([]) = true`  
 Die leere Liste ist offensichtlich sortiert.
2. `ordered([ $x$ ]) = true`  
 Eine Liste, die nur aus einem Element besteht, ist ebenfalls sortiert.
3.  `$x \leq y \rightarrow$  ordered( $[x, y|r]$ ) = 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 5.9: Der Bubble-Sort Algorithmus

Abbildung 5.9 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

```
append(L1, [Y, X|L2])
```

gebildet und diese Liste wird rekursiv sortiert. Wenn es nicht möglich ist, die Liste L so in zwei Listen L1 und [X, Y | L2] zu zerlegen, dass Y kleiner als X ist, dann muss die Liste L schon sortiert sein. In diesem Fall greift die zweite Klausel, die allerdings noch Überprüfen muss, ob 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 5.10 zeigt eine Implementierung, bei der nur eine Lösung berechnet wird. Dies wird durch den Cut-Operator in Zeile 4 erreicht. Ist einmal eine Zerlegung der Liste L in L1 und [X, Y | L2] gefunden, bei der Y kleiner als X ist, so bringt es nichts mehr, nach anderen Zerlegungen zu suchen, denn die ursprünglich gegebene Liste L läßt sich ja auf jeden Fall dadurch sortieren, dass rekursiv die Liste

```
append(L1, [Y, X|L2])
```

sortiert wird. Dann kann auch der Aufruf der Prädikats `ordered/1` im Rumpf der zweiten Klausel des Prädikats `bubble_sort` entfallen, denn diese wird beim Backtracking ja nur dann erreicht, wenn es keine Zerlegung der Liste L in L1 und [X, Y | L2] gibt, bei der Y kleiner als X ist. Dann muß aber die Liste L schon sortiert sein.

---

```

1  bubble_sort( List, Sorted ) :-
2      append( L1, [ X, Y | L2 ], List ),
3      X > Y,
4      !,
5      append( L1, [ Y, X | L2 ], Cs ),
6      bubble_sort( Cs, Sorted ).
7
8  bubble_sort( Sorted, Sorted ).
```

---

Abbildung 5.10: 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 5.11 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 5.11: 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.



## 5.6 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 dies an einem Beispiel verdeutlichen. Wir implementieren dazu ein *Prolog*-Programm mit dessen Hilfe es möglich ist, in einem Graphen eine Verbindung von einem gegebenen Start-Punkt zu einem Ziel-Punkt zu finden. Als Beispiel betrachten wir den Graphen in Abbildung 5.12. 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).
```

---

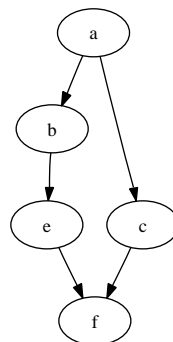


Abbildung 5.12: Ein einfacher Graph ohne Zykeln

Wir wollen nun ein *Prolog*-Programm entwickeln, mit dem es möglich ist für zwei vorgegebene Punkte  $x$  und  $y$  zu entscheiden, ob es einen Weg von  $x$  nach  $y$  gibt. Außerdem soll dieser Weg dann als Liste von Punkten berechnet werden. Unser erster Ansatz besteht aus dem Programm, das in Abbildung 5.13 gezeigt ist. Die Idee ist, dass der Aufruf

`connect(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 5.13: 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 Punkt  $X$ .
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 Punkt  $Y$  gibt und wenn es dann von diesem Punkt  $Y$  eine

Verbindung zu dem Punkt 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 Punkt 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
5  P = [a, c, f] ;
6
7  No

```

---

Durch Backtracking werden also alle möglichen Wege von a nach b gefunden. Als nächstes testen wir das Programm mit dem in Abbildung 5.14 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).

```

---

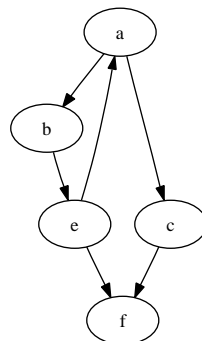


Abbildung 5.14: Ein Graph mit einem Zykel

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 )`.

Jetzt sind wir aber wieder in der selben Situation wie zu Beginn, wir sind in also in einer Endlos-Schleife gefangen. Das Problem ist, das *Prolog* immer die erste Klausel nimmt, die paßt. Wenn später die Reduktion der Anfrage scheitert, wird zwar nach Backtracking die nächste Klausel ausprobiert, aber wenn das Programm in eine Endlos-Schleife läuft, dann gibt es eben kein Backtracking, denn das Programm weiß ja nicht, dass es in einer Endlos-Schleife ist.

Es ist aber leicht das Programm so umzuschreiben, dass keine Endlos-Schleife mehr auftreten kann. Die Idee ist, dass wir uns merken, welche Punkte wir bereits besucht haben und diese nicht mehr auswählen. Wir implementieren also nun ein Prädikat `connect/4`. Die Idee ist, dass der Aufruf

```
connect(Start, Goal, Visited, Path)
```

einen Pfad berechnet, der von *Start* nach *Goal* führt und der zusätzlich keine Punkte benutzt, die in der Liste *Visited* aufgeführt sind. Diese Liste füllen wir bei den rekursiven Aufrufen nach und nach mit den Punkten an, die wir bereits besucht haben. Wir vermeiden mit dieser Liste, einen Punkt zweimal zu besuchen. Abbildung 5.15 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 5.15: 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 Punkte wir schon besucht haben.
2. In der zweiten Klausel überprüfen wir in Zeile 6, ob der Punkt *Y* in der Liste *Visited*, die die Punkte enthält, die bereits besucht wurden, auftritt. Nur wenn dies nicht der Fall ist, versuchen wir rekursiv von *Y* einen Pfad nach *Z* zu finden. Dabei erweitern wir die Liste *Visited* um den Punkt *Y*, den diesen Punkt haben wir ja jetzt besucht.

Mit dieser Implementierung ist es jetzt möglich, auch in dem zweiten Graphen einen Weg von **a** nach **f** zu suchen, wir erhalten

---

```

1  ?- find_path(a,f,[a],P).
2  P = [a, b, e, f] ;
3  P = [a, c, f] ;
4  No

```

---

### 5.6.1 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, dieses Problem durch einen Graphen zu modellieren. Die Punkte 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 gibt an, dass 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 5.16 auf Seite 132 zeigt ein *Prolog*-Programm, was das Rätsel löst. Wir diskutieren dieses Programm nun Zeile für Zeile.

1. Wir beginnen mit dem Hilfs-Prädikat `switch/2`, das in den Zeilen 27 – 32 implementiert ist. Für eine vorgegebene Situation `side(M, K, B)` berechnet der Aufruf

`switch(side(M, K, B), OtherSide)`

einen Term, der die Situation am entgegengesetzten Ufer beschreibt.

2. Das Prädikat `problem/2` in den Zeilen 34 – 38 überprüft, ob es bei einer vorgegeben Anzahl von Missionaren und Kannibalen ein Problem gibt.

3. Bei der Implementierung des Prädikats `edge/2` verwenden wir in den Zeilen 12 und 13 das Prädikat `between/3`, das in dem *SWI-Prolog*-System vordefiniert ist. Beim Aufruf

`between(Low, High, N)`

sind  $Low$  und  $High$  ganze Zahlen mit  $Low \leq High$ . Der Aufruf instantiiert die Variable  $N$  nacheinander mit den Zahlen

$Low, Low + 1, Low + 2, \dots, High$ .

Beispielsweise produziert die Anfrage

`between(1,3,N), write(N), nl, fail.`

die folgende Ausgabe:

```

1
2
3

```

---

```

1  solve :-
2      find_path( side(3,3,1), side(0,0,0), [ side(3,3,1) ], Path ),
3      nl,
4      write('Lösung:' ),
5      nl, nl,
6      print_path(Path).
7
8  % edge( +Point, -Point ).
9
10 edge( side( M, K, 1 ), side( MN, KN, 0 ) ) :-
11     !,
12     between( 0, M, MB ),
13     between( 0, K, KB ),
14     MB + KB >= 1,          % boat must not be empty
15     MB + KB <= 2,          % no more than two passengers
16     MN is M - MB,
17     KN is K - KB,
18     \+ problem( MN, KN ),  % no problem on the left side
19     switch( side(MN, KN, 1), side(MR, KR, 0) ),
20     \+ problem( MR, KR ).  % no problem on the right side
21
22 edge( Side, Next ) :-
23     switch( Side, Switched ),
24     edge( Switched, SwitchedNext ),
25     switch( SwitchedNext, Next ).
26
27 % switch( +Point, -Point ).
28
29 switch( side(M1, K1, B1), side(M2, K2, B2) ) :-
30     M2 is 3 - M1,
31     K2 is 3 - K1,
32     B2 is 1 - B1.
33
34 % problem( +Number, +Number ).
35
36 problem(Missionare, Kannibalen) :-
37     Missionare > 0,
38     Missionare < Kannibalen.

```

---

Abbildung 5.16: Missionare und Kannibalen

4. Die Implementierung des Prädikats `edge/2` besteht aus zwei Klauseln. In der ersten Klausel betrachten wir den Fall, dass das Boot auf dem linken Ufer ist. In den Zeilen 12 und 13 berechnen wir zunächst die Zahl der Missionare und die Zahl der Kannibalen, die im Boot übersetzen sollen. In Zeile 14 testen wir, dass es mindestens einen Passagier gibt, der mit dem Boot übersetzt und in Zeile 15 testen wir, dass es höchstens zwei Passagiere gibt. In Zeile 16 und 17 berechnen wir die Zahl der Missionare und der Kannibalen, die nach der Überfahrt auf dem linken Ufer verbleiben und testen in Zeile 18, dass es am linken Ufer kein Problem für die am verbleibenden Missionare gibt. In Zeile 19 berechnen wir, wie die neue Situation am gegenüber liegenden Ufer aussieht und stellen in Zeile 20 sicher, dass sich auch dort kein Problem ergibt.

Die zweite Klausel befaßt sich mit dem Fall, dass das Boot am rechten Ufer liegt. Wir könnten die zweite Klausel einfach durch Copy&Paste aus der ersten Klausel generieren. Wenn wir die Symmetrie der Situation ausnutzen, geht es aber eleganter. Der erste Aufruf von `switch/2` in Zeile 23 berechnet die Situation *Switched*, die wir am rechten Ufer haben. Wir berechnen nun mit dem Aufruf von `edge/2` in Zeile 24 eine mögliche Folge-Situation für das rechte Ufer. Durch einen weiteren Aufruf von `switch/2` berechnen wir dann die dazu korrespondierende Situation am linken Ufer.

5. In den Zeilen 1 – 6 definieren wir nun das Prädikat `solve/0`, dessen Aufruf das Problem löst. Dazu wird zunächst das Prädikat `find_path/4` mit dem Start-Punkt `side(3,3,1)` und dem Ziel-Punkt `side(0,0,0)` aufgerufen. Der berechnete Pfad wird dann ausgegeben mit dem Prädikat `print_path/1`. Aus Platzgründen wird die Implementierung dieses Prädikats nicht gezeigt. Abbildung 5.17 zeigt den ersten Fahrplan, der mit dem gerade diskutierten Programm generiert wird.

---

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

# Literaturverzeichnis

- [1] *M. Davis and H. Putnam, A computing procedure for quantification theory*, Journal of the ACM, Vol. **7**, pp. 201-214, 1960.
- [2] *M. Davis, G. Logemann, and D. Loveland: A Machine Program for Theorem-Proving*, Communications of the ACM, Vol. **5**, No. 7, pp. 394-397, 1962.
- [3] *Robert K. Dewar with modifications by Robert Hummel:*  
*A Gentle Introduction to the Setl2 Programing Language*,  
verfügbar im Internet unter  
`ftp://cs.nyu.edu/pub/languages/setl2/doc/2.2/intro.ps`,  
Technical Report, New York University 1990.
- [4] *Seymour Lipschutz: Set Theory and Related Topics*, Second Edition, McGraw-Hill, 1998.
- [5] *Uwe Schöning: Logik für Informatiker*, Spektrum Akademischer Verlag, 1987.
- [6] *Jack Schwarz: Programming in SETL*, verfügbar im Internet unter  
`http://www.settheory.com`.
- [7] *W. Kirk Snyder: The SETL2 Programing Language*, verfügbar im Internet unter  
`ftp://cs.nyu.edu/pub/languages/setl2/doc/2.2/report.ps`  
Technical Report, New York University, 1990.
- [8] *W. Kirk Snyder: The SETL2 Programing Language: Update on Current Developments*,  
verfügbar im Internet unter  
`ftp://cs.nyu.edu/pub/languages/setl2/doc/2.2/update.ps`  
Technical Report, New York University, 1990.
- [9] *Leon Sterling and Ehud Shapiro: The Art of Prolog* The MIT Press, 1986.