

# Prolog - Programming in Logic

Patrick Tschiorn  $\diamond$  Kim Wallum  $\diamond$  Timo Steffens

Copyright (c) 2002 Patrick Tschorn, Kim Wallum, Timo Steffens. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with an Invariant Sections being “GNU Free Documentation License”, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

# Inhaltsverzeichnis

<b>Vorwort</b>	<b>ix</b>
<b>Einleitung</b>	<b>xi</b>
<b>1 Aussagen, Anfragen, Regeln</b>	<b>1</b>
1.1 Einfache Aussagen (Fakten) . . . . .	1
1.2 Prolog benutzen . . . . .	2
1.2.1 SWI-Prolog . . . . .	2
1.2.2 VC Prolog Tutor (ehemals Virtueller Campus) . . . . .	3
1.3 Hinweise zu Syntax und Bezeichnungen . . . . .	4
1.4 Konkrete Anfragen stellen . . . . .	5
1.5 Anfragen mit Platzhaltern stellen . . . . .	5
1.5.1 Mehrere Antworten . . . . .	6
1.6 Undverknüpfte Anfragen mit gemeinsamen Platzhaltern . . . . .	7
1.7 Universelle Fakten . . . . .	7
1.7.1 Der anonyme Platzhalter . . . . .	8
1.7.2 Die Ausgabe von Platzhaltern unterdrücken . . . . .	8
1.8 Kompliziertere Aussagen (Regeln) . . . . .	9
1.8.1 Eine <code>eltern</code> -Regel . . . . .	9
1.8.2 Aufbau von Regeln . . . . .	10
1.8.3 Verhalten von Platzhaltern in Regeln . . . . .	10
1.8.4 Bedeutung von Regeln . . . . .	10
1.8.5 Was passiert, wenn eine Anfrage gestellt wird? . . . . .	11
1.9 Redundanz . . . . .	12
1.10 weitere Beispielregeln . . . . .	13
1.10.1 <code>grossmutter</code> - alternative Regelrümpfe . . . . .	13
1.10.2 <code>geschwister</code> - ein merkwürdiges Phänomen . . . . .	14
1.10.3 Verwendung von <code>not</code> . . . . .	16
1.11 <code>vorfahr</code> - eine weitere Spielart von Regeln . . . . .	16
1.11.1 Beobachtung der Anfrage <code>vorfahr(X, jens)</code> . . . . .	18
1.12 das vollständige Beispielprogramm . . . . .	23
<b>2 Terme, Strukturen, Unifikation</b>	<b>27</b>
2.1 Terme . . . . .	27
2.2 Strukturen . . . . .	27
2.2.1 Strukturen mit einfachen Termen als Komponenten . . . . .	28
2.2.2 Bedeutung von Strukturen als Argumente von Aussagen .	28

2.2.3	Verhalten von Platzhaltern in Strukturen . . . . .	29
2.2.4	ausführliche Beispielanfragen . . . . .	29
2.2.5	Platzhalter durch Strukturen ersetzen . . . . .	30
2.2.6	Strukturen mit Strukturen als Komponenten . . . . .	31
2.2.7	Beispielanfragen . . . . .	32
2.3	Unifikation . . . . .	34
2.3.1	Zusammenfassung des Unifikationsverfahrens . . . . .	37
2.3.2	Test auf Vorkommen . . . . .	37
<b>3</b>	<b>rekursive Datenstrukturen</b>	<b>39</b>
3.1	binäre Bäume . . . . .	39
3.1.1	Repräsentation in Prolog . . . . .	39
3.1.2	Beispiele für Zugriff . . . . .	41
3.2	Listen . . . . .	43
3.2.1	Repräsentation in Prolog . . . . .	43
3.2.2	grafische Darstellung . . . . .	44
3.2.3	verschiedene Notationen . . . . .	45
3.2.4	Gegenüberstellung der Notationen . . . . .	45
3.2.5	Listen von Listen . . . . .	46
3.2.6	Verhalten von Platzhaltern in Listen . . . . .	46
3.2.7	Unifikation von Listen . . . . .	46
3.2.8	ein beliebter Fehler . . . . .	47
3.2.9	ASCII-Zeichencodes . . . . .	48
3.3	Strukturen vs. Listen . . . . .	48
3.3.1	“univ” . . . . .	48
3.4	einige “neue” Begriffe . . . . .	48
<b>4</b>	<b>rekursive Programme</b>	<b>51</b>
4.1	Rekursion . . . . .	51
4.1.1	Sukzessornotation in Prolog . . . . .	52
4.1.2	Weitere Aspekte . . . . .	53
4.1.3	Iterative vs. rekursive Schleifen . . . . .	53
4.2	Rekursive Programme auf Listen . . . . .	53
4.2.1	Auf Liste prüfen / Liste generieren . . . . .	53
4.2.2	Ausführliche Beispiele, Rekursionsabstieg und -aufstieg . . . . .	55
4.2.3	Veranschaulichung von Rekursionsabstieg und -aufstieg . . . . .	57
4.2.4	Die andere Richtung - Erzeugen von Listen . . . . .	58
4.2.5	<code>is_list/1</code> , schwächere Version des Checks . . . . .	58
4.2.6	Rekursive Suche - <code>enthalten/2</code> . . . . .	59
4.2.7	<code>write/1</code> , <code>n1/0</code> - Ausgabe . . . . .	60
4.2.8	Umdrehen von Listen, Strukturaufbau beim Rekursionsabstieg . . . . .	60
4.2.9	Restrekursion . . . . .	63
4.2.10	<code>verkette/3</code> - Listen verketten, Strukturaufbau beim Rekursionsaufstieg . . . . .	63
4.2.11	Suffix, Präfix, Infix - Listen . . . . .	64
4.2.12	Löschen von Elementen aus Listen . . . . .	64
4.2.13	Menge, Teilmenge, Vereinigung, Schnitt . . . . .	64
4.2.14	Permutationen von Listen . . . . .	64
4.2.15	Sortieren von Listen . . . . .	64

4.2.16	Flachklopfen von verschachtelten Listen . . . . .	64
4.2.17	Umformungen von Listen . . . . .	64
4.2.18	Palindrome . . . . .	64
4.3	Rekursive Programme auf Bäumen . . . . .	64
4.3.1	Traversierung . . . . .	64
4.3.2	Prolog, Logik und Ein-/Ausgabe . . . . .	65
<b>5</b>	<b>Die Beweisprozedur von Prolog</b>	<b>67</b>
5.1	Klauselauswahl, Choicepoints . . . . .	67
5.2	Backtracking . . . . .	67
5.3	Und-/Oder-Bäume . . . . .	68
5.3.1	Beispielbaum . . . . .	68
5.3.2	Beispielbeweis . . . . .	71
5.3.3	Indexing . . . . .	75
5.4	Ablaufverfolgung / Trace . . . . .	76
5.4.1	Das Vierportmodell . . . . .	77
5.5	Prozedurales Programmieren in Prolog . . . . .	77
5.6	Implikationen des Prologbeweisverfahrens . . . . .	78
5.6.1	Endlosschleifen . . . . .	78
5.6.2	Reihenfolgeprobleme . . . . .	78
5.6.3	Programmierprinzipien . . . . .	78
5.7	alternative Beweisstrategien . . . . .	78
5.7.1	Beispiele . . . . .	78
5.8	Das Prädikat <b>fail</b> . . . . .	79
5.8.1	Backtrackingschleifen . . . . .	79
5.9	Der Cut . . . . .	80
5.9.1	“guarded gate”-Metapher . . . . .	80
5.9.2	Beispiel . . . . .	80
5.9.3	“rote” und “grüne” Cuts . . . . .	81
5.10	<b>[cut]-[fail]</b> Kombinationen zur Simulation von Negation . . . . .	81
5.10.1	Das Prädikat <b>not</b> . . . . .	82
5.10.2	weitere Aspekte des Cuts . . . . .	83
5.11	Prädikate kommentieren II – Konventionen + - ? . . . . .	84
5.12	Typprädictate . . . . .	84
<b>6</b>	<b>Die Beziehung zwischen Prolog und der PL1</b>	<b>85</b>
6.1	Einführung . . . . .	85
6.2	Syntax der PL1 . . . . .	85
6.2.1	Quantoren . . . . .	86
6.2.2	Junktoren . . . . .	86
6.3	Semantik der PL1 . . . . .	87
6.3.1	Junktoren . . . . .	87
6.4	Transformationen . . . . .	87
6.5	Konjunktive Normalenform . . . . .	88
6.5.1	Umwandlung von Formeln in die KNF . . . . .	88
6.5.2	KNF und Prolog . . . . .	89
6.5.3	Closed-world-assumption . . . . .	90
6.6	Resolution . . . . .	90
6.6.1	Logische Resolution für Aussagenlogik . . . . .	90
6.6.2	Logische Resolution für Prädikatenlogik . . . . .	91

6.6.3	Resolution in Prolog . . . . .	92
<b>7</b>	<b>Arithmetik und Operatoren</b>	<b>95</b>
7.1	Operatoren . . . . .	95
7.1.1	Operatoren und Argumente, In-, Pre- und Postfixnotation	95
7.1.2	Umwandlung in Prefixnotation, Vorrang und Assoziativität von Operatoren . . . . .	96
7.1.3	vordefinierte Operatoren . . . . .	97
7.1.4	eigene Operatordefinitionen . . . . .	98
7.1.5	Ausführliche Beispiele zur Umwandlung von Ausdrücken in Prefixnotation . . . . .	99
7.1.6	Semantik von Operatoren . . . . .	99
7.2	Arithmetik . . . . .	100
7.2.1	Arithmetische Ausdrücke, <code>[is]</code> . . . . .	101
7.2.2	eigene Arithmetikprädikate . . . . .	102
7.2.3	Länge einer Liste . . . . .	102
7.2.4	Zugriff auf n-tes Element . . . . .	103
<b>8</b>	<b>Graphen und Suche</b>	<b>105</b>
8.1	Einleitung . . . . .	105
8.2	Graphen . . . . .	106
8.2.1	DAGs . . . . .	107
8.2.2	Anwendungen von Graphen und Suche . . . . .	107
8.2.3	Anmerkungen . . . . .	110
8.3	Graphen und Suche in Prolog . . . . .	110
8.3.1	mit einfachsten Mitteln . . . . .	111
8.3.2	verbessertes Vorgehen . . . . .	113
8.3.3	Tiefensuche . . . . .	118
8.3.4	Tiefensuchalgorithmus . . . . .	119
8.3.5	Tiefensuchalgorithmus in Prolog . . . . .	119
8.3.6	Breitensuche . . . . .	125
8.3.7	Anmerkungen . . . . .	128
8.3.8	Veranschaulichung von Tiefen- und Breitensuche . . . . .	129
8.3.9	Best-First . . . . .	135
8.3.10	Vergleich der drei Verfahren . . . . .	136
8.3.11	Ausblick . . . . .	136
8.4	to do . . . . .	136
<b>9</b>	<b>Sprachverarbeitung</b>	<b>137</b>
9.1	Natürliche und formale Sprachen . . . . .	137
9.1.1	Ein Beispiel einer Grammatik . . . . .	138
9.1.2	Kontextfreie Grammatiken . . . . .	140
9.2	Kontextfreie Grammatiken in Prolog . . . . .	140
9.2.1	DCG . . . . .	143
9.2.2	Generierung . . . . .	144
9.2.3	Zusätzliche Argumente . . . . .	145
9.2.4	Einfügen von Prolog-Code . . . . .	146
9.2.5	Anmerkungen . . . . .	147
9.2.6	Syntaxstrukturen . . . . .	147
9.2.7	Ambiguität . . . . .	148

9.3 Todo . . . . .	149
<b>10 Ein- und Ausgabe</b>	<b>151</b>
10.1 Lesen und Schreiben von Termen . . . . .	151
10.1.1 Lesen von Termen: <code>read/1</code> . . . . .	151
10.1.2 Schreiben von Termen: <code>write/1</code> . . . . .	152
10.2 Lesen und Schreiben von Zeichen . . . . .	152
10.2.1 Lesen von Zeichen: <code>get/1</code> , <code>get0/1</code> . . . . .	152
10.2.2 Schreiben von Zeichen: <code>put/1</code> . . . . .	153
10.2.3 Tabulatoren und Zeilenumbruch . . . . .	153
10.3 Ein- und Ausgabe auf Dateien . . . . .	154
<b>11 Dynamische Änderung der Wissensbasis</b>	<b>157</b>
11.1 <code>asserta/1</code> , <code>assertz/1</code> und <code>retract/1</code> . . . . .	157
11.1.1 <code>consult/1</code> . . . . .	157
11.1.2 <code>findall/3</code> . . . . .	159
11.2 Weitere Prädikate zum Zugriff auf die Wissensbasis . . . . .	159
11.3 todo . . . . .	159
<b>GNU Free Documentation License</b>	<b>161</b>
11.1 Applicability and Definitions . . . . .	161
11.2 Verbatim Copying . . . . .	162
11.3 Copying in Quantity . . . . .	163
11.4 Modifications . . . . .	163
11.5 Combining Documents . . . . .	165
11.6 Collections of Documents . . . . .	165
11.7 Aggregation With Independent Works . . . . .	166
11.8 Translation . . . . .	166
11.9 Termination . . . . .	166
11.10 Future Revisions of This License . . . . .	166



# Vorwort

All programming languages found in the classroom are prone to a wide range of misconceptions – but arguably, the language that has been found to engender the greatest number of problems is Prolog.<sup>1</sup>

Dieser Text soll eine möglichst verständliche Einführung in die Programmiersprache Prolog darstellen.

Kim Wallum hat die Übersetzung ins Englische durchgeführt.

Das Kapitel über die Beziehung zwischen Prolog und der PL1 stammt von Timo Steffens.

Tobias Thelen hat die erste Version dieses Textes sehr gewissenhaft und kritisch gegengelesen.

Dieser Text steht unter der GNU Free Documentation License, damit ist jeder herzlich eingeladen, ihn frei zu verwenden und zu erweitern.

Die L<sup>A</sup>T<sub>E</sub>X-Quellen können von folgender URL bezogen werden:

todo :-)

Stand: 24. Januar 2004

Patrick Tschorn

---

<sup>1</sup>( P. Mendelsohn et al., 1990, Programming Languages in Education: The Search for an Easy Start, In: Psychology of Programming, Seite 186, 1990, Academic Press Limited)



# Einleitung

Prolog wurde um 1970 von Alain Colmerauer und anderen an der Universität von Aix-Marseille entwickelt. Ziel war es, natürliche Sprache zu verarbeiten. Prolog ist neben Lisp die Standardsprache im Bereich Computerlinguistik und künstlicher Intelligenz.

Prolog steht für "PROgramming in LOGic". Die Prolog zugrundeliegende Logik ist die so genannte Hornlogik. Sie ist eine Untergruppe der Prädikatenlogik erster Stufe (PL1). Man erstellt in Prolog eine so genannte Wissensbasis, indem man eine Reihe von Aussagen formuliert. Mit diesen Aussagen werden Dinge/Individuen und deren Beziehungen (Relationen) zueinander beschrieben. In der Logik werden die Aussagen der Wissensbasis als Axiome bezeichnet. Anschließend möchte man wissen, ob sich bestimmte weitere Aussagen aus den Axiomen ableiten lassen. Dazu werden so genannte Anfragen gestellt. Prolog versucht nun, die in einer Anfrage enthaltene Aussage aus der Wissensbasis abzuleiten. Für die Hornlogik existiert ein effizientes mechanisches Verfahren, das diese Aufgabe erfüllt: die Resolution.

\*[vage:Wissensbasis, vage:Anfragen]

\*[vage:Resolution]

Von diesem "logischen Background" sollte man sich jedoch nicht abschrecken lassen und Prolog einfach als Programmiersprache sehen, die man Schritt für Schritt lernt. Wir werden diesen Weg gehen und die formalen logischen Hintergründe erst in einem späteren, eigenständigen Kapitel betrachten.

Prologprogramme haben zum einen eine deklarative (logische) Bedeutung, aber zum anderen auch eine prozedurale Bedeutung, wie man sie aus konventionellen Programmiersprachen, etwa C, kennt.

\*[vage:prozedurale vs deklarative Bedeutung]

Prolog kommt mit einer minimalistischen Syntax aus. Es gibt nur einen Datentyp – den *Term*, durch den Programme und Daten gleichermaßen dargestellt werden.

Prolog ist eine interaktive Sprache, d.h. der Benutzer führt eine Art Dialog mit dem System. In diesem Dialog werden Anfragen gestellt. So gesehen ähnelt Prolog Lisp, in einen Topf darf man die beiden Sprachen jedoch keinesfalls werfen.

\*[vage:Interaktion]

Prolog ist von der Handhabung her eine interpretierte Sprache. Allerdings werden Prologprogramme zumeist in eine Art Bytecode für eine virtuelle Maschine, die so genannte Warren Abstract Machine (WAM), kompiliert.

In Prolog gibt es zunächst keine Objektorientierung. Verschiedene ausgewachsene Prologsysteme bieten (proprietäre) objektorientierte Erweiterungen.

Dieses Skript stützt sich auf das von Jan Wielemaker geschriebene SWI-Prolog und den Virtuellen Campus. SWI-Prolog ist mitsamt Referenzhandbuch für eine breite Palette von Plattformen erhältlich. Es ist auf den meisten Rechnern der Universität installiert. Für zuhause kann man das System von der folgenden Internetadresse beziehen. Es ist außerdem Bestandteil der meisten

Linux-Distributionen.

<http://swi.psy.uva.nl/projects/SWI-Prolog/download.html>  
<ftp://swi.psy.uva.nl/pub/SWI-Prolog/>

Der Virtuelle Campus ist eine am Institut für Semantische Informationsverarbeitung entwickelte netzwerkgestützte Lernumgebung. Es existiert eine gedruckte Einführung in dieses System. Des Weiteren geben die (menschlichen) Tutoren gerne Auskunft.

Die Startseite des Virtuellen Campus verbirgt sich hinter

<http://www.virtcampus.uni-osnabrueck.de/>

Hier befinden sich neben den verschiedenen Kursen hilfreiche Ressourcen wie z.B. das VC-Skript, das eine Vielzahl wichtiger Konzepte in einer Ontologie zur Verfügung stellt. Um an einem Kurs teilnehmen zu können, benötigt jeder Benutzer ein so genanntes Login. Logins werden im Rahmen der Vorlesung vergeben.

Eine sehr umfangreiche Sammlung von Links zum Thema Prolog befindet sich bei

<http://archive.comlab.ox.ac.uk/logic-prog.html>

# Kapitel 1

## Aussagen, Anfragen, Regeln

### 1.1 Einfache Aussagen (Fakten)

Die Wissensbasis in Prolog wird unter Verwendung von Aussagen aufgebaut. Diese Aussagen betreffen Individuen (Dinge), die Eigenschaften von Individuen und die Beziehungen (Relationen) zwischen Individuen. Synonym zu Wissensbasis werden oft die Begriffe Universum, Welt, Weltausschnitt und Domäne gebraucht.

In nahezu jeder Prolog-Einführung müssen Verwandtschaftsbeziehungen verschiedener Mitglieder einer großen Familie als einleitendes Beispiel herhalten. Wir wollen hier nicht mit Traditionen brechen. Zunächst brauchen wir einige tapfere Frauen und Männer. In der folgenden Tabelle stehen in der linken Spalte natürlichsprachliche Aussagen, in der rechten Spalte sind entsprechende Formulierungen in Prolog-Syntax aufgeführt.

natürliche Sprache:	Übersetzungen in Prolog:
Maria ist eine Frau.	<code>frau(maria).</code>
Michaela ist eine Frau.	<code>frau(michaela).</code>
Jens ist ein Mann.	<code>mann(jens).</code>
Bernd ist ein Mann.	<code>mann(bernd).</code>

Nun wollen wir einige Beziehungen zwischen diesen Individuen formulieren:

natürliche Sprache:	Übersetzungen in Prolog:
Maria ist die Mutter von Jens.	<code>mutter(maria, jens).</code>
Anke ist die Mutter von Maria.	<code>mutter(anke, maria).</code>
Bernd ist der Vater von Jens.	<code>vater(bernd, jens).</code>

Diese Art von Aussagen in Prolog werden *Fakten* genannt. Ein Fakt wie z.B. \*[Aussagen:Fakten]

`frau(maria).` spricht man als “[f]rau von [maria]”, d.h. die Eigenschaft, [f]rau zu sein, trifft auf [maria] zu. Was [f]rau im eigentlichen Sinne bedeutet, weiß Prolog nicht. Wir hätten ebensogut `woman(maria).` oder `g123xy(abc).` schreiben können, denn auch [maria] ist nur ein willkürlicher Bezeichner. Wir müssen wissen, was wir mit `frau(maria).` meinen und uns in der gesamten Wissensbasis an diese selbst geschaffene Konvention halten. Für den Fall, dass

es mehrere Marias gibt, die unterschieden werden sollen, bietet es sich an, diese auch unterschiedlich zu benennen: `frau(maria_1).`, `frau(maria_2).`, usw. statt fünfmal `frau(maria).`. Nur dadurch, dass ihre Bezeichner unterschiedlich sind, können die Marias auseinander gehalten werden. Ein weiteres Beispiel ist `vater(bernd,jens).`: Wir müssen uns entscheiden, ob `bernd` `vater` von `jens` ist oder etwa `jens` der `vater` von `bernd` sein soll. Die Reihenfolge der so genannten *Argumente* spielt eine wichtige Rolle, sie muss konsequent durchgehalten werden.

Als letztes Beispiel dieser Art soll uns `lieben(hans, maria).` dienen. Wir müssen nicht nur die Reihenfolge (wer liebt wen?) kennen, sondern auch die Möglichkeit in Betracht ziehen, dass diese Relation symmetrisch gemeint ist, die beiden sich also gegenseitig lieben. Die `vater`-Relation sollte dagegen nicht symmetrisch gemeint sein. Es ist keine schlechte Idee, in einem Kommentar die intendierte Bedeutung - oft *Key* genannt - festzuhalten. Wie solche Kommentare aussehen können, zeigt das Listing des kompletten Beispielprogramms, das wir in diesem Kapitel erstellen werden, in Abschnitt 1.12 auf Seite 23.

\*[Semantik: Key]

## 1.2 Prolog benutzen

Nun werden wir dem Prologsystem diese Fakten als Wissensbasis zur Verfügung stellen und anschließend eine Beispielenfrage stellen. Zunächst schauen wir uns an, wie man SWI-Prolog benutzt, anschließend wenden wir uns dem Virtuellen Campus zu.

### 1.2.1 SWI-Prolog

?[Umgang mit Shell + Texteditor + Dateisystem]

Mit einem Texteditor erstellen wir eine Datei mit folgendem Inhalt:

```
frau(maria).
frau(michaela).
```

Nun speichern wir sie unter `dateiname.pl` ab. Die Endung `.pl` ist eine gängige Konvention. `.pro` ist ebenfalls weit verbreitet. Als nächstes müssen wir das Prologsystem starten. Auf Windows-Maschinen muss dazu `plwin.exe` aus dem `bin`-Verzeichnis der SWI-Installation ausgeführt werden. Bei Unix/Linux genügt es, das Kommando `p1` in einer Shell auszuführen. Das sollte eigentlich kein Problem darstellen. Falls es doch Schwierigkeiten gibt, kann man sich vertrauensvoll an erfahrenere Studenten wenden. Jetzt sollte man etwa folgende Bildschirmausgabe sehen:

```
Welcome to SWI-Prolog (Version 2.8.6)
Copyright (c) 1993-1997 University of Amsterdam. All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
```

\*[SWI: Prompt]

Das Zeichen `?-` ist der so genannte Prompt. Hinter diesem Zeichen werden wir unsere Anfragen eintippen. Mit der Eingabe `consult('dateiname.pl')`.

veranlassen wir das System, unsere eben angelegte Datei als Wissensbasis zu laden. Der Punkt (.) kennzeichnet das Ende der Anfrage. Vergisst man die-  
sen Punkt und drückt die Eingabetaste, so antwortet Prolog mit einem neuen  
Prompt, und zwar so lange bis man sich erbarmt, dem System schließlich doch  
einen Punkt zu spendieren. Ging alles gut, sollte etwa die folgende Ausgabe  
erscheinen:

```
1 ?- consult('erstefakten.pl').
erstefakten.pl compiled, 0.00 sec, 748 bytes.
```

Yes  
2 ?-

Als abkürzende Schreibweise für `consult('x.pl').` kann `consult(x).`,  
`['x.pl'].`, bzw. `[x].` dienen. Weitere wichtige Kommandos sind `halt.`  
zum Verlassen des Systems und `edit('dateiname.pl')`. zum Aufrufen eines Editors. Ruft man einen Editor aus SWI-Prolog heraus auf, blockiert SWI-Prolog so lange, bis der Editor wieder beendet wird. Sollte SWI-Prolog die editierte Datei nicht automatisch neu laden, muß man wiederum `consult('dateiname.pl')` bemühen. `edit('dateiname.pl')` zaubert auf Unix/Linux zumeist den vi hervor. Im SWI-Manual ist beschrieben, wie man einen anderen Editor einsetzen kann. Nun steht dem System also eine Wissensbasis zur Verfügung. Eine erste einfache Anfrage könnte zum Beispiel so aussehen:

```
2 ?- frau(maria).
```

Yes  
3 ?-

Es handelt sich hierbei um die einfache Frage, ob in der Wissensbasis der Fakt `frau(maria).` steht. Bevor wir richtig loslegen, versuchen wir dasselbe im Virtuellen Campus.

\*[einfache Anfrage]

### 1.2.2 VC Prolog Tutor (ehemals Virtueller Campus)

Voraussetzung für die folgenden Schritte ist natürlich eine erfolgreiche Anmeldung im Virtuellen Campus. Ein Prologprogramm lässt sich unmittelbar in das Editorfenster eintippen. Bevor Anfragen gestellt werden können, muss das Programm kompiliert werden. Danach öffnet sich ein Fenster, in das nacheinander Anfragen eingegeben werden können. Probieren wir es aus:

- Prologprogramm in das Editorfenster eingeben

```
frau(maria).
frau(michaela).
```

Diese Sammlung von Fakten kann man bereits als Prologprogramm auffassen.

- Programm kompilieren

Dazu muss im Menüpunkt **Services** der Eintrag **Compile** ausgewählt werden.

- Anfragen stellen

Nach dem Kompilieren steht die Wissensbasis dem Prologsystem zur Verfügung. Es erscheint ein Fenster, in dem man seine Anfragen absetzen kann. Wir tippen **[frau(maria).]** in die obere Zeile des Fensters und klicken auf **execute**. Im unteren Teil des Fensters sollte folgender Text erscheinen:  
**Yes**

Die Ausgabe von Yes bestätigt, dass der Fakt, der die Eigenschaft Marias, eine Frau zu sein, beschreibt, in der Wissensbasis eingetragen ist.

Die Grundzüge der Bedienung der beiden Systeme haben wir also kennen gelernt. Für das Skript ist es unerheblich, welches System Sie benutzen. Im Rahmen des Übungsbetriebes wird der Virtuelle Campus verwendet.

### 1.3 Hinweise zu Syntax und Bezeichnungen

Wir klären einige Syntaxmerkmale anhand des folgenden Faktes:

**[frau(maria).]**

\*[Funktor]  
*Syntax*  
\*[Argumente]

\*[Stelligkeit]

\*[Atom]

\*[Konstanten]  
\*[integerkonstanten]

\*[Kommentar]

**[frau]** wird als *Funktor* bezeichnet. Unmittelbar auf den Funktor folgen die in runden Klammern eingeschlossenen *Argumente*. Mit einem Punkt (**[.]**) wird die Aussage abgeschlossen. Zwischen dem Funktor und der öffnenden Klammer darf kein Leerzeichen stehen! Das einzige Argument in diesem Beispiel ist **[maria]**. Mehrere Argumente werden durch Kommata getrennt. Die Anzahl der Argumente wird als *Stelligkeit* bezeichnet, der Beispiefakt ist also *einstellig* - man fügt die Stelligkeit oft an den Funktor an, um deutlich zu machen, über welche Aussagen man spricht. **[frau/1]** meint eine andere Sammlung von Fakten als beispielsweise **[frau/2]**, daher ist die Unterscheidung wichtig. Funktoren müssen mit einem Kleinbuchstaben beginnen.

**[maria]** an sich wird als *Atom* bezeichnet. Prologatome lassen sich nicht in kleinere Einheiten zerlegen - daher der Name. Atome müssen mit einem Kleinbuchstaben beginnen. (Will man Atome, die mit Großbuchstaben oder Zahlen beginnen, so muss man diese in einfachen Hochkommata einschließen.) Atome sind *Konstanten*, sie lassen sich nicht "ändern". Neben den Atomen sind die ganzen Zahlen (Integer) eine weitere Art der Konstanten. Atome und Argumente sind verschiedene Dinge! Atome dürfen als Argumente verwendet werden, aber nicht alle Argumente sind automatisch Atome!

Der häufigste Fehler dürfte wohl sein, ein Leerzeichen zwischen den Funktor und die öffnende runde Klammer zu setzen. Ein weiterer Fehler, der oft auftritt, ist, den abschließenden Punkt wegzulassen. Auch Anfragen müssen mit einem **[.]** abgeschlossen werden.

Es gibt zwei Möglichkeiten, Bemerkungen in den *Quelltext* einzufügen: **/\*** und **\*/** schließen einen Kommentar ein, **%** verwandelt die folgenden Zeichen bis zum Zeilenende in einen Kommentar. Kommentare werden vom Prologsystem ignoriert und dienen dazu, erhellende Bemerkungen in natürlicher Sprache in einer Quelldatei festzuhalten.

## 1.4 Konkrete Anfragen stellen

Wir wollen folgendes Prologprogramm für eine Reihe von Anfragen benutzen: \*[konkrete Anfragen]

```
frau(maria).
frau(michaela).
mann(jens).
mann(bernd).
mutter(maria, jens).
mutter(anke, maria).
vater(bernd, jens).
```

Stellen wir die Anfrage `[frau(michaela).]`, so erhalten wir vom System die Antwort `yes`, also die Bestätigung, dass diese Aussage aus der Wissensbasis abgeleitet werden kann. (Das ist kein Kunststück, da sie Teil der Wissensbasis ist.) Ein wenig interessanter ist das Ergebnis der Anfrage `[frau(franziska).]`. Die Antwort des Systems lautet `no`. Das bedeutet nicht etwa, dass `franziska` keine Frau ist, sondern vielmehr, dass diese Aussage nicht aus der Wissensbasis abgeleitet werden kann. Man spricht davon, dass das System diese Aussage nicht beweisen kann. Es besteht ein Unterschied zwischen dem fehlgeschlagenen Beweis einer Anfrage und der Negation einer Aussage! Analog zu diesen Beispielen lassen sich weitere Anfragen stellen.

\*[vage: Closed World Assumption, vage: Negation]

## 1.5 Anfragen mit Platzhaltern stellen

Man kann in Prolog mehr als nur abklopfen, was man wortwörtlich eingegeben hat. Es lassen sich Anfragen mit Platzhaltern formulieren. Eine Anfrage mit Platzhalter bedeutet deklarativ: "Ist die Aussage wahr, dass es (mindestens) einen Kandidaten gibt, der die Anfrage erfüllt?" Die prozedurale Bedeutung lautet: "Versuche, einen Kandidaten zu finden, der die Anfrage erfüllt!"

\*[Anfragen mit Platzhaltern]
? [Anfragen]

Wir verwenden weiterhin das Programm aus dem vorigen Abschnitt. Diesmal wollen wir die folgende Anfrage stellen: `[frau(X).]`

\*[prozedurale vs. deklarative Semantik von einfachen Anfragen]

Natürlichsprachlich formuliert hieße diese Anfrage: "Gibt es ein `[X]`, für das `frau(X).` gilt?" Oder einfacher: "Welches `[X]` ist eine `[frau]`?"

```
2 ?- frau(X).          % unsere Anfrage in SWI-Prolog
X = maria              % diese Antwort mit Return bestätigen
Yes                   % Der Beweis ist gelungen.
```

Im Virtuellen Campus entfällt das Bestätigen mit Return. Das System antwortet mit `yes`. Der Beweis ist also gelungen. Darüber hinaus gibt das System noch eine weitere Information preis: `X = maria`. Prolog hat den Platzhalter `[X]` mit `maria` ausgefüllt bzw. *belegt*. Man sagt auch: "`[X]` wurde mit `maria` instantiiert." Mit der gewählten Instantiierung `X = maria` ist der Beweis gelungen. Da die Existenz eines einzigen `[X]` mit `frau(X)` für das Gelingen des Beweises

\*[Instantiierung]

*Syntax*

ausreicht, spricht man davon, dass Anfragen in Prolog *implizit existenzquantifiziert* sind. (Diese Betrachtungs- und Benennungsweise stammt aus der Logik.)

“Namen” von Platzhaltern beginnen in Prolog mit einem Großbuchstaben. (Nach dem ersten Buchstaben darf beliebig groß oder klein weitergeschrieben werden.) **[X]**, **[Y]**, **[Z123ab]**, **[HANS]** und **[Maria]** sind demnach Platzhalter, während **[x]**, **[y]**, **[z123ab]**, **[hans]** und **[maria]** Atome sind.

Platzhalter in Prolog sind *logische Variablen*. Sie haben andere Eigenschaften als die Variablen in einer konventionellen Programmiersprache wie etwa C. Um eine Verwechslung der Eigenschaften von Prolog-Variablen und Variablen aus anderen Programmiersprachen zu vermeiden, behalten wir zunächst den Begriff Platzhalter bei. (Es ist jedoch üblich, auch in Prolog von Variablen zu sprechen.) Platzhalter sind zunächst uninstantiiert. Wurde einem Platzhalter ein Wert zugewiesen, so behält der Platzhalter diesen Wert für die Dauer des aktuellen Beweises bei, der Wert lässt sich nicht mehr ändern. Erst bei der Suche nach weiteren Antworten probiert das System andere Belegungen aus.

Weitere Beispieldaten:

3 ?- frau(Y).

Y = maria

Yes

4 ?- mann(Z123ab).

Z123ab = jens

Yes

5 ?- mutter(Mutter, maria).

Mutter = anke

Yes

6 ?- vater(VATER, N).

VATER = bernd

N = jens

Yes

7 ?- mutter(Mutter, N).

Mutter = maria

N = jens

Yes

Die letzten beiden Anfragen machen deutlich, dass man beliebig viele Platzhalter verwenden kann.

### 1.5.1 Mehrere Antworten

\*[mehrere Antworten]  
?[Anfragen]

Um die Anfrage **frau(X).** beweisen zu können, reicht bereits ein einziges **[X]** mit **[frau(X)]** aus. In unserer Wissensbasis gibt es mehrere solche **[X]**. Man spricht dann davon, dass die Anfrage mehrere Lösungen hat.

Bestätigt man in SWI-Prolog die erste Antwort (Lösung) einer Anfrage mit Platzhalter(n) nicht mit Return, sondern mit **[;]**, erhält man weitere Lösungen,

## 1.6. UNDVERKNÜPFTE ANFRAGEN MIT GEMEINSAMEN PLATZHALTERN

falls diese existieren.

```
?- frau(X).  
X = maria ;  
X = michaela ;  
No
```

Die erste vom System für **[X]** gefundene Belegung, für die der Beweis gelingt, ist **[maria]**. Der Beweis gelingt auch, wenn das System für **[X] [michaela]** einsetzt. Da das System anschließend keine weiteren *Alternativen* für die Belegung von **[X]** findet, gibt es **[no]** aus. **[no]** bedeutet, dass keine weiteren Lösungen konstruiert werden konnten. Im VC-Prolog Tutor tritt der Menüeintrag ‘nächste Lösung’ an die Stelle des ‘;’.

## 1.6 Undverknüpfte Anfragen mit gemeinsamen Platzhaltern

Es ist möglich, mehrere Anfragen mit **,** zu verketteten. Das System liefert **Yes**, wenn *alle* Teile der Anfrage bewiesen werden konnten, anderenfalls **[no]**. Die Anfragen sind also undverknüpft.

```
?- vater(bernd,jens), mutter(maria,jens).  
Yes
```

Verkettete Anfragen können gemeinsame Platzhalter besitzen:

```
?- vater(bernd,X), mutter(maria,X).  
X = jens  
Yes
```

Alle Vorkommen von **[X]** müssen im obigen Beispiel durch **[jens]** ersetzt werden, egal, in welcher Teilanfrage sie stehen. Der Platzhalter **[X]** wird also von beiden Teilanfragen gemeinsam genutzt. Ein weiteres Beispiel:

```
?- vater(Vater,Kind), mutter(Mutter,Kind).  
Vater = bernd  
Kind = jens  
Mutter = maria  
Yes
```

Das Komma zwischen zwei Teilanfragen entspricht also dem logischen “und”. Es sollte nicht mit dem Komma zwischen den Argumenten verwechselt werden, das nur als Trennzeichen fungiert. Der Gültigkeitsbereich eines Platzhalters erstreckt sich über die (einfache oder komplexe) Aussage, in der der Platzhalter auftritt.

## 1.7 Universelle Fakten

Wir wollen ausdrücken, dass alle Familienmitglieder aus unserem Beispielprogramm Bananen mögen. Eine Möglichkeit wäre es, für jede Person einen Fakt einzutragen:

\*[undverknüpfte Anfragen mit gemeinsamen Platzhaltern]  
?[Anfragen, Platzhalter]  
\*[Undverknüpfung]

\*[universelle Fakten]  
?[Fakten, Platzhalter]

```
mag(maria, bananen).
mag(michaela, bananen).
mag(jens, bananen).
mag(bernd, bananen).
% ...
```

Eine elegantere Möglichkeit ergibt sich unter Verwendung von Platzhaltern:

```
mag(X, bananen). % Wer/was immer X ist, er/sie/es mag Bananen.
```

Auf diese Weise lassen sich also universelle Fakten formulieren. Während Anfragen implizit existenzquantifiziert sind, sind universelle Fakten implizit allquantifiziert. Für alle **[X]** gilt: **[mag(X, bananen)..]**

\*[implizite Allquantifikation]

\*[anonyme Platzhalter]

?[Platzhalter, Fakten, Anfragen]

### 1.7.1 Der anonyme Platzhalter

In diesem Beispiel ist es unerheblich, welchen Inhalt der Platzhalter **[X]** zugewiesen bekommt. In einer solchen Situation würde man statt **[X]** das Zeichen **[\_]**, den so genannten *anonymen* Platzhalter verwenden. Die zentralen Eigenschaften des anonymen Platzhalters sind zum einen, dass das System nicht ausgibt, auf welche Weise der anonyme Platzhalter belegt wurde, und zum anderen, dass mehrere Vorkommen des anonymen Platzhalters ihre Belegung nicht teilen (so wie "normale" Platzhalter es tun), also verschieden instantiiert werden können. Das System ersetzt jedes einzelne Auftreten von **[\_]** durch einen eigenständigen Platzhalter der Bauart **[\_GNummer]**. Das erste Auftreten von **[\_]** könnte beispielsweise durch **[\_G163]** ersetzt werden, das zweite durch **[\_G164]** usw.

```
vater(_,X), mutter(_,X).
% wird vom System umgeformt in
vater(_G768,X), mutter(_G769,X).
```

% Oben treten drei verschiedene Platzhalter auf: \_G768, X, \_G769

```
?- vater(bernd,_). % Ist bernd Vater?
Yes
```

### 1.7.2 Die Ausgabe von Platzhaltern unterdrücken

\*[Ausgabe von Platzhaltern unterdrücken]

?[Platzhalter, anonymer Platzhalter]

Platzhalter, die mit dem Zeichen **[\_]** beginnen, werden bei der Ausgabe von Ergebnissen nicht berücksichtigt. Wann immer man an den Belegungen bestimmter Platzhalter nicht interessiert ist, stellt man ihnen einen Unterstrich (**[\_]**) voran. Solche Platzhalter verhalten sich abgesehen von der unterdrückten Ausgabe wie "normale" Platzhalter. Beispiel:

```
?- vater(_Vater,Kind). % _Vater wird nicht ausgegeben
Kind = jens
Yes
```

Achtung: Die Platzhalter **[P]** und **[\_P]** sind voneinander verschieden! Alle Auftreten von **[P]** in einer Aussage haben dieselbe Belegung. Alle Auftreten von **[\_P]**

in einer Aussage haben ebenfalls dieselbe Belegung. Die Belegung von **P** kann sich jedoch von der Belegung **P**'s unterscheiden!

Die weiter oben gemachte Aussage, dass alle Individuen Bananen mögen, **mag(X,bananen).**, führt bei der Übersetzung des Programms zu folgender Warnung:

```
?- ['erstefakten.pl'].
[WARNING: (/home/ptschorn/erstefakten:18)
    Singleton variables: X]
erstefakten.pl compiled, 0.00 sec, 2,080 bytes.
```

Yes

Prolog hat in der betroffenen Aussage nur ein Auftreten des Platzhalters **X** gefunden. Im Allgemeinen treten Platzhalter in Regeln paarweise auf, einmal im Kopf der Regel und einmal im Rumpf, so dass Informationen durch Regeln 'fließen' können. Tritt ein Platzhalter nur einmal auf, könnte das daran liegen, dass sein Partner falsch geschrieben worden ist, der Informationsfluss würde zerstört werden. Ohne eine Warnung seitens des Prologsystems wäre es extrem schwierig, solchen Fehlern auf die Spur zu kommen.

In unserem Beispiel ist das einzelne Auftreten des Platzhalters dagegen beabsichtigt. Unter Verwendung des anonymen Platzhalters bzw. eines Platzhalters mit vorangestelltem Unterstrich kann die Warnung des Systems unterdrückt werden.

```
mag(_,bananen). % Wer/was immer _ ist, er/sie/es mag Bananen.

% auch möglich :
mag(_X,bananen). % Wer/was immer _X ist, er/sie/es mag Bananen.
```

## 1.8 Kompliziertere Aussagen (Regeln)

Bisher haben wir einige einfache Aussagen über unsere Familie getroffen. Wir haben einigen Individuen die Eigenschaft gegeben, weiblich oder männlich zu sein, und zwei **mutter**- sowie eine **vater**-Relation aufgeschrieben. Auf diesen einfachen Aussagen aufbauend können wir kompliziertere Aussagen, so genannte *Regeln*, formulieren. Dabei werden Platzhalter eine wichtige Rolle spielen. Während Fakten die Eigenschaften von und Beziehungen zwischen Individuen (oder Dingen) charakterisieren, beschreiben Regeln Beziehungen zwischen Aussagen.

\*[Regeln]  
[Fakten, Platzhalter, gemeinsame Platzhalter, undverknüpfte Aussagen]

### 1.8.1 Eine **eltern**-Regel

Als Erstes wollen wir unser bisheriges Programm um eine **eltern**-Regel erweitern, die die Beziehung zwischen einem Vater, einer Mutter (also den Eltern) und einem Kind beschreibt. Wir möchten etwa folgende Anfragen stellen können:

```
% Sind bernd und maria die eltern von jens?
1 ?- eltern(bernd, maria, jens).
Yes
```

```
% Welche(s) Kind(er) haben bernd und maria?
```

```
2 ?- eltern(bernd, maria, Kind).
```

```
Kind = jens;
```

```
No
```

```
% Wer sind die eltern von jens?
```

```
3 ?- eltern(Vater, Mutter, jens).
```

```
Vater = bernd
```

```
Mutter = maria
```

```
Yes
```

Eigentlich ist es ganz einfach: Die Aussage `eltern(Vater, Mutter, Kind).`

ist *wahr*, wenn folgende beiden Aussagen gelten: `vater(Vater, Kind).` und `mutter(Mutter, Kind).`. `Vater`, `Mutter` und `Kind` sind Platzhalter. In Prolog geschrieben lautet diese Regel:

```
eltern(Vater, Mutter, Kind) :- vater(Vater, Kind),
                                mutter(Mutter, Kind).
```

### 1.8.2 Aufbau von Regeln

#### Syntax

\*[Regelkopf, Regelrumpf]

Die obige Regel wird durch das Zeichen `:-` in *Kopf* und *Rumpf* geteilt. Der Kopf ist wie eine einfache Aussage aufgebaut, es folgt jedoch kein Punkt, sondern das so genannte *Implikationszeichen*, an das sich der Rumpf der Regel anschließt. Den Rumpf kann man als eine Aneinanderreihung von weiteren Aussagen verstehen. Diese werden durch Kommata getrennt, nur die letzte Aussage wird mit einem Punkt abgeschlossen.

### 1.8.3 Verhalten von Platzhaltern in Regeln

\*[Verhalten von Platzhaltern in Regeln]

?[Platzhalter, Regeln, Regelkopf, Regelrumpf]

Der Rumpf einer Regel entspricht einer bzw. mehreren undverknüpften Anfragen, wie wir sie schon kennen gelernt haben. Gleiche Platzhalter teilen ihren Wert in der gesamten Regel (also in Kopf und Rumpf), nicht jedoch außerhalb der Regel. Der Gültigkeitsbereich der Platzhalter erstreckt sich über die Regel. Die gleichen Platzhalter dürfen also in verschiedenen Regeln auftreten, ohne dass Namenskonflikte entstehen.

### 1.8.4 Bedeutung von Regeln

\*[Bedeutung von Regeln]

?[Regeln, Regelkopf, Regelrumpf]

Der Kopf einer Regel lässt sich beweisen, wenn sich alle Aussagen des Rumpfes beweisen lassen. Das Zeichen `:-` ließe sich logisch also so interpretieren: "Der linke Teil ist wahr, wenn der gesamte rechte Teil wahr ist." Ausgesprochen wird `:-` häufig als "falls". Von der Form her soll `:-` an  $\leftarrow$  aus der Prädikatenlogik erinnern.

Man spricht davon, dass die Aussagen im Rumpf der Regel *undverknüpft* sind. Die prozedurale Bedeutung einer Regel lautet: "Um den Kopf der Regel zu beweisen, muss der Rumpf bewiesen werden."

### 1.8.5 Was passiert, wenn eine Anfrage gestellt wird?

Wir haben die obige Regel in den Quelltext eingefügt und das Programm erneut übersetzt. Die Regel wurde mit Hilfe von Platzhaltern formuliert, wir können sie daher als eine Art *Schablone* betrachten. Hier noch einmal die Regel:

```
eltern(Vater, Mutter, Kind) :- vater(Vater, Kind),
                                mutter(Mutter, Kind).
```

\*[Arbeiten von Anfragen]  
?Fakten, Regeln, Anfragen, (gemeinsame)  
Platzhalter, Atome]

Was passiert, wenn wir die Anfrage `eltern(bernd, maria, jens).` stellen?

- Prolog erkennt, dass `eltern(bernd, maria, jens).` eine Aussage ist, die in die Schablone `eltern(Vater, Mutter, Kind) :- ...` passt. In diesem konkreten Fall funktioniert das, weil
  - der Funktor der Aussage und der Funktor der Regel übereinstimmen (beide Male `eltern`).
  - der Platzhalter `Vater` das Atom `bernd` aufnehmen kann
  - der Platzhalter `Mutter` das Atom `maria` aufnehmen kann
  - der Platzhalter `Kind` das Atom `jens` aufnehmen kann

Man spricht davon, dass die Anfrage und der Kopf der Regel *matchen*.

- Da die Anfrage und der Regelkopf matchen, belegt Prolog die Platzhalter, die in der Regel auftreten, mit den entsprechenden Argumenten aus der Anfrage. Dabei werden *alle* Vorkommen eines Platzhalters auf dieselbe Art und Weise instantiiert. Dadurch gelangen wir zu der folgenden *Instanz* der `eltern`-Regel:

<code>eltern(bernd, maria, jens).</code>	% Anfrage
<code>eltern(Vater, Mutter, Kind)</code>	% Kopf der Regel

% Platzhalter nehmen die Argumente der Anfrage auf.  
% Resultierende Instanz der Regel:

```
eltern(bernd, maria, jens) :- vater(bernd, jens),
                                mutter(maria, jens).
```

- Wir erinnern uns an die Bedeutung von Regeln: "Der linke Teil ist wahr, wenn der gesamte rechte Teil wahr ist." Der linke Teil der eben entstandenen Regelinstanz entspricht genau unserer ursprünglichen Anfrage. `bernd` und `maria` sind genau dann die Eltern von `jens`, wenn Prolog die rechte Seite der Regelinstanz, also die beiden Aussagen `vater(bernd, jens).` und `mutter(maria, jens).` beweisen kann. Man könnte sagen, dass die Anfrage `eltern(bernd, maria, jens).` zwei weitere Anfragen (nämlich `vater(bernd, jens).` und `mutter(maria, jens.).`) erzeugt hat.
- Prolog versucht nun, die beiden neuen Anfragen zu beweisen. Das gelingt in beiden Fällen, weil sowohl `vater(bernd, jens).` als auch `mutter(maria, jens.).` als Fakten in der Wissensbasis stehen.

- Der Rumpf der Regelinstanz ist also wahr. Demnach ist auch der Kopf der Regelinstanz wahr. Damit wurde unsere Anfrage bewiesen und das System antwortet mit Yes.

Analog zum obigen Beispiel wollen wir die Anfrage `eltern(V, M, jens).` betrachten.

- `eltern(V, M, jens).` matcht mit dem Regelkopf `eltern(Vater, Mutter, Kind) :- ...`
- Die Platzhalter in der Regel werden der Anfrage entsprechend gefüllt, es entsteht folgende Instanz der Regel:

```
eltern(V, M, jens) :- vater(V, jens),
                     mutter(M, jens).
```

- Das System versucht, den Rumpf der Regelinstanz zu beweisen.
- Der Beweis gelingt, wenn das System die Platzhalter folgendermaßen instantiiert:

```
V = bernd
M = maria
```

## 1.9 Redundanz

Die gleichen Ergebnisse hätten wir erzielen können, indem wir statt der o.a. Regel folgenden Fakt in die Wissensbasis eingetragen hätten: `eltern(bernd, maria, jens).` Damit würde die Wissensbasis dieselbe Information auf mehrere Weisen gleichzeitig repräsentieren - sie wäre *redundant*. Wenn wir uns vorstellen, wir hätten 1000 `vater`- und 1000 entsprechende `mutter`-Relationen in unserer Wissensbasis, könnten wir die `eltern`-Relationen explizit als 1000 Fakten oder aber als *eine* Regel aufschreiben.

Ein Problem mit Redundanz ist, dass durch Fehler Inkonsistenzen entstehen können. Wenn alle Fakten auf zwei Arten repräsentiert werden und sich nun die zwei Repräsentationen desselben Faktes durch einen Fehler oder durch Absicht widersprechen, liegt eine Inkonsistenz vor.

Wir ziehen es also vor, das für eine Aufgabe nötige Wissen möglichst redundanzarm zu repräsentieren. Wir sind in Bezug auf den benötigten Speicherplatz effizient. Es sind aber auch Szenarien denkbar, in denen es auf die Ausführungs geschwindigkeit eines Programmes ankommt. Nehmen wir an, eine sehr häufig gebrauchte problemspezifische Regel wäre in Bezug auf ihre Laufzeit 100 mal *teurer* als das Auffinden eines Faktes, so würde es sehr wohl Sinn machen, das Wissen in Form von Fakten bereitzustellen. Eine gute Repräsentation ist oft schon eine halbe Lösung. An dieser Stelle sei darauf hingewiesen, dass wir unsere kleine Beispieldfamilie auch auf beliebig viele andere Arten repräsentieren könnten. Eine Möglichkeit wäre z.B.:

```
% Fakten
```

```
% bernd und maria sind die eltern von jens
```

```
% bernd und maria sind die eltern von susanne
% ...
eltern(bernd, maria, jens).
eltern(bernd, maria, susanne).

...
% Regeln
vater(Vater, Kind)      :- eltern(Vater, _, Kind).
mutter(Mutter, Kind)    :- eltern(_, Mutter, Kind).
```

## 1.10 weitere Beispielregeln

### 1.10.1 **grossmutter** - alternative Regelrümpfe

Im folgenden Beispiel soll die bisherige Wissensbasis um eine **grossmutter**-Relation erweitert werden. Wir betrachten zuerst die Großmutter mütterlicherseits.

\*[alternative Regeln]  
Regeln, Aufbau von Regeln, mehrere Antworten, Platzhalter, Bedeutung von Regeln, Fakten]

```
grossmutter(Grossmutter, Enkel) :-
    mutter(Grossmutter, Mutter),
    mutter(Mutter, Enkel).
```

Das hieße also: Eine **Grossmutter** ist die **grossmutter** von einem **Enkel**, falls **Grossmutter** die **mutter** einer **Mutter** und **Mutter** die **mutter** von **Enkel** ist.

Die Definition der **Grossmutter** väterlicherseits sieht ganz analog aus:

```
grossmutter(Grossmutter, Enkel) :-
    mutter(Grossmutter, Vater),
    vater(Vater, Enkel).
```

Eine **Grossmutter** ist die **grossmutter** von einem **Enkel**, falls **Grossmutter** die **mutter** eines **Vater** und **Vater** der **vater** von **Enkel** ist.

Beide Regeln haben identische Köpfe. Eine **grossmutter** kann mütterlicher- oder väterlicherseits sein. Gibt es also mehrere Möglichkeiten, eine Relation auszudrücken, so schreibt man für jede gewünschte Möglichkeit eine eigene Regel. *Pragmatik*

Wir wollen einen zusätzlichen Fakt in die Wissensbasis aufnehmen:

```
mutter(erika, bernd). % in die Wissensbasis aufnehmen
```

Die Anfrage, welche **grossmutter**-Relationen in der Wissensbasis bestehen, liefert:

```
?- grossmutter(X,Y).
```

```
X = anke          % mütterlicherseits
Y = jens ;
```

```
X = erika        % väterlicherseits
```

```
X = jens ;
```

No

Prolog benutzt auf der Suche nach Lösungen beide **grossmutter**-Regeln.

### 1.10.2 **geschwister** - ein merkwürdiges Phänomen

Geschwister sind Kinder, die dieselben Eltern haben.

```
geschwister(G1, G2) :-  
    eltern(V, M, G1),  
    eltern(V, M, G2).
```

Die folgenden Fakten wollen wir dem bisherigen Programm hinzufügen:

```
frau(susanne).  
mutter(maria, susanne).  
vater(bernd, susanne).
```

Schließlich stellen wir die Frage nach allen bestehenden **geschwister**-Relationen:

```
?- geschwister(X,Y).
```

```
X = jens  
Y = jens ;
```

```
X = jens  
Y = susanne ;
```

```
X = susanne  
Y = jens ;
```

```
X = susanne  
Y = susanne ;
```

No

Offenbar schließt unsere Regel nicht aus, dass jemand sein eigenes Geschwister-Teil ist.

In dem bisherigen Beispielprogramm treten folgende Individuen auf: **maria**, **michaela**, **susanne**, **jens**, **bernd**, **anke** und **erika**. Wir können aufschreiben, dass diese Personen paarweise verschieden sind:

```
ungleich(maria, michaela).  
ungleich(maria, susanne).  
ungleich(maria, jens).  
ungleich(maria, bernd).  
ungleich(maria, anke).  
ungleich(maria, erika).
```

```

ungleich(michaela, susanne).
ungleich(michaela, jens).
ungleich(michaela, bernd).
ungleich(michaela, anke).
ungleich(michaela, erika).

ungleich(susanne, jens).
ungleich(susanne, bernd).
ungleich(susanne, anke).
ungleich(susanne, erika).

ungleich(jens, bernd).
ungleich(jens, anke).
ungleich(jens, erika).

ungleich(bernd, anke).
ungleich(bernd, erika).

ungleich(anke, erika).

```

Auf diesen Fakten aufbauend formulieren wir eine Relation **ist\_nicht(X,Y)**, die ausdrückt, dass X von Y verschieden ist:

```

ist_nicht(X,Y) :- ungleich(X,Y).
ist_nicht(X,Y) :- ungleich(Y,X).

```

Die erste Regel wird direkt auf **ungleich(X,Y)** abgebildet. Die zweite Regel vertauscht die Position der Argumente, um so die Symmetrie der Relation auszudrücken. Mit Hilfe der Relation **ist\_nicht** gelangen wir zu einer neuen Version der **geschwister**-Relation:

```

g2(G1, G2) :-
    eltern(V, M, G1),
    eltern(V, M, G2),
    ist_nicht(G1,G2).

```

Diese Regel verhält sich intuitiver als ihre Vorgängerin:

```

?- g2(X,Y).
X = jens
Y = susanne ;
X = susanne
Y = jens ;

```

No

Es ist möglich, dasselbe Ergebnis auch ohne die **ungleich**-Fakten und die **ist\_nicht**-Relation zu erzielen. Der obige Ansatz ist für den Einsatz in größeren Programmen mühselig.

### 1.10.3 Verwendung von `not`

\*[Negation, vage:eingeschränkt]

Es gibt in Prolog eine eingeschränkte Form von Negation, die wir später genauer beleuchten werden.

Wir einigen uns darauf, zwei Individuen als verschieden anzusehen, wenn ihre Bezeichnungen syntaktisch verschieden sind. Das Prologsystem kann Atome syntaktisch vergleichen. Zwei Atome matchen, wenn sie syntaktisch gleich sind. Wir übergeben zwei Atome als Argumente an den folgenden Fakt:

```
gleich(X,X). % gelingt wenn beide Argumente gleich sind, scheitert sonst
```

Sind sie verschieden, scheitert der Beweis des Faktes, da `X` nicht gleichzeitig zwei verschiedenen Werte haben kann.

Könnten wir den Wahrheitswert der `gleich`-Aussage umdrehen, hätten wir ohne viel Tipparbeit erreicht, was `ungleich` aus dem vorherigen Abschnitt leistet. Diese Funktionalität stellt Prolog mit `not` bereit. `not` gelingt, wenn der Versuch, das übergebene Argument zu beweisen scheitert. Gelingt der Beweis, so scheitert `not`. `geschwister` lässt sich also auch folgendermassen definieren:

```
geschwister(G1, G2) :-  
    eltern(V, M, G1),  
    eltern(V, M, G2),  
    not(gleich(G1, G2)).
```

Anmerkung: `not` entspricht nicht der logischen Negation! Mehr Informationen über diesen mystischen Ausspruch finden sich in den Kapiteln über die Beweisprozedur(5, 5.10) und die logische Fundierung (6). Außerdem wird statt `not` standartmäßig `\+` verwendet, das äquivalent ist. Dadurch soll vermieden werden, dass man sich in bezug auf Negation und das was Prolog in dieser Hinsicht zu bieten hat auf seine Intuition verlässt.

## 1.11 `vorfahr` – eine weitere Spielart von Regeln

Die Vorfahren eines Individuums können in zwei Gruppen eingeteilt werden: Zum einen gibt es die direkten (unmittelbaren) Vorfahren, also die Eltern eines Individuums. Zum anderen gibt es mittelbare Vorfahren, also Großeltern, Urgroßeltern, Ururgroßeltern ... eines Individuums. Ein Elternteil ist Mutter oder Vater eines Individuums. Wir definieren eine entsprechende Regel:

```
elternteil(A, B) :- % A ist elternteil von B, wenn  
    mutter(A, B). % A mutter von B ist  
  
elternteil(A, B) :- % A ist elternteil von B, wenn  
    vater(A, B). % A vater von B ist
```

?[Anfragen, Aussagen, Platzhalter]

Unter Verwendung dieser Regel können wir nach einzelnen direkten Vorfahren fragen:

```
?- elternteil(DirVorfahr, jens).
```

```
DirVorfahr = maria ;
```

```
DirVorfaehr = bernd ;
```

No

Als Regel:

```
% direkte Vorfahren
vorfaehr(A, B) :- elternteil(A, B).
```

```
% Anfrage
?- vorfaehr(X, jens).
```

```
X = maria ;
```

```
X = bernd ;
```

No

Die mittelbaren Vorfahren von **jens** sind die Vorfahren seiner direkten Vorfahren. Die mittelbaren Vorfahren können natürlich mehr oder weniger weit von **jens** entfernt sein (z.B.: Großmutter, Urgroßmutter, Ururgroßmutter ...). Stellen wir zunächst mal eine Anfrage nach den direkten Vorfahren seiner direkten Vorfahren. (Das sind die nahesten mittelbaren Vorfahren.)

?fundverknüpfte Anfragen

```
?- vorfaehr(Mittelbar,Direkt), vorfaehr(Direkt, jens).
```

```
Mittelbar = anke
Direkt = maria
```

Yes

Ein Vorfahr ist ein direkter oder mittelbarer Vorfahr eines Individuums. Die Regel für direkte Vorfahren haben wir schon formuliert. Ein mittelbarer Vorfahr eines Individuums ist ein Vorfahr (mittelbar oder direkt!) eines direkten Vorfahrens (also Elternteils) des Individuums.

```
% direkte Vorfahren
vorfaehr(A, B) :- elternteil(A, B).
```

```
% MV ist mittelbarer Vorfahr von I, wenn
```

```
vorfaehr(MV, I) :- elternteil(D,I), % D direkter Vorfahr von I ist und
                  vorfaehr(MV, D). % MV ein Vorfahr von D ist
```

Diese Regel wird als *rekursiv* bezeichnet, da in ihrem Rumpf eine **vorfaehr**-Relation steht, sie sich also “auf ihre eigene Definition stützt”. Dem Prologsystem stehen nun zwei Regeln zur Verfügung!

\*“rekursiv”, rekursive Regel

Durch das Zusammenspiel der beiden **vorfaehr**-Regeln lassen sich alle Vorfahren finden, die in der Wissensbasis eingetragen sind.

```
?- vorfaehr(X,jens).
```

```
X = maria ;
```

```
X = bernd ;
```

```
X = anke ;
```

```
X = erika ;
```

No

In unserem bisherigen Beispielprogramm sind die entferntesten Vorfahren schon die Großeltern. Die *Rekursion* findet in der zweiten Regel statt, da sie eine weitere **vorfahr**-Anfrage erzeugt. Die erste Regel veranlasst keine Rekursion, sie dient als Ende einer beliebig langen Kette von rekursiven Anfragen. Man nennt sie daher *Rekursionsabbruch*. Der Rekursion ist ein eigenes ausführliches Kapitel gewidmet. Trotzdem wollen wir uns anschauen, was im System passiert, wenn die obige Anfrage gestellt wird. (Das genauere Wie und Warum wird sich in späteren Kapiteln klären ...)

\*[Rekursion, Rekursionsabbruch]

### 1.11.1 Beobachtung der Anfrage **vorfahr(X, jens).**

Im Folgenden werden wir die im Verlauf des Beweises entstehenden Regelinstanzen betrachten. Dabei werden wir insofern schummeln, als das Prologsystem immer sofort die passenden Aussagen heranzieht. (Dass sich das System in Wirklichkeit oft irrt und wie es in einem solchen Fall reagiert, werden wir in einem späteren Kapitel erörtern.)

\*[page: Verlauf eines Beweises mit mehreren Lösungen]

Unsere bisherige Wissensbasis ist im Laufe des vorigen Abschnitts um folgende Regeln erweitert worden:

```
elternteil(A, B) :-          % A ist elternteil von B, wenn
    mutter(A, B).            % A mutter von B ist

elternteil(A, B) :-          % A ist elternteil von B, wenn
    vater(A, B).            % A vater von B ist

% vorfahr/2

% 1.
% A ist direkter Vorfahr von B,      wenn
vorfahr(A, B) :- elternteil(A, B). % A elternteil von B ist

% 2.
% MV ist mittelbarer Vorfahr von I,  wenn
vorfahr(MV, I) :- elternteil(D,I),  % D direkter Vorfahr von I ist
                 vorfahr(MV, D). % und MV ein Vorfahr von D ist

Wir stellen also die Anfrage vorfahr(X, jens).

?- vorfahr(X, jens).
```

X = maria

Gemäß der 1. **vorfahr**-Regel ist **maria** für **X** eingesetzt worden, da **maria** in **elternteil**-Relation zu **jens** steht. (**maria** ist **mutter** von **jens**.) Das System wartet nun auf weitere Anweisungen. Wir verlangen nach einer alternativen Lösung.

X = maria ;

X = bernd

Auf **bernd** stößt das System, da auch **bernd** in **elternteil**-Relation zu **jens** steht. Es wurde also zunächst eine Alternative für das Ergebnis der **elternteil**-Anfrage aus dem Rumpf der 1. **vorfahr**-Regel gefunden. Weitere Lösungen für diese *Unteranfrage* existieren nicht; damit sind auch alle alternativen Lösungen für diesen Aufruf der 1. Regel aufgezählt. (**jens** hat nur zwei **elternteile**.) Wir fordern die nächste Lösung an:

X = bernd ;

X = anke

Diesmal hat das System die 2. **vorfahr**-Regel benutzt. Dabei hat es zunächst **maria** als **elternteil** (also als direkten **vorfahr**) von **jens** ermittelt und anschließend festgestellt, dass **anke** in **vorfahr**-Relation zu **maria** steht. Das hat das System mit Hilfe der 1. **vorfahr**-Regel herausgefunden. Damit ist **anke** mittelbarer **vorfahr** von **jens**.

X = anke ;

X = erika

**erika** ist eine weitere Lösung, die (anhand der zweiten Regel) gefunden wird, wenn das System **bernd** als direkten Vorfahr von **jens** auswählt und danach feststellt, dass **erika** in **vorfahr**-Relation zu **bernd** steht.

An dieser Stelle wollen wir die Regelinstanzen *für den aktuellen Fall* anschauen:

- Die Anfrage lautete: **vorfahr(X, jens).**
- Sie matcht mit dem Kopf beider **vorfahr**-Regeln. Aufgrund des bisherigen Verlaufs des Beweises wählt das Prologsystem die 2. Regel. Folgende Regelinstantz entsteht:

```
% Anfrage:  
vorfahr(X, jens).
```

```
% 2. Regel:  
% MV ist mittelbarer Vorfahr von I, wenn
```

```
vorfahr(MV, I) :- elternteil(D,I),    % D direkter Vorfahr von I ist
                  vorfahr(MV, D).    % und MV ein Vorfahr von D ist
```

```
% resultierende Regelinstanz:
% X ist mittelbarer Vorfahr von jens,      wenn
vorfahr(X, jens) :- elternteil(D,jens),    % D direkter Vorfahr von jens
                  vorfahr(X, D).    % und X ein Vorfahr von D ist
```

% Anmerkung: MV muss nicht in X umbenannt werden, da uninstantiiert. Ich finde das aber schöner.

- Zwei Unteranfragen müssen nun bewiesen werden. Die erste lautet: `elternteil(D,jens),`.

```
% Anfrage
elternteil(D,jens),
```

```
% passende Regel für 1. Teilanfrage
elternteil(A, B) :-          % A ist elternteil von B, wenn
                  vater(A, B).    % A vater von B ist
```

```
% resultierende Instanz
elternteil(D, jens) :-          % D ist elternteil von jens, wenn
                  vater(D, jens).    % D vater von jens ist
```

- `elternteil(D, jens)` gilt also, wenn `vater(D, jens).` gilt. Aus diesem Grund können wir die Regelinstanz der ursprünglichen Anfrage abändern:

?[Semantik von Regeln]

```
% bisherige Regelinstanz der ursprünglichen Anfrage:
vorfahr(X, jens) :- elternteil(D,jens),
                  vorfahr(X, D).
```

```
% resultierende Regelinstanz der ursprünglichen Anfrage:
vorfahr(X, jens) :- vater(D, jens),
                  vorfahr(X, D).
```

- `vater(D, jens),` lässt sich aus der Wissensbasis ableiten, wenn `D` das Atom `bernd` zugewiesen bekommt. Die neue Instanz stimmt mit dem Fakt `vater(bernd, jens).` überein. Der Platzhalter `[D]` hat nun also einen Wert, und sofort können wir alle Auftreten von `[D]` in der Regelinstanz der ursprünglichen Anfrage auf den neuen Stand bringen:

```
% resultierende Regelinstanz der ursprünglichen Anfrage:
vorfahr(X, jens) :- vater(bernd, jens),
                  vorfahr(X, bernd).
```

- Die erste Unteranfrage ist also bewiesen worden. Die zweite Unteranfrage lautet (nachdem `[D]` belegt wurde): `vorfahr(X, bernd).` Wir betrachten nun die daraus entstehende Instanz der `vorfahr`-Regel:

```
% 2. Unteranfrage
vorfahr(X, bernd).

% passende Regel für 2. Teilanfrage
vorfahr(A, B) :- elternteil(A, B).

% resultierende Instanz
vorfahr(X, bernd) :- elternteil(X, bernd).
```

- Damit erhalten wir eine neue Version der Regelinstanz der ursprünglichen Anfrage:

```
% resultierende Regelinstanz der ursprünglichen Anfrage:
vorfahr(X, jens) :- vater(bernd, jens),
                  elternteil(X, bernd).
```

(Diese Ersetzung ergibt sich aus: `vorfahr(X, bernd) :- elternteil(X, bernd).`)

- Nun ist also `elternteil(X, bernd).` zu beweisen.

```
% die Unteranfrage
elternteil(X, bernd).
```

```
% passende Regel
elternteil(A, B) :-          % A ist elternteil von B, wenn
                  mutter(A, B).    % A mutter von B ist
```

```
% sich ergebende Regelinstanz
elternteil(X, bernd) :-          % A ist elternteil von X, wenn
                  mutter(X, bernd).
```

- Die aktualisierte Version der Regelinstanz der ursprünglichen Anfrage:

```
% resultierende Regelinstanz der ursprünglichen Anfrage:
vorfahr(X, jens) :- vater(bernd, jens),
                  mutter(X, bernd).
```

Da `elternteil(X, bernd) :- mutter(X, bernd).` gilt, durfte die obige Ersetzung durchgeführt werden.

- Schließlich muss `mutter(X, bernd).` bewiesen werden. Dies gelingt dem System, wenn es den Platzhalter `X` mit dem Atom `erika` instantiiert. (Dann stimmt die Anfrage mit dem entsprechenden in der Wissensbasis eingetragenen Fakt `mutter(erika, bernd).` überein.) Nun können wir alle Vorkommen von `X` auf den neuesten Stand bringen:

```
% resultierende Regelinstanz der ursprünglichen Anfrage:
vorfahr(erika, jens) :- vater(bernd, jens),
                      mutter(erika, bernd).
```

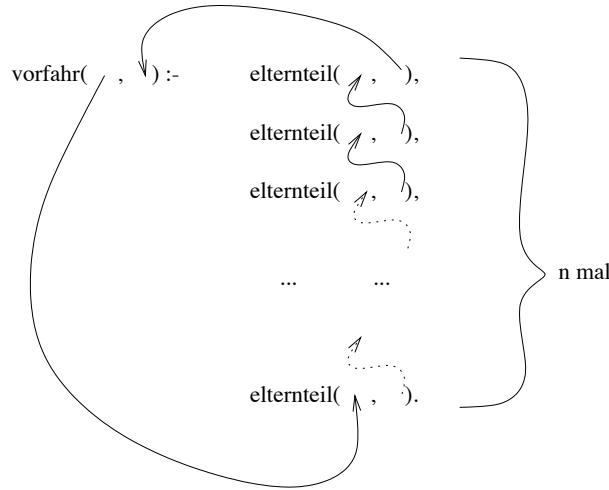


Abbildung 1.1: Vorfahr — rekursives Schema

- Das System hat sowohl `vater(bernd, jens)`, als auch `mutter(erika, bernd)` bewiesen, und damit ist der Beweis von `vorfahr(X, jens)` mit `X = erika` gelungen.

Zurück zur Beobachtung der Frage nach allen Vorfahren.

Das bisher letzte vom System gelieferte Ergebnis ist also:

`X = erika ;`

No

Weitere Lösungen lassen sich nicht aus der Wissensbasis ableiten. Würde man weitere Individuen als Väter und Mütter von `anke` und `erika` eintragen, so würden diese nach demselben Schema gefunden werden. Wäre z.B. `wolfgang` `vater` von `erika`, so wäre `wolfgang` (mittelbarer) `vorfahr` von `jens`, da

- `bernd` (direkter) `vorfahr` von `jens` ist,
- `erika` (direkter) `vorfahr` von `bernd` und damit (mittelbarer) `vorfahr` von `jens` ist,
- `wolfgang` (direkter) `vorfahr` von `erika` und damit (mittelbarer) `vorfahr` von `bernd` und damit (mittelbarer) `vorfahr` von `jens` ist.

Dieses Schema wird in Abbildung ?? verdeutlicht:

Mit den beiden gegebenen `vorfahr`-Regeln kann das Prologsystem dieses Schema ausfüllen,  $n$  kann dabei beliebig groß werden. Die Eigenschaft, dass  $n$  beliebig groß werden kann, ist nur durch eine rekursive Definition zu erreichen. Wollte man ohne Rekursion auskommen, müsste man  $n$  einzelne Regeln notieren und  $n$  in seiner Größe einschränken. Rekursion ist also ein mächtiges und elegantes Konzept.

Ein beliebtes Mißverständnis ist es, Rekursion als “Schleife” statt als “Kopie” aufzufassen. Bei einem neuen Teilbeweis wird immer eine “frische” Kopie einer Regel benutzt.

Damit ist unser erster Ausflug in diese Gefilde beendet.

## 1.12 das vollständige Beispielprogramm

```

frau(maria).           % maria ist eine frau
frau(michaela).       % ...
frau(susanne).
frau(anke).
frau(erika).

mann(jens).           % jens ist ein mann
mann(bernd).          % ...

mutter(anke, maria).  % anke ist die mutter von maria
mutter(maria, jens).   % ... ...
mutter(erika, bernd).
mutter(maria, susanne).

vater(bernd, jens).    % bernd ist der vater von jens
vater(bernd, susanne). % ... ...

mag(_, bananen).      % alles und jeder mag bananen

% eltern/3
% Vater und Mutter sind die eltern von Kind, wenn
% Vater der vater und Mutter die mutter von Kind ist.

eltern(Vater, Mutter, Kind) :-
    vater(Vater, Kind),
    mutter(Mutter, Kind).

% grossmutter/2
%
% mütterlicherseits:
% Grossmutter ist die grossmutter von Kind, wenn
% Grossmutter die mutter von Mutter und Mutter die mutter
% von Kind ist.
%
% väterlicherseits:
% Grossmutter ist die grossmutter von Kind, wenn
% Grossmutter die mutter von Vater und Vater der vater
% von Kind ist.

grossmutter(Grossmutter, Kind) :-      % mütterlicherseits
    mutter(Grossmutter, Mutter),
    mutter(Mutter, Kind).

```

```

grossmutter(Grossmutter, Kind) :-      % väterlicherseits
    mutter(Grossmutter, Vater),
    vater(Vater, Kind).

% geschwister/2
% G1 und G2 sind Geschwister, wenn sie identische
% Eltern haben.

geschwister(G1, G2) :-
    eltern(V, M, G1),
    eltern(V, M, G2).

% g2/2
% G1 und G2 sind Geschwister, wenn sie identische
% Eltern haben und G1 nicht dieselbe Person wie G2 ist.

g2(G1, G2) :-
    eltern(V, M, G1),
    eltern(V, M, G2),
    ist_nicht(G1, G2).

% ist_nicht/2
% X ist nicht identisch mit Y, wenn
% X ungleich Y oder Y ungleich X ist

ist_nicht(X, Y) :- ungleich(X, Y).
ist_nicht(X, Y) :- ungleich(Y, X).

% ungleich/2
% die folgenden Individuen sind nicht identisch

ungleich(maria, michaela).
ungleich(maria, susanne).
ungleich(maria, jens).
ungleich(maria, bernd).
ungleich(maria, anke).
ungleich(maria, erika).
ungleich(michaela, susanne).
ungleich(michaela, jens).
ungleich(michaela, bernd).
ungleich(michaela, anke).
ungleich(michaela, erika).
ungleich(susanne, jens).
ungleich(susanne, bernd).
ungleich(susanne, anke).
ungleich(susanne, erika).
ungleich(jens, bernd).
ungleich(jens, anke).
ungleich(jens, erika).

```

```
ungleich(bernd, anke).
ungleich(bernd, erika).
ungleich(anke, erika).

% elternteil/2

elternteil(A, B) :-      % A ist elternteil von B, wenn
    mutter(A, B).          % A mutter von B ist

elternteil(A, B) :-      % A ist elternteil von B, wenn
    vater(A, B).          % A vater von B ist

% vorfahr/2

% 1.
% A ist direkter Vorfahr von B,      wenn
vorfahr(A, B) :- elternteil(A, B).    % A elternteil von B ist

% 2.
% MV ist mittelbarer Vorfahr von I,  wenn
vorfahr(MV, I) :- elternteil(D,I),   % D direkter Vorfahr von I ist
                 vorfahr(MV, D).    % und MV ein Vorfahr von D ist
```



# Kapitel 2

## Terme, Strukturen, Unifikation

### 2.1 Terme

In Prolog werden alle Arten von Daten in Form so genannter Terme abgelegt.  
Es gibt drei Arten von Termen:

- Konstanten ?[Konstanten, Atome, Integer]  
\*[einfache Terme]  
Beispiele: `[maria]`, `[abc]`, `[123]`
- Platzhalter ?[Platzhalter]  
Beispiele: `[X]`, `[Y]`, `[_G256]`
- Strukturen (zusammengesetzte Terme) ?[Funktor, Argumente]  
Diese Art von Termen werden wir im Folgenden kennen lernen.

Die erforderlichen Grundbegriffe wurden in Abschnitt 1.3 eingeführt.

### 2.2 Strukturen

Strukturen bestehen aus einem Funktor und aus den in Klammern eingeschlossenen Argumenten, die wir in diesem Kontext *Komponenten*<sup>1</sup> nennen werden. Zwischen Funktor und öffnender Klammer darf kein Leerzeichen stehen! Es muss mindestens eine Komponente in den Klammern stehen. Die Komponenten von Strukturen sind wiederum Terme, d.h. eine Komponente einer Struktur ist entweder eine Konstante, ein Platzhalter oder wiederum eine Struktur.

\*[Strukturen]  
\*[einfache Terme, Aussagen]  
*Syntax*

`funktor(Komponente1, Komponente2, ..., Komponenten)`

Strukturen werden auch als *komplexe Terme* bzw. als *zusammengesetzte Terme* bezeichnet. Sie eignen sich, um Informationen zusammenzufassen und zu ordnen. Wir werden nun Fälle betrachten, in denen Strukturen auftreten, deren Komponenten einfache Terme - also Konstanten und Platzhalter - sind.

---

<sup>1</sup>Argument und Komponente bezeichnen dieselbe Art von Baustein. Im Zusammenhang mit Strukturen ziehen wir der Klarheit halber den Begriff Komponente vor.

### 2.2.1 Strukturen mit einfachen Termen als Komponenten

Mit Strukturen können zusammengehörige Einzelinformationen übersichtlicher gegliedert werden. Vater und Mutter zusammen sind Eltern. Von dieser Erkenntnis beflügelt, stellen wir uns `paar(Vater, Mutter)` vor - eine Struktur mit dem Funktor `paar` und zwei Platzhaltern `Vater`, `Mutter` als Komponenten. Mit dieser Struktur haben wir eine Schablone für zusammengehörige Informationen. Die Platzhalter in der Schablone werden entweder wir (konkrete Anfragen) oder das Prologsystem instantiiieren (Anfragen mit Platzhaltern).

Unsere `paar`-Struktur wollen wir nun in die `eltern`-Regel aus dem bisherigen Beispielprogramm einbauen. Die alte Regel lautet:

```
% eltern/3
% Vater und Mutter sind die eltern von Kind, wenn
% Vater der vater und Mutter die mutter von Kind ist.
```

```
eltern(Vater, Mutter, Kind) :-
    vater(Vater, Kind),
    mutter(Mutter, Kind).
```

Wir wollen die ersten beiden Argumente der Regel (`Vater`, `Mutter`) durch unsere Struktur ersetzen. Die neue Version der Regel lautet:

```
% neue Version
% eltern/2
eltern( paar(Vater, Mutter), Kind) :-
    vater(Vater, Kind),
    mutter(Mutter, Kind).
```

Die neue Regel hat im Gegensatz zu ihrer Vorgängerin nur noch zwei Argumente. Beide Regeln können vom Prologsystem aufgrund ihrer Stelligkeit unterschieden werden. `eltern/2` und `eltern/3` sind für das System völlig verschieden. Es ist also nicht notwendig, die alte Regel zu löschen. Unsere erste Anfrage lautet: Sind `bernd` und `maria` die `eltern` von `susanne`?

```
?- eltern(paar(bernd,maria), susanne).
```

Yes

`bernd` und `maria` mussten wir dazu natürlich als Komponenten in unsere Struktur einsetzen. Die so entstandene Struktur, `paar(bernd,maria)`, haben wir als erstes Argument unserer Anfrage verwendet. Strukturen erstellt man auf denkbar einfache Weise: Man schreibt sie hin.

?[Stelligkeit]

\*[nochmal: Regeln mit verschiedener Stelligkeit sind unterschiedlich]

Pragmatik

\*[Bedeutung von Strukturen an Argumentstelle]

### 2.2.2 Bedeutung von Strukturen als Argumente von Aussagen

Damit die folgenden Ausführungen verwirrender werden, stellen wir uns für einen Moment vor, dass wir als Funktor für unsere `paar`-Struktur statt dem Atom `paar` das Atom `eltern` gewählt hätten:

```
% verwirrende Version
% eltern/2
eltern( eltern(Vater, Mutter), Kind) :-
    vater(Vater, Kind),
    mutter(Mutter, Kind).
```

Dass der Funktor der Aussage (= das äußere `eltern`) und der Funktor der Struktur (= das innere `eltern`) gleich heißen sieht verwirrend aus, spielt jedoch keine Rolle!

Aussagen und damit Prologprogramme sind nämlich letzten Endes auch (nur) Strukturen, die von dem Prologsystem in einer bestimmten Art und Weise interpretiert werden. Eine Eigenschaft des Interpreters ist, dass Strukturen, die an Argumentstelle von Aussagen stehen, als Daten angesehen werden<sup>2</sup>. D.h. es handelt sich nicht etwa um eingeschachtelte Aussagen, die bewiesen werden, sondern schlicht und einfach um strukturierte Daten.

\*[Programme und Daten sind Strukturen]

?frage: Programme werden interpretiert]

### 2.2.3 Verhalten von Platzhaltern in Strukturen

In Strukturen enthaltene Platzhalter verhalten sich genauso wie die Platzhalter, die wir bisher kennengelernt haben. Auch ihr Gültigkeitsbereich ist die Aussage, in der sie auftreten, nicht etwa nur die Struktur!

\*[Platzhalter in Strukturen]

### 2.2.4 ausführliche Beispielanfragen

Was passiert, wenn wir die Anfrage `eltern(paar(bernd, maria), susanne).` stellen?

- Prolog erkennt, dass `eltern(paar(bernd, maria), susanne).` eine Aussage ist, die in die Schablone `eltern(paar(Vater, Mutter), Kind) :- ...` passt. In diesem Fall funktioniert das, weil
  - der Funktor der Aussage und der Funktor der Regel übereinstimmen (beide Male `eltern`).
  - der Funktor der Struktur, die das erste Argument in der Anfrage ist, mit dem Funktor der Struktur übereinstimmt, die als erstes Argument in dem Kopf der Regel steht. Es handelt sich in beiden Fällen um `paar`.
  - die erste Komponente der `paar`-Struktur im Kopf der Regel, also der Platzhalter `Vater`, das Atom `bernd` aufnehmen kann, das an korrespondierender Stelle in der `paar`-Struktur aus der Anfrage steht.
  - der Platzhalter `Mutter` das Atom `maria` aufnehmen kann. Der Platzhalter ist die zweite Komponente der Struktur in der Regel, das Atom ist die zweite Komponente der Struktur aus der Anfrage.
  - der Platzhalter `Kind` das Atom `susanne` aufnehmen kann. `Kind` ist das zweite Argument der Regel, `susanne` das zweite Argument der Anfrage.

?{Argument, Komponente}

---

<sup>2</sup>vergleichbar mit gequoteten Ausdrücken in Lisp

- Da die Anfrage mit dem Kopf der Regel matcht, instantiiert Prolog die Platzhalter mit den Werten aus der Anfrage. Folgende Instanz der `eltern/2`-Regel entsteht:

```
eltern(paar(bernd, maria), susanne).          % Anfrage
eltern(paar(Vater, Mutter), Kind)             % Kopf der Regel
```

% Platzhalter nehmen die Argumente der Anfrage auf.  
% Resultierende Instanz der Regel:

```
eltern(paar(bernd, maria), susanne) :- vater(bernd, susanne),
                                         mutter(maria, susanne).
```

- Prolog muss nun den Rumpf der Regelinstanz beweisen. Das gelingt, da sowohl `vater(bernd, susanne)`. als auch `mutter(maria, susanne)`. aus der Wissensbasis abgeleitet werden können. (Sie sind Fakten.)
- Dadurch, dass der Rumpf bewiesen wurde, ist der Kopf der Regelinstanz bewiesen. Dieser entspricht der Anfrage, die wir gestellt haben. Das System antwortet mit Yes.

Als nächsten Fall betrachten wir die Anfrage `eltern(paar(V, M), jens)`.

- `eltern(paar(V, M), jens)`. passt in die Schablone `eltern(paar(Vater, Mutter), Kind) :- ..`
- Die Platzhalter werden instantiiert. Folgende Instanz der Regel resultiert daraus:

```
eltern(paar(V, M), jens) :- vater(V, jens),
                           mutter(M, jens).
```

- Das System versucht, den Rumpf der Regelinstanz zu beweisen.
- Der Beweis gelingt mit den folgenden Instantiierungen:

```
V = bernd
M = maria
```

## 2.2.5 Platzhalter durch Strukturen ersetzen

\*[Platzhalter mit Strukturen instantiiieren]

Die aktuelle `eltern`-Regel hat zwei Argumente. In der Anfrage `eltern(paar(V, M), jens)`. haben wir als erstes Argument eine Struktur mit zwei Platzhaltern als Komponenten eingesetzt. Stattdessen kann auch ein Platzhalter als erstes Argument eingesetzt werden:

```
?- eltern(E,jens).
```

```
E = paar(bernd, maria)
```

Yes

Die Ausgabe des Systems zeigt, dass Platzhalter auch mit Strukturen instantiiert werden können.

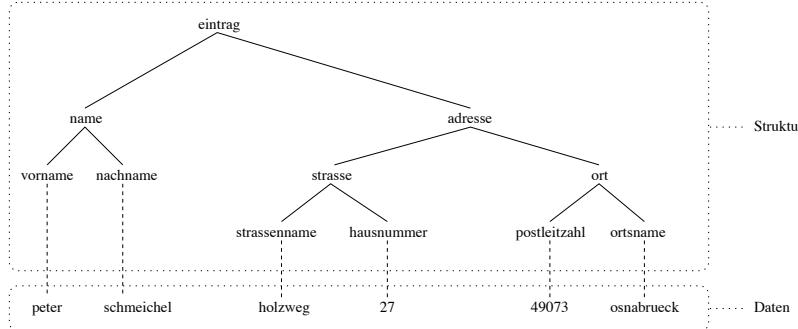
### 2.2.6 Strukturen mit Strukturen als Komponenten

Die Komponenten einer Struktur können wiederum Strukturen sein. Das werden wir ausnutzen, um eine einfache Adressdatenbank zu erstellen. Folgende Informationen sollen in diesem Beispiel relevant sein:

?[strukturen bisher]

- Ein Eintrag soll einen Namen und eine Adresse enthalten.
- Ein Name soll aus einem Vor- und einem Nachnamen bestehen.
- Eine Adresse soll eine Straße und einen Ort beinhalten.
- Eine Straße soll aus einem Straßennamen und einer Hausnummer aufgebaut sein.
- Ein Ort soll eine Postleitzahl und einen Ortsnamen umfassen.

Wir wissen nun also nicht nur, welche Daten eine Rolle spielen, sondern auch auf welche Weise sie strukturiert sind:



Wir werden den Datensatz aus dem Diagramm in einen Prolog-Fakt umsetzen. Der Fakt wird `eintrag` heißen.

`eintrag(...).`

Der `eintrag`-Fakt hat zwei Argumente: eine `name`- und eine `adresse`-Struktur.

`eintrag(name(...), adresse(...)).`

Wir betrachten zuerst die `name`-Struktur, sie hat die Komponenten `vorname` und `nachname`, die beide wiederum Strukturen sind. Es ist syntaktisch in Ordnung, nach runden Klammern Leerzeichen einzufügen. Damit können wir den Beispiefakt ein wenig lesbare machen.

*Syntax*

*Pragmatik*

\*[Fakten lesbare machen]

`eintrag( name( vorname(...), nachname(...) ), adresse(...) ).`

Die Struktur `vorname` wird das Atom `peter` als Komponente aufnehmen.

`name` wird `schmeichel` aufnehmen. Wenn man an den richtigen Stellen Leerzeichen und Zeilenumbrüche einfügt, kann man den Fakt optisch aufbessern (dariüber lässt sich streiten), ohne Syntaxfehler zu erzeugen. (Noch ist der Fakt jedoch nicht fertig!)

```

eintrag(
    name(
        vorname(peter),
        nachname(schmeichel)
    ),
    adresse(
        ...
    )
).

```

Wenden wir uns der **adresse**-Struktur zu: Sie hat die Komponenten **strasse** und **ort**. **strasse** wird **strassennamen(holzweg)** und **hausnummer(27)** aufnehmen. **ort** werden wir mit **postleitzahl(49073)** und **ortsname(osnabrueck)** füllen. Der fertige Datensatz sieht wie folgt aus:

```

eintrag(
    name(
        vorname(peter),
        nachname(schmeichel)
    ),
    adresse(
        strasse(
            strassenname(holzweg),
            hausnummer(27)
        ),
        ort(
            postleitzahl(49073),
            ortsname(osnabrueck)
        )
    )
).

```

Oder ohne zusätzliche Zeilenumbrüche: **eintrag( name( vorname(peter), nachname(schmeichel) ), adresse( strasse( strassenname(holzweg), hausnummer(27) ), ort( postleitzahl(49073), ortsname(osnabrueck) ) )**.

### 2.2.7 Beispielanfragen

Den eben erstellten Datensatz wollen wir als Grundlage für einige Anfragen gebrauchen. Die erste Anfrage lautet:

?- **eintrag(Name, Adresse).**

```

Name = name(vorname(peter), nachname(schmeichel))
Adresse = adresse(strasse(strassenname(holzweg), hausnummer(27)), ort(postleitzahl(49073), ortsname(osnabrueck)))

```

Yes

Wie wir bereits ausprobiert haben, können Platzhalter jede Art von Term aufnehmen. **X** wird mit der **name**-Struktur belegt, **Y** mit der **adresse**-Struktur.

Es ist sicher nicht das Ziel, von großen Strukturen erschlagen zu werden, sondern ganz gezielt auf bestimmte Komponenten zuzugreifen. Dazu muss allerdings die Form der Struktur bekannt sein.

Nun soll uns einmal nur die **strasse**-Komponente interessieren:

\*[Zugriff auf Strukturkomponenten]

?[Ausgabe von Platzhaltern unterdrücken]

```
?- eintrag(_N, adresse(S,_0)).
```

```
S = strasse(strassenname(holzweg), hausnummer(27))
```

Yes

Jetzt nur noch der Straßename:

?[anonyme Platzhalter]

```
?- eintrag(_, adresse( strasse( strassenname(SN),_),_)).
```

```
SN = holzweg
```

Yes

Zu den interessanten Komponenten müssen wir uns an den Funktoren entlanghangeln. Die uninteressanten Komponenten können wir von anonymen Platzhaltern verschlucken lassen. *Pragmatik*

Man könnte auf die Idee kommen, zu versuchen, in Anfragen anstelle von Funktoren Platzhalter zu verwenden. Funktoren sind keine Terme, können also nicht Platzhaltern zugewiesen werden. Der Versuch, es dennoch zu tun, endet in einer Fehlermeldung, die ein wenig irreführend ist.

```
% funktioniert:  
eintrag(X,Y).
```

```
X = ...  
Y = ...
```

Yes

```
% funktioniert nicht:  
?- FUNKTOR(X,Y).
```

```
[WARNING: Syntax error: Operator expected  
FUNKTOR(X,Y  
** here **  
) . ]
```

Es sollte deutlich geworden sein, wie man auf die Komponenten von Strukturen zugreifen kann. Anfragen lassen sich mit Regeln erleichtern:

```
% Beispielregel  
adresse(Wer, Wo) :- eintrag(name(vorname(Wer),_), Wo ).
```

```
% Wo wohnt peter?  
?- adresse(peter,Wo).
```

```
Wo = adresse(strasse(strassenname(holzweg), hausnummer(27)),
             ort(postleitzahl(49073), ortname(osnabrueck)))
```

Yes

Prolog ermöglicht es, direkt nach Funktor, Anzahl und bestimmten Komponenten einer Struktur zu fragen. Dazu dienen:

- `functor(Struktur, Funktor, Stelligkeit).`

```
% Beispiel:  
?- functor(s(k1,k2), Funktor, Stelligkeit).  
Funktör = s  
Stelligkeit = 2  
Yes
```

- `arg(Stelle, Struktur, X).`

```
% Beispiel:  
?- arg(2,s(k1,k2),X).  
X = k2  
Yes
```

Mit den bisher vorgestellten Mitteln ist es möglich, abstrakte Datentypen wie z.B. binäre Bäume und Listen zu erstellen. Darauf wird später eingegangen.

## 2.3 Unifikation

\*[Unifikation]  
?/Terme]

Die *Unifikation* dient dazu, zwei Terme "gleichzumachen". Sie ist eines der Schlüsselkonzepte in Prolog und wir haben sie informell schon ganz gut kennen gelernt. Der erste und einfachste Fall, in dem Unifikation auftrat, war die Anfrage `frau(maria).`. Sie wurde vom System mit Yes beantwortet, da der Fakt `frau(maria).` in der Wissensbasis eingetragen war. Das System sucht anhand des Funktors und der Stelligkeit einer Anfrage nach passenden Aussagen in der Wissensbasis.

Eine Aussage "passt" mit einer Anfrage zusammen, wenn Funktor und Stelligkeit von Anfrage und Aussage identisch sind und sich korrespondierende Argumente erfolgreich *unifizieren* lassen.

Funktör und Stelligkeit von Anfrage und Aussage stimmen im obigen Fall überein (`frau/1`). Nun muss das System die Argumente unifizieren, also so angeleichen, dass gegebene Informationen erhalten bleiben und die beiden Terme syntaktisch gleich werden. In diesem ersten Fall ist das sehr einfach. Das Argument der Anfrage stimmt bereits mit dem der Aussage überein. Es handelt sich beide Male um das Atom `maria`.

`maria` und `maria` sind gleich. Die Atome sind syntaktisch identisch. Die Unifikation gelingt, da beide Terme schon gleich sind.

Der nächste Fall, der uns begegnet ist, war die Anfrage `frau(X)`. Auch sie matcht mit dem Fakt `frau(maria)`. Hier muss das System die beiden Terme `X` und `maria` unifizieren. Das gelingt, indem das System `X` mit `maria` instantiiert. Platzhalter sind in der Lage, mit jeder Art von Term belegt zu werden. Wenn das System `X` mit `maria` belegt, erhalten wir wieder "`maria` gleich `maria`" und die Unifikation ist erfolgreich abgeschlossen. Verlangen wir nach weiteren Lösungen, sucht Prolog nach weiteren Aussagen, die mit der Anfrage matchen, und stößt so auf die Fakten `frau(michaela)` und `frau(susanne)`. `X` muss dann mit `michaela`, bzw. mit `susanne` unifiziert werden. Man spricht von *Substitutionen*: Wenn man `X` mit `michaela` substituiert, sind die beiden Terme `X` und `michaela` gleich. Mit der Substitution `X=susanne` kann man die Terme `X` und `susanne` unifizieren.

\*[Substitution]

Man kann die Unifikationsroutine von Prolog direkt ansprechen. Dazu dient folgende Anfrage: `= (Term1, Term2)`. Es ist üblich, eine bequemere Schreibweise zu verwenden: `Term1 = Term2`.<sup>3</sup> Der Funktor der Anfrage besteht aus dem Gleichheitszeichen (`=`). Zwei zu unifizierende Terme füllen die Argumente der Anfrage. Obige und ähnliche Beispiele lassen sich schnell überprüfen:

\*[direktes Anstoßen der Unifikation]

```
?- =(atom, atom).
```

Yes

```
?- =(hans, peter).
```

No

```
?- =(X, maria).
```

X = maria

Yes

```
?- =(maria, X).
```

X = maria

Yes

```
?- Hans = peter.      % ab jetzt bequeme Schreibweise
```

Hans = peter

Yes

```
?- 33 = Zahl.
```

Zahl = 33

Yes

Wenn versucht wird, zwei (uninstantiierte) Platzhalter miteinander zu unifizieren, substituiert das System beide durch einen neuen Platzhalter. Man kann das als Umbenennung von Platzhaltern auffassen.

\*[Umbenennen von Platzhaltern]

```
?- X = Y.
```

X = \_G130

Y = \_G130

<sup>3</sup>Unter welchen Bedingungen und aus welchen Gründen diese Schreibweise möglich ist, lässt sich in dem Abschnitt über Operatoren (7.1) erfahren. Beide Formen sind äquivalent.

Hier wurden **[X]** und **[Y]** in **[G130]** umbenannt und somit unifiziert. Das System merkt sich alle Substitutionen, die es vorgenommen hat. Daher ist es möglich, die Belegungen der ursprünglichen Platzhalter zu konstruieren und auszugeben. Wenn wir uns bisher angesehen haben, was beim Stellen bestimmter Anfragen geschieht und dabei Regelinstanzen erzeugt haben, die Platzhalter enthielten, haben wir die Platzhalter aus der Regel in die Platzhalter aus der Anfrage umbenannt. Das haben wir der Einfachheit halber getan. Das Prologsystem führt in Wirklichkeit einen neuen, bisher unbekannten Platzhalter ein und substituiert die beiden zu unifizierenden Platzhalter mit dem neuen. Die zugrundeliegende Idee bleibt dieselbe.

Eine Substitution wirkt sich auf alle Vorkommen eines Platzhalters (gleichzeitig) aus. Das ist bei verketteten Anfragen bzw. bei Regelinstanzen sehr wichtig.

Im nächsten Beispiel soll einer der Terme eine Struktur sein.

```
?- X = struktur(k1,k2,k3).
X = struktur(k1, k2, k3)
Yes
```

Auch diesen Fall haben wir schon kennen gelernt. Interessanter wird es, wenn zwei Strukturen unifiziert werden müssen.

```
% 1.
?- struktur(k1,k2,k3) = struktur(k1,k2,k3).
Yes

% 2.
?- struktur(A,k2,k3) = struktur(k1,k2,B).
A = k1
B = k3
Yes
```

Die Vorgehensweise ist dabei die folgende: Zunächst müssen die Funktoren der beiden Strukturen übereinstimmen. Ist das gegeben, müssen die Komponenten von links nach rechts unifiziert werden. Gelingt dies, ist die Unifikation der Terme gelungen. Im 1. Beispiel stimmen sowohl Funktor als auch Komponenten der beiden Strukturen syntaktisch überein. Die Unifikation gelingt. Es müssen keine Substitutionen durchgeführt werden. Im 2. Beispiel tauchen zwei Platzhalter als Komponenten auf. Die Funktoren stimmen überein, die Komponenten werden paarweise von links nach rechts unifiziert. Dies gelingt mit den Substitutionen **[A = k1]** und **[B = k3]**.

Nun werfen wir einen Blick auf Strukturen, die nicht unifiziert werden können:

```
% Funktoren matchen nicht
?- vorname(peter) = maerchengestalt(peter).
No

% Anzahl der Komponenten stimmt nicht überein
?- reihe(1,2,3,4) = reihe(1,2).
No

% Nach der ersten Substitution ist keine Unifikation
```

```
% mehr möglich. (X kann nicht 1 und 2 gleichzeitig sein.)
?- reihe(X,X) = reihe(1,2).
No
```

```
% s(X) und 3 sind nicht unifizierbar
?- test(1,2,s(X)) = test(1,2,3).
No
```

Mit der Unifikation lassen sich grundlegende Operationen zur Manipulation von Daten durchführen: Zuweisungen an Platzhalter, Parameterübergabe, Zugriff auf Datenelemente und schließlich auch Konstruktion von Datenstrukturen.

### 2.3.1 Zusammenfassung des Unifikationsverfahrens

Die Unifikation zweier Terme  $T_1$  und  $T_2$  läuft nach folgendem Schema:

- Sind  $T_1$  und  $T_2$  identische Konstanten, so gelingt die Unifikation.
- Sind  $T_1$  und  $T_2$  uninstantiierte Platzhalter, so werden beide Platzhalter durch einen neuen uninstantiierten Platzhalter substituiert. (Umbenennung)
- Ist  $T_1$  ein uninstantierter Platzhalter und  $T_2$  eine Konstante oder Struktur, so wird  $T_1$  durch  $T_2$  ersetzt.
- Ist  $T_2$  ein uninstantierter Platzhalter und  $T_1$  eine Konstante oder Struktur, so wird  $T_2$  durch  $T_1$  ersetzt.
- Instantiierte Platzhalter werden dem Typ ihres Wertes entsprechend behandelt.
- Ist  $T_1$  eine Struktur der Form  $f(X_1, \dots, X_n)$  und  $T_2$  eine Struktur der Form  $f(Y_1, \dots, Y_n)$ , so gelingt die Unifikation genau dann, wenn es gelingt, alle Paare  $X_i, Y_i$  zu unifizieren. (Rekursion!)
- Ansonsten scheitert die Unifikation.

### 2.3.2 Test auf Vorkommen

Das obige Schema hat einen kleinen Schönheitsfehler. Damit das Unifikationsverfahren korrekt ist, muss in dem Fall, dass  $T_1$  ein Platzhalter und  $T_2$  eine Struktur ist, sichergestellt werden, dass  $T_1$  nicht in  $T_2$  vorkommt. (Der symmetrische Fall gilt ebenfalls.) Diese (prinzipiell nötige) Überprüfung wird als *Occur-Check* bezeichnet. Aus Performance-Gründen verzichten die meisten Prologsysteme auf diesen Vorkommenstest. So auch SWI-Prolog:

\*[Occur-Check]

```
% X tritt in f(X) auf
?- X = f(X).
[FATAL ERROR:
  Could not allocate memory: Nicht genügend Hauptspeicher verfügbar]
```

Ohne Vorkommenstest wird  $\boxed{X}$  durch  $\boxed{f(X)}$  substituiert. Das Malheur geschieht bei dem Versuch, diese Substitution auszugeben. Da  $\boxed{X = f(X)}$  gilt, entsteht ein immer länger werdender Ausdruck der Form  $\boxed{f(f(f(f(\dots))))}$ , der eigentlich unendlich groß würde. Nach einigen Sekunden ist jedoch der gesamte verfügbare Hauptspeicher aufgebraucht.

Fazit: Wenn  $\boxed{X}$  mit etwas substituiert werden müsste, das  $\boxed{X}$  enthält und nicht damit identisch ist, gibt es Tränen. Konstruktionen, die auf  $\boxed{=(X, f(X))}.$ , o.ä. hinauslaufen, sollten vermieden werden. Viele Prologsysteme erlauben jedoch das Ein- und Ausschalten des Occur-Checks. Was SWI-Prolog angeht, so sei auf das Referenzhandbuch verwiesen.

# Kapitel 3

## rekursive Datenstrukturen

Dieses Kapitel bildet in Verbindung mit dem Kapitel über Rekursion einen Überblick über abstrakte Datentypen, so genannte ADTs. Im Folgenden werden wir die Datenstrukturen besprechen. Auf ihnen mögliche Operationen werden wir in dem Kapitel über Rekursion (4) kennen lernen.

### 3.1 binäre Bäume

Ein binärer Baum hat eine Wurzel, die ein Datum enthält. Ein binärer Baum kann genau 0, 1 oder 2 Unterbäume haben, diese werden oft als *Söhne* oder *Nachfolger* bezeichnet. Die Unterbäume sind wiederum binäre Bäume. Ein Baum mit 0 Nachfolgern wird *Blatt* genannt.

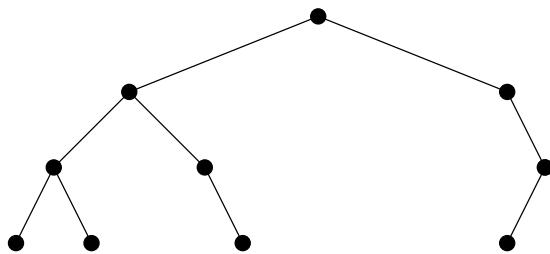
\*[Def: Binäre Blume]

\*[Sohn, Nachfolger, Unterbaum, Blatt]

Folgende Bausteine stehen also zur Verfügung:



Mit diesen Bausteinen können binäre Bäume beliebiger Größe konstruiert werden. Ein Beispiel:



Jeder “Knoten” enthält ein Datum und ggf. Unterbäume.

#### 3.1.1 Repräsentation in Prolog

Wie wir bereits wissen, werden Daten jeglicher Art in Prolog durch Terme repräsentiert. Um binäre Bäume zu notieren, benötigen wir Strukturen. Den

?[Terme]

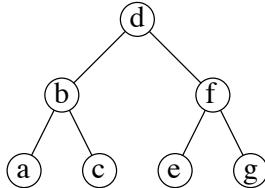
Funktor dieser Strukturen werden wir **baum** nennen. Eine **baum**-Struktur hat drei Komponenten — die erste enthält den linken Unterbaum, die zweite das Datum, die dritte den rechten Unterbaum. Das Atom **nil** steht für einen nicht vorhandenen Unterbaum<sup>1</sup>. In der Tabelle sind Schablonen für die Bausteine aufgeführt.

	Baustein	Repräsentation
I		baum(L, Datum, R)
II		baum(L, Datum, nil)
III		baum(nil, Datum, R)
IV		baum(nil, Datum, nil)

?[Strukturen, Platzhalter (also Terme:-)]

Die Platzhalter **L** und **R** stehen für den linken, bzw. rechten Teilbaum, also für weitere **baum**-Strukturen. Der Platzhalter **Datum** nimmt das eigentliche Datum auf. **nil** an der Position eines Unterbaums kennzeichnet das Fehlen desselben. (**nil** wurde willkürlich und in Anlehnung an Lisp gewählt.)

Mit diesen Schablonen werden wir folgenden Beispielbaum in Prolog formulieren. Im Adressenbeispiel aus Abschnitt 2.2.6 haben wir die Struktur von oben nach unten (top down) aufgebaut, d.h. haben uns von der obersten Ebene der Struktur aus Ebene für Ebene nach unten bis zu den eigentlichen Daten vorgearbeitet. In diesem Beispiel werden wir umgekehrt (bottom up) vorgehen, indem wir von den Daten ausgehen und diese immer weiter in Strukturen verpacken bis die Baumstruktur komplett ist<sup>2</sup>.



Wir beginnen bei Knoten **a**. Es handelt sich um ein Blatt. Nach der Schablone lautet die entsprechende Struktur: **baum(nil, a, nil)** – Knoten **c** wird analog dazu durch **baum(nil, c, nil)** repräsentiert.

Die Knoten **a** und **c** sind Teilbäume von Knoten **b**. Wir setzen sie also anstelle der Platzhalter in Schablone I ein und erhalten:

```

baum( baum(nil, a, nil), b, baum(nil, c, nil) )
      |           |           |
      linker Teilbaum   rechter Teilbaum
      |
      Wurzel
  
```

Mit den Knoten **e**, **g** und **f** verfahren wir analog und erhalten schließlich:

<sup>1</sup>Es sind auch andere Reihenfolgen wie z.B. Datum - linker Unterbaum - rechter Unterbaum denkbar.

<sup>2</sup>Bottom up und top down ergeben die gleiche Struktur.

```

baum( baum(nil, e, nil), f, baum(nil, g, nil) )
      |   |
linker Teilbaum   |   rechter Teilbaum
      |
Wurzel

```

Diese beiden Strukturen sind linker und rechter Teilbaum des Knotens d. Damit können wir die Struktur für den Beispielbaum komplett aufschreiben:

```

baum(
  baum( baum(nil, a, nil), b, baum(nil, c, nil) ), % linker Sohn
        d,                                     % Wurzel
  baum( baum(nil, e, nil), f, baum(nil, g, nil) ) % rechter Sohn
)

```

Oder baumähnlicher formatiert:

```

baum(
  baum(
    baum(                                %
      baum(nil, a, nil),      %     /L    )
      b,                      %     W    > linker Teilbaum
      baum(nil, c, nil)      %     / \R   )
  ),
  %
  %
d,                                %
% \                                %
% \                                %
  baum(                                %
    baum(                                %
      baum(nil, e, nil),      %     \ /L    )
      f,                      %     W    > rechter Teilbaum
      baum(nil, g, nil)      %     \R   )
  )
)

```

Es handelt sich hier wohlgernekt um eine Struktur, nicht um einen Fakt. Für spätere Anwendung heben wir die Struktur in einem Fakt auf:

```

beispielbaum(
  baum( baum( baum(nil, a, nil), b, baum(nil, c, nil) ),
        d,
        baum( baum(nil, e, nil), f, baum(nil, g, nil) )
  )
).

```

### 3.1.2 Beispiele für Zugriff

Mit diesem Fakt in der Wissensbasis können wir herumexperimentieren:

*[Unifikation]*

% X mit der Struktur instantiiieren

```
?- beispielbaum(X).
```

```
X = baum(baum(baum(nil, a, nil), b, baum(nil, c, nil)), d, baum(baum(nil, e, nil), f, baum(nil, g, nil)))
```

Yes

```
% X holen und mittels Unifikation in Wurzel und Teilbäume "zerlegen"
?- beispielbaum(X), X = baum(Links, Wurzel, Rechts).
```

```
X = baum(baum(baum(nil, a, nil), b, baum(nil, c, nil)), d, baum(baum(nil, e, nil), f, baum(nil, g, nil)))
```

```
Links = baum(baum(nil, a, nil), b, baum(nil, c, nil))
```

```
Wurzel = d
```

```
Rechts = baum(baum(nil, e, nil), f, baum(nil, g, nil))
```

Yes

```
% X holen und mittels Unifikation in Wurzel und Teilbäume "zerlegen",
% linken Teilbaum auf gleiche Art "zerlegen"
?- beispielbaum(baum(baum(L,W,R), Wurzel, Rechts)).
```

```
L = baum(nil, a, nil)
```

```
W = b
```

```
R = baum(nil, c, nil)
```

```
Wurzel = d
```

```
Rechts = baum(baum(nil, e, nil), f, baum(nil, g, nil))
```

Yes

```
% Dasselbe Spielchen noch eine Ebene tiefer getrieben:
```

```
?- beispielbaum(baum(baum(baum(L3, W3, R3), W2, R2), W1, R1)).
```

```
L3 = nil
```

```
W3 = a
```

```
R3 = nil
```

```
W2 = b
```

```
R2 = baum(nil, c, nil)
```

```
W1 = d
```

```
R1 = baum(baum(nil, e, nil), f, baum(nil, g, nil))
```

Yes

```
% Gleicher Beispiel ohne ‘integrierte Unifikation’ und
% mit unterdrückter Ausgabe:
?- beispielbaum(_X), _X = baum(_L1, _W1, _R1),
   _L1 = baum(_L2, _W2, _R2), _L2 = baum(_L3, W3, _R3).
```

W3 = a

Yes

In den letzten Beispielen haben wir den Inhalt eines Knotens (auch *Element* genannt) durch das mehrmalige Anwenden derselben Operation auf dem Ergebnis ihrer Vorgängeroperation “erreicht”.

## 3.2 Listen

Eine Liste ist eine Folge einer beliebigen Anzahl von “Elementen” mit festgelegter Reihenfolge. Die Liste ohne Elemente wird als die *leere Liste* bezeichnet.

\*[vage: Element]  
\*[Definition von Liste]

Beispillisten:

morgens, mittags, abends  
essen, arbeiten, schlafen, essen, essen, schlafen, träumen

Es gibt zwei Bausteine, mit denen beliebig lange Listen konstruiert werden können:

- die Leere Liste
- eine Struktur, die ein Element und einen weiteren Listenbaustein enthält.

Eine Liste ist also entweder die leere Liste, oder ein Element gefolgt von einer Liste.

### 3.2.1 Repräsentation in Prolog

Listen werden, wie alle Datenstrukturen, in Prolog in Form von Strukturen notiert.

?[Strukturen]

Als Funktor der Struktur benutzen wir den Punkt (`.`). Diese Wahl ist eine historisch bedingte Konvention. (Das Prologsystem ist durch die direkt folgende Runde Klammer in der Lage, den Punkt-Funktor von dem Punkt zu unterscheiden, der Aussagen abschließt.) Die erste Komponente der Struktur nimmt das Element auf. Das Element kann ein beliebiger Term sein. Die zweite Komponente der Struktur nimmt wiederum eine Liste auf, d.h. eine weitere Struktur, bzw. das Symbol für die leere Liste. Die leere Liste wird in Prolog durch das Symbol `[]` repräsentiert.

Die erste Komponente wird als *Kopf* der Liste bezeichnet, während die zwei-

\*[Kopf vs Restliste]

te Komponente *Restliste* genannt wird. Der Kopf einer Liste ist ein Element (also ein beliebiger Term), die Restliste ist wiederum eine Liste. Entspricht die Restliste einer Liste der leeren Liste, so wurde das Ende der Liste erreicht. Die leere Liste kann nicht mehr in Kopf und Restliste geteilt werden. Mit der leeren Liste wird das Ende der Folge von Elementen gekennzeichnet. Die Bausteine sehen also so aus:

	Baustein	Repräsentation
I	leere Liste	$[]$
II	Kopf/Restliste-Paar	$.(\text{Kopf}, \text{Restliste})$

\*[Funktorschreibweise]

Beispiel: die Liste mit den Elementen  $e_1, e_2, e_3, e_4, e_5$  lässt sich demnach wie folgt aufschreiben:

$$. (e_1, . (e_2, . (e_3, . (e_4, . (e_5, [])))) )$$

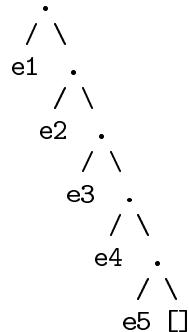
Dabei wurde folgendermaßen vorgegangen:

- Die Liste  $e_1, e_2, e_3, e_4, e_5$  erhält man, indem man Schablone II aus der obigen Tabelle füllt.  $e_1$  ist der Kopf,  $e_2, e_3, e_4, e_5$  die Restliste.
- Diese Restliste lässt sich mit Hilfe von II notieren:  $e_2$  ist der Kopf,  $e_3, e_4, e_5$  die Restliste.
- Diese Restliste lässt sich wieder mit II aufschreiben:  $e_3$  ist der Kopf,  $e_4, e_5$  die Restliste.
- Diese Restliste lässt sich wieder mit II aufschreiben:  $e_4$  ist der Kopf,  $e_5$  die Restliste.
- Diese Restliste lässt sich wieder mit II aufschreiben:  $e_5$  ist der Kopf, und die Restliste entspricht der leeren Liste.
- Diese Restliste notieren wir gemäß I als  $[]$ .

### 3.2.2 grafische Darstellung

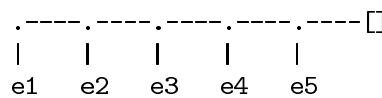
\*[grafische Darstellung 1]

Die so entstandene Struktur kann baumartig dargestellt werden:



\*[Weindigramm]

Leicht deformiert wird die Grafik anschaulicher:



Diese Art der Darstellung ist sehr beliebt und wird “Weindiagramm” genannt, weil die Elemente der Liste (eine gewisse Menge Fantasie vorausgesetzt) wie Weinreben herunterhängen.

### 3.2.3 verschiedene Notationen

Neben der *Funktorschreibweise*, die wir oben kennengelernt haben, existieren noch zwei weitere syntaktische Formen für Listen: Die *Cons-Paar-Notation* und die *Listennotation*. Sie dienen dazu, den Umgang mit Listen zu erleichtern.

Die Cons-Paar-Notation<sup>3</sup> ergibt sich durch einfache Änderungen aus der Funktornotation. Der Punkt und die runde öffnende Klammer werden durch die eckige öffnende Klammer ersetzt. Die schließende runde Klammer wird ebenfalls durch ihr eckiges Gegenstück ersetzt. Das Komma wird durch das so genannte Pipe-Symbol (‘|’) ausgetauscht. Kurz und bündig:

?[Funktorschreibweise]

\*[Cons-Paar-Notation]

.(Kopf, Rest)	wird zu	[Kopf   Rest]
□	bleibt	□

Unsere Beispielliste `.(e1, .(e2, .(e3, .(e4, .(e5, []))))` lautet in der Cons-Paar-Notation `[e1| [e2| [e3| [e4| [e5| []]]]]]`.

Die Listennotation stellt eine Vereinfachung der Cons-Paar-Notation dar. Aus `[e1| [e2| [e3| [e4| [e5| []]]]]` wird `[e1, e2, e3, e4, e5]`. Die Elemente der Liste werden hier durch Kommata getrennt. Die gesamte Liste wird von eckigen Klammern eingeschlossen. Die leere Liste wird nicht mehr explizit als abschließendes Element angehängt. Auch in der Listennotation spielt die Bauart “Kopf, Restliste” eine Rolle. Eine Restliste kann auch hier mit Hilfe des Pipe-Symbols kenntlich gemacht werden. Innerhalb einer Listenebene darf nur ein Pipe-Symbol verwendet werden und es muss eine Liste (in einer der drei Syntaxvarianten) folgen. Vor dem Pipe-Symbol darf eine beliebige Folge von durch Kommata getrennten Elementen stehen.

\*[Listennotation]

Diese drei Notationen werden vom Prologsystem absolut gleichwertig behandelt. Für die Begriffe “Kopf” und “Restliste” gibt es eine Reihe von üblichen Bezeichnungen: `[Kopf | Rumpf]`, `[Head | Tail]`, `[H | T]`, `[Kopf | Schwanz]`, `[Erstes | Rest]`, `[E | R]` und Ähnliches. Die Bezeichnungen “Kopf” und “Rumpf” tauchen auch im Zusammenhang mit Regeln auf, Regelkopf, Regelrumpf und Listenkopf, Listenrumpf lassen sich jedoch nur schwer verwechseln.

\*[andere Begriffe für Kopf und Restliste]

### 3.2.4 Gegenüberstellung der Notationen

Es folgt eine kleine Gegenüberstellung der Schreibweisen:

Funktornotation	Cons-Paar-Notation	Listennotation
□	□	□
.(E, R)	[E R]	[E R]
.(a, R)	[a R]	[a R]
.(a, .(b, R))	[a [b R]]	[a, b R]
.(a, .(b, .(c, R)))	[a [b [c R]]]	[a, b, c R]

<sup>3</sup>Der Name wurde in Anlehnung an Cons-Zellen in Lisp gewählt. car entspricht dem Kopf, cdr dem Rest einer Liste.

Da die Notationen zueinander kompatibel sind, können sie sogar gemischt werden, wozu ich nicht rate.

`[1,2,3,4| .(5, .(6, [7|[]]))]`

ist gleichwertig zu

`[1, 2, 3, 4, 5, 6, 7]`

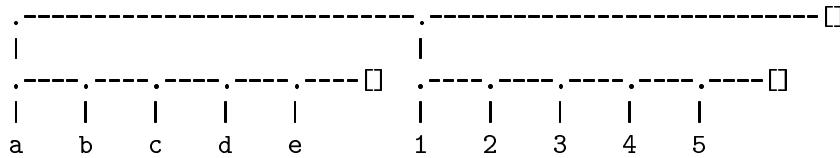
Die Listennotation ist am gebräuchlichsten.

### 3.2.5 Listen von Listen

Es ist häufig der Fall, dass die Elemente einer Liste Listen sind. Man spricht von einer solchen Konstruktion als "Liste von Listen".

Die zweielementige Liste `[[a,b,c,d,e], [1,2,3,4,5]]` besteht aus den Unterlisten `[a,b,c,d,e]` und `[1,2,3,4,5]`. Der Kopf ist `[a,b,c,d,e]`, die Restliste lautet: `[[1,2,3,4,5]]`.

Das entsprechende Weindigramm sieht folgendermaßen aus:



### 3.2.6 Verhalten von Platzhaltern in Listen

Platzhalter in Listen verhalten sich genauso wie Platzhalter in anderen Strukturen.

### 3.2.7 Unifikation von Listen

Da Listen Strukturen sind, muss die Unifikationsroutine nicht erweitert werden, um Listen behandeln zu können.

Eine zentrale Anwendung der Unifikation zweier Listen ist das Zerlegen einer Liste in Kopf und Rumpf. Das Bearbeiten von Listen erfolgt in der Regel nach dem Schema: Bearbeite den Kopf der Liste, verarbeite die Restliste anschließend auf die gleiche Weise, breche ab, wenn die zu verarbeitende Liste leer (geworden) ist. Das ist ein Grund für die Wichtigkeit der oben genannten Operation. Die Zerlegung in Kopf und Restliste kann man vom System leicht erhalten, indem man die gewünschte Liste mit der Schablone `[Kopf|R]` oder `.(K,R)` unifiziert.

```
?- [affe, banane, problem] = [E|R].
```

E = affe

R = [banane, problem]

Yes

Weitere Beispiele für die Zerlegung einer Liste in Kopf und Rumpf:

Liste	Kopf	Rumpf
[x,y,z]	x	[y,z]
[]		Unifikation schlägt fehl
[[eins, zwei], drei]	[eins, zwei]	[drei]
[eins, [zwei, drei]]	eins	[[zwei, drei]]
[1, [2, 3], 4]	1	[[2, 3], 4]

Natürlich kann die Unifikation auch genutzt werden, um Listen zu verlängern:

*\*[Verlängern von Listen mittels Unifikation]*

```
?- X = [1,2,3 | Rest], Rest = [4, a, b, c, d].
```

```
X = [1, 2, 3, 4, a, b, c, d]
```

```
Rest = [4, a, b, c, d]
```

```
Yes
```

```
?- X = [a|R1], R1 = [b|R2], R2 = [c].
```

```
X = [a, b, c]
```

```
R1 = [b, c]
```

```
R2 = [c]
```

```
Yes
```

```
?- X = [1,2,3 | Rest], Rest = [].
```

```
X = [1, 2, 3]
```

```
Rest = []
```

```
Yes
```

Einige allgemeine Beispiele für die Unifikation zweier Listen:

Liste 1	Liste 2	Instantiierungen der Platzhalter		
[X, Y, Z]	[1, 2, 3]	X = 1	Y = 2	Z = 3
[X,Y Z]	[1, 2, 3]	X = 1	Y = 2	Z = [3]
[1]	[X Y]	X = 1	Y = []	
[1,2]	[1,X]	X = 2		
[1 R]	[1,2]	R = [2]		
[[1, X] Y]	[[Z,2],[3,4]]	X = 2	Y = [[3,4]]	Z = 1

### 3.2.8 ein beliebter Fehler

Folgende Darstellung einer zweielementigen Liste ist falsch:

```
[a | b]
```

Es handelt sich zwar um einen korrekten Term, nicht jedoch um eine Liste. Der Kopf des Ausdrucks ist korrekt, der Rest entspricht jedoch keiner Liste. Richtig sind:

```
.(a, .(b, []))
```

```
[a | [b | []]]
```

```
[a | [b]].
```

[a, b]

?[Lisp, dotted pair]

Die Konstruktion [a | b] ist mit einem “dotted pair” in Lisp vergleichbar.

### 3.2.9 ASCII-Zeichencodes

Schließt man eine Zeichenkette in doppelte Anführungszeichen ein, so wird sie vom System als eine Liste von ASCII-Zeichencodes gespeichert. Diese Liste wird sichtbar, sobald die Zeichenkette per Unifikation einem Platzhalter zugewiesen wird:

```
?- ASCII = "Prolog".
ASCII = [80, 114, 111, 108, 111, 103]
Yes
```

## 3.3 Strukturen vs. Listen

Mit Listen lassen sich auch Datenstrukturen wie Bäume, Graphen, Formeln, usw. repräsentieren. Listen sind natürlich nichts anderes als syntaktisch frisierte Prologstrukturen. Man hat also grundsätzlich die Wahl, Daten in Form von Prologlisten oder in Form von Prologstrukturen zu speichern.

### 3.3.1 “univ”

Darüberhinaus ist es möglich, die beiden Formen mittels [=..] ineinander zu konvertieren.

```
?- =..(X, [a,b,c]). 
X = a(b, c)
Yes

?- =..(funktor(komp1, komp2, komp3), X).
X = [funktor, komp1, komp2, komp3]
Yes
```

Das erste Argument von [=..] muss dabei ein Platzhalter oder eine Struktur sein. Das zweite Argument kann ein Platzhalter oder eine Liste sein. Beide Argumente dürfen nicht gleichzeitig Platzhalter sein. Bezeichnet wird [=..] als “univ”. Auch [=..] darf ‘infix’ geschrieben werden: [X =.. [a,1,2,3]].

## 3.4 einige “neue” Begriffe

Es ist an der Zeit, unseren Wortschatz um einige neue Vokabeln (für zum größten Teil schon bekannte Dinge) zu erweitern, die im Zusammenhang mit Prolog zum allgemeinen Sprachgebrauch gehören.

- Fakten und Regeln (also Aussagen) werden *Klauseln* genannt.
- Klauseln mit gleichem Funktor und gleicher Stelligkeit bilden die Definition eines sog. *Prädikats*.

- Eine zu beweisende Aussage (Anfrage) wird auch als *Goal* bezeichnet.
- Ein Goal (Ziel) kann *Subgoals* (Unterziele/ Unteranfragen) haben.  
Regeln haben die Form:

*goal* : –*subgoal*<sub>1</sub>, …, *subgoal*<sub>*n*</sub>.

Fakten kann man sich als Regeln mit leerem Rumpf vorstellen.

- Die meisten Leute sagen statt “Platzhalter” lieber “*Variable*”.

Ein illustrierendes Beispiel:

```
% Definition eines Prädikates vorfahr/2

% 1. Klausel
% Goal
vorfahr(A, B) :- elternteil(A, B). % Subgoal 1

% 2. Klausel
% Goal
vorfahr(MV, I) :- elternteil(D,I), % Subgoal 1
                 vorfahr(MV, D). % Subgoal 2
```



# Kapitel 4

## rekursive Programme

### 4.1 Rekursion

Endliche Mengen lassen sich aufschreiben, indem man alle ihre Elemente aufschreibt. Bei unendlich großen oder wenigstens potentiell unendlich großen Mengen scheitert dieses Vorgehen. Solche Mengen sind beispielsweise “die Menge der natürlichen Zahlen” oder “die Menge aller möglichen Prolog-Programme”.

Wir können “die Menge der natürlichen Zahlen” nicht komplett aufschreiben, aber wir können sie durch zwei einfache Regeln vollständig beschreiben:

- 0 ist eine natürliche Zahl.
- jede natürliche Zahl  $n$  hat einen Nachfolger  $n+1$ . In der so genannten Sukzessornotation schreibt man den Nachfolger  $s(n)$ .

Diese Regeln erlauben uns zwei Dinge. Zum einen können wir mit ihnen jede beliebige natürliche Zahl erzeugen. Die erste Regel dient uns dabei als Ausgangspunkt. Wir beginnen also mit der 0 und erhalten 1 (also  $s(0)$ ) als ihren Nachfolger, deren Nachfolger ist 2 (also  $s(s(0))$ ) usw. Auf diese Weise können alle natürlichen Zahlen erzeugt werden.

Zum anderen können wir mit den beiden Regeln überprüfen, ob es sich bei einer gegebenen Zahl um eine natürliche Zahl handelt: Wir gehen von der gegebenen Zahl zu ihrem Vorgänger zurück und erhalten damit eine neue (einfachere) Zahl. Diesen Vorgang wiederholen wir von der jeweils neu erhaltenen Zahl aus (Rekursion), bis die Zahl 0 erreicht ist (Rekursionsabbruch). Von 0 wissen wir, dass sie eine Zahl ist. Damit haben wir “bewiesen”, dass unsere Zahl von oben eine natürliche Zahl ist. Die erste Regel diente dabei als so genannte Abbruchbedingung.

Werden solche Regeln zur Erzeugung von Objekten/Strukturen eingesetzt, so spricht man von *Induktion*. Induktion und Rekursion bedeuten salopp ausgedrückt das gleiche mit jeweils umgekehrter “Laufrichtung”. Von Rekursion wird gesprochen, wenn ein komplexeres Objekt(/Struktur) durch wiederholtes Anwenden von Regeln auf ein einfacheres Objekt zurückgeführt wird.

Rekursion/Induktion kann also genutzt werden, um Objekte/Strukturen einer bestimmten (durch die Regeln vorgegebenen) Bauart zu verarbeiten bzw. zu erzeugen.

### 4.1.1 Sukzessornotation in Prolog

Gemäß der obigen Erläuterungen erstellen wir eine Definition der natürlichen Zahlen in Sukzessornotation in Prolog:

```
nat_zahl(0).                      % 0 ist nat_zahl
nat_zahl(s(X)) :- nat_zahl(X).    % s(X) ist nat_zahl, wenn X nat_zahl ist
```

Das Prädikat lässt sich sowohl zum Testen als auch zum Erzeugen von natürlichen Zahlen einsetzen:

```
?- nat_zahl(s(s(s(0)))).          % ist 3 nat_zahl?
```

Yes

```
?- nat_zahl(X).                  % Generierung
```

X = 0 ;

X = s(0) ;

X = s(s(0)) ;

X = s(s(s(0))) ;

X = s(s(s(s(0))))

.

.

.

.

.

Die Addition zweier natürlicher Zahlen könnte folgendermaßen aussehen:

```
addiere(0,C,C).
addiere(s(A),B,C) :- addiere(A,s(B),C).
```

Die ausgedrückte Relation besagt, dass **C** das Ergebnis der Addition von **A** und **B** ist.

Bei jedem Rekursionsschritt wird ein ‘Stück’ der ersten Zahl abgeknabbert und an die zweite Zahl angeklebt. Ist die erste Zahl auf diese Weise zur Null geworden, ist die zweite Zahl um den Betrag der ersten angewachsen und stellt das Ergebnis der Addition dar. Daher wird die im zweiten Argument enthaltene Zahl in der ersten Klausel an die dritte Argumentstelle gesetzt. Diese Konstruktion ist sehr gebräuchlich. (Unerfahrene Prologprogrammierer tendieren dazu, **addiere(0,B,C) :- B = C.** zu schreiben. Die obige Lösung wird als ‘eleganter’ empfunden.)

```
% Addition
%      1 + 2 = C
?- addiere(s(0),s(s(0)),C).
C = s(s(s(0)))                      % C = 3
Yes
```

Das Prädikat kann auch zur Subtraktion eingesetzt werden. Dazu muss  $\boxed{C}$  sowie entweder  $\boxed{A}$  oder  $\boxed{B}$  instantiiert sein. Es ist nicht schwer, Prädikate für weitere Rechenoperationen zu formulieren. In der Praxis benutzt man aufgrund der Laufzeit (je größer die Zahlen, desto länger dauert es) in Prolog die eingebauten Arithmetikprädikate, die wesentlich effizienter, jedoch nicht “logisch sauber”<sup>1</sup> sind.

### 4.1.2 Weitere Aspekte

Logisch (deklarativ) gesehen bedeutet Rekursion die Definition eines Prädikates unter Rückgriff auf das Prädikat selbst. Das heißt, dass die durch das Prädikat definierte Relation gilt, wenn eine (einfachere) Instanz der Relation gilt. Diese Verschachtelung setzt sich fort, bis die Abbruchbedingung erreicht worden ist. Sie stellt eine Relation dar, von der wir wissen, dass sie gilt, ohne dass sie auf eine einfache Relation zurückgeführt werden müßte.

Prozedural betrachtet ruft ein rekursives Prädikat sich selbst wieder auf. Das bedeutet, dass eine neue Instanz des Prädikats mit geänderten Werten erstellt wird und erfolgreich ausgeführt werden muss, bevor die erzeugende Instanz befriedigt werden kann. Diese Verschachtelung setzt sich fort, bis die Abbruchbedingung erreicht worden ist. Ist die Abbruchbedingung erreicht, so muss keine weitere Weiterverarbeitung initiiert werden.

Unabhängig vom präferierten Blickwinkel entsteht dadurch eine Art Schleife. Da diese Schleife im Allgemeinen nicht endlos lange fortgeführt werden soll, muss für ein Abbruchkriterium gesorgt werden, das die Schleife beendet. Mit dieser Technik lassen sich potentiell unendlich große Datenstrukturen verarbeiten.

### 4.1.3 Iterative vs. rekursive Schleifen

Es gibt in Prolog keine iterativen Schleifen (z.B. For-Schleife), wie sie etwa aus Pascal, C, Java, etc. als Basiskonstruktionen bekannt sind, in denen Variablenbindungen erhalten bleiben. Schleifen lassen sich in Prolog durch Rekursion realisieren. Das Problem mit iterativen Schleifen ist, dass es in Prolog nicht möglich ist, eine Variable als Laufvariable zu benutzen, also in jeder Iteration auf die gleiche Variablenbindung zuzugreifen. Platzhalter (Variablen) erlauben nur eine einmalige Wertzuweisung. Alle iterativen Schleifen lassen sich in rekursive Schleifen umwandeln. Später werden wir mit *Backtrackingschleifen* (siehe 5.8.1 auf Seite 79) einen weiteren Schleifentyp kennenlernen.

## 4.2 Rekursive Programme auf Listen

### 4.2.1 Auf Liste prüfen / Liste generieren

Unser erstes rekursives Programm auf Listen soll prüfen, ob es sich bei seinem Argument um eine Liste handelt. Wir erinnern uns an die Fallunterscheidung, die die rekursive Definition von “Liste” ausmacht:  $[]$  ist eine Liste,  $[E|L]$  ist eine Liste. Eine Liste ist also entweder die leere Liste oder ein Element gefolgt von einer Liste. Diese Vereinbarung können wir in Prolog aufschreiben:

---

<sup>1</sup>Dies schlägt sich in der Tatsache nieder, dass in den Arithmetikprädikaten nur jeweils eine uninstantiierte Variable vorkommen darf – im Gegensatz zur Sukzessornotation, die auch vor drei Unbekannten nicht halt macht.

```

liste([]).           % Die leere Liste ist eine Liste.

liste(X) :- X = [E|R],   % X in Kopf und Rumpf zerlegen
            liste(R).    % X ist Liste, wenn Rumpf von X eine Liste ist.

```

### Programmierstrategie

Die zweite Klausel ist unschön – die Zeile `[X = [E|R]]` kann man auch implizit durch das Prologsystem ausführen lassen, indem man statt `[X]` sofort `[E|R]` schreibt. (Dieses Vorgehen ist nicht nur ‘schöner’, sondern auch effizienter, da wesentlich weniger Terme mit `[E|R]` matchen als mit `[X]`.) Angemessener sieht das Ganze also so aus:

```

liste([]).           % Die leere Liste ist eine Liste.

liste([_E|R]) :- liste(R). % Das Argument ist eine Liste, wenn es sich in
                      % Kopf und Rumpf teilen lässt, und der Rumpf
                      % wiederrum eine Liste ist.

```

Wird eine Liste als Argument in das Prädikat hineingesteckt, so dient die erste Klausel als Abbruchbedingung. Die zweite Klausel verkürzt die übergebene Liste um ein Element und veranlasst die rekursive Weiterverarbeitung der Restliste. Irgendwann wird die Restliste leer und die Abbruchbedingung greift. Der Beweis gelingt. Handelt es sich bei dem Argument, das dem Prädikat übergeben wird, nicht um eine Liste, so schlägt im Verlauf des Beweises die Unifikation der Anfrage mit den Klauselköpfen fehl. In diesem Falle lässt sich also keine passende Regel für die aktuelle (Unter-) Anfrage finden und das System antwortet mit No.

Die Elemente einer Liste dürfen jede Art gültiger Term sein. Unserem Prädikat `liste` ist es egal, wie die Elemente aussehen — Hauptsache die oberste Ebene ist eine Liste. `liste` prüft nicht, ob die Elemente der Liste wohlgeformte Listen sind, sondern bleibt auf der obersten Listenebene und schenkt den Elementen an sich keine Beachtung.

Probieren wir das Prädikat aus:

```

?- liste([]).           % leere Liste ist eine Liste (Fakt)
Yes

?- liste([1,2,3]).       % alles in Ordnung
Yes

?- liste([1,2,3|[]]).    % selbe Liste, andere (äquivalente)
                        % Notation
Yes

?- liste([1|[2|[3|[]]])). % selbe Liste, andere (äquivalente)
                           % Notation
Yes

?- liste((1,(2,(3,[])))). % selbe Liste, andere (äquivalente)
                           % Notation
Yes

?- liste([1|2]).          % Rest ist keine Liste! (dotted pair)
No

```

```

?- liste([[1|2]]).           % einelementige Liste, deren Element
Yes                         % ein dotted pair ist

?- liste([a,[1|2],b]).      % dotted pair als Element ist ok!
Yes

?- liste(abc).              % Atom lässt sich nicht
No                           % in Kopf und Rumpf aufspalten

?- liste([[1,2,3],[a,b,c]]). % Listenelemente können wiederum
Yes                         % Listen sein

```

Das Prologsystem spaltet mit der zweiten Klausel solange je ein Element (den Kopf) der gegebenen Liste ab und ruft das Prädikat wieder auf mit der nun kürzer gewordenen Liste, bis es sich bei der Liste für den rekursiven Aufruf um die leere Liste handelt. Wird die leere Liste erreicht, so ist der Beweis beendet. Keine weiteren rekursiven Aufrufe finden statt.

#### 4.2.2 Ausführliche Beispiele, Rekursionsabstieg und -aufstieg

Die Unifikation von Listen mit dem Term [E|R] wurde in Abschnitt 3.2.7 auf Seite 46 vorgestellt. Wir wollen die Anfrage `liste([a,b,c]).` ausführlich betrachten:

- `liste([a,b,c]).` matcht mit dem Kopf der zweiten Klausel, es entsteht folgende Instanz:

```
% Anfrage 1      :- Anfrage 2
liste([a|[b,c]]) :- liste([b,c]).
```

Damit der Beweis von Anfrage 1 erfolgreich enden kann, muss eine neue Instanz (Anfrage 2) des `liste`-Prädikats befriedigt werden. Gelingt Anfrage 2, so kann der Beweis von Anfrage 1 fortgesetzt werden. Da in Anfrage 1 anschließend nichts mehr zu beweisen ist, ist das System fertig.

- Das Matchen von Anfrage 2 mit der zweiten Klausel von `liste` führt zu folgender Instanz:

```
% Anfrage 2      :- Anfrage 3
liste([b|[c]]) :- liste([c]).
```

Anfrage 2 gelingt erst, wenn Anfrage 3 befriedigt worden ist.

- Anfrage 3 matcht mit der 2. Klausel und führt zu folgender Instanz:

```
% Anfrage 3      :- Anfrage 4
liste([c|[]]) :- liste([]).
```

Anfrage 3 gelingt erst, nachdem Anfrage 4 erfolgreich abgearbeitet worden ist.

- Anfrage 4 matcht mit der 1. Klausel:

```
% Anfrage 4
liste([]).
```

Anfrage 4 gelingt. (Sie entspricht einem Fakt aus der Wissenbasis.)

`[]` ist also eine Liste.

Den Teil des Beweises bis hierhin nennt man *Rekursionsabstieg*. Jeder rekursive Aufruf ist wie ein Schritt eine Treppe hinunter. An dieser Stelle haben wir das Ende der Treppe erreicht. Es folgt der so genannte *Rekursionsaufstieg*, bei dem wir die Treppe wieder Stufe für Stufe herauflaufen, bis wir den Punkt erreichen, an dem die ganze Rennerei begonnen hat.

- Mit dem Gelingen von Anfrage 4 gelingt Anfrage 3.  
`[c]` ist also eine Liste.
- Mit dem Gelingen von Anfrage 3 gelingt Anfrage 2.  
`[b,c]` ist also eine Liste.
- Mit dem Gelingen von Anfrage 2 gelingt Anfrage 1.  
`[a,b,c]` ist also eine Liste.

Das Ganze nochmal mit der Anfrage `liste([[1|2],3])`:

- `liste([[1|2],3])`. matcht mit der 2. Klausel:

```
liste([[1|2],3]).                      % Anfrage 1
liste([_E|R]) :- liste(R).            % 2. Klausel

liste([[1|2]|[3]]) :- liste([3]).    % resultierende Instanz
                                % Anfrage 2
                                liste([3]).
```

Der Platzhalter `_E` enthält nun den Kopf der Liste, also den Wert `[1|2]`. Dies ist ein Element der Liste, es darf jeder gültige Term sein, also auch ein “dotted pair”. Dieses Listenelement wird außerdem nirgendwo im Prädikat “angeguckt”. Der Liste ist egal, wie die in ihr enthaltenen Elemente aussehen.

An diesem Punkt geht es mit der Unteranfrage `liste([3])` weiter.

- Folgende Regelinstanz entsteht nun:

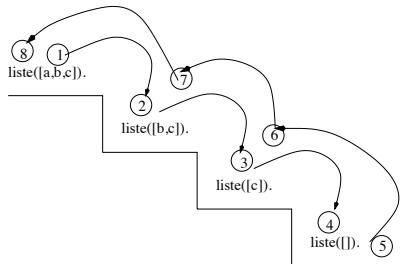
```
liste([3]).                          % Anfrage 2
liste([_E|R]) :- liste(R).          % 2. Klausel

liste([3|[ ]]) :- liste([]).        % resultierende Instanz
                                liste([]).      % Anfrage 3
```

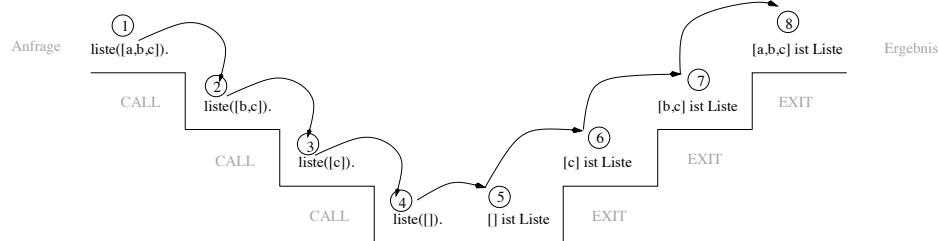
Auch die `3` wird nicht verarbeitet. Hier interessiert immer nur der Rest der Liste.

- Die Untermanfrage 3 `liste([])` gelingt gemäß der 1. Klausel des Prädikates. Der Rekursionsabstieg ist beendet, es folgt der Rekursionsaufstieg.
- Da Anfrage 3 gelungen ist, gelingt Anfrage 2.
- Da Anfrage 2 gelungen ist, gelingt Anfrage 1. Damit ist der Beweis gelungen.

#### 4.2.3 Veranschaulichung von Rekursionsabstieg und -aufstieg



Etwas übersichtlicher:



CALL: Aufruf einer Untermanfrage, EXIT: Erfolgreicher Beweis einer Untermanfrage

Man kann den Beweis einer Anfrage am Bildschirm verfolgen, indem man das Prologsystem veranlasst, eine Ablaufverfolgung (*Trace*) auszugeben. Auch in der Trace können Rekursionsabstieg und -aufstieg wiedererkannt werden.

```
?- trace(liste).
%          liste/1: [call, redo, exit, fail]
```

```
Yes
[debug] ?- liste([a,b,c]).
T Call: (6) liste([a, b, c])
T Call: (7) liste([b, c])
T Call: (8) liste([c])
T Call: (9) liste([])
T Exit: (9) liste([])
T Exit: (8) liste([c])
T Exit: (7) liste([b, c])
T Exit: (6) liste([a, b, c])
```

```
Yes
[debug] ?- notrace.
```

In einem späteren Kapitel werden wir die Trace ausführlicher betrachten. \*\*wo? verweis!\*

#### 4.2.4 Die andere Richtung - Erzeugen von Listen

Das Prädikat funktioniert auch in der anderen Richtung: Man kann es einsetzen, um beliebig lange Listen zu generieren. Dazu muss man es mit einer uninstantiierten Variable aufrufen. Hier wird die Abbruchbedingung (1. Klausel) als Ausgangspunkt aufgefasst, mit dem man die Erzeugung beginnt. Hat man erstmal auf diese Weise eine Liste erzeugt, so kann man aus ihr mit der zweiten Regel des Prädikats eine neue, längere Liste erzeugen.

```
?- liste(X).

X = [] ;
X = [_G231] ;
X = [_G231, _G234] ;
X = [_G231, _G234, _G237] ;
X = [_G231, _G234, _G237, _G240] ;
X = [_G231, _G234, _G237, _G240, _G243] ;
X = [_G231, _G234, _G237, _G240, _G243, _G246];
⋮ ⋮ ⋮ ⋮ ⋮
⋮ ⋮ ⋮ ⋮ ⋮
⋮ ⋮ ⋮ ⋮ ⋮
Yes
```

#### 4.2.5 `is_list/1`, schwächere Version des Checks

SWI-Prolog verfügt über ein eingebautes Prädikat (ein so genanntes Built-in) namens `is_list/1`. Es handelt sich dabei um eine schwächere Version des oben vorgestellten Prädikates. `is_list/1` akzeptiert auch “dotted pairs” als wohlgeformte Listen und funktioniert nur als Check, kann also keine Listen erzeugen. Diese schwächere Version wird benutzt, weil sie die Liste gar nicht rekursiv abarbeitet, sondern lediglich das erste Cons-Paar überprüft. Das bringt einen enormen Geschwindigkeitsvorteil. Eine eigene Implementation könnte so aussehen:

```
listencheck([]). % wie gehabt
listencheck([_|_]). % passt der Anfang der Struktur in [_|_]?

% Beispiele
?- listencheck([1,2,3,4]).
Yes

?- listencheck([1|2,3|4]). % das geht bei liste/1 nicht!
Yes
```

Dieses Prädikat ist zwar sehr schnell, aber nicht so sicher wie `listo`.

#### 4.2.6 Rekursive Suche - `enthalten/2`

Oft muss man ein bestimmtes Element in einer Liste finden. Das folgende Vorgehen wird “rekursive Suche” genannt. Es handelt sich wieder um eine Fallunterscheidung:

- Das gesuchte Element ist Kopf der zu durchsuchenden Liste.
- Das gesuchte Element ist nicht der Kopf der Liste. In diesem Fall starte man eine rekursive Suche in der Restliste.

Auf diese Weise lässt sich das gesuchte Element finden, wenn es tatsächlich in der Liste vorkommt. Tritt das gesuchte Element nicht in der Liste auf, so scheitert die Suche, da die Restliste, auf der rekursiv weitergesucht werden soll, schließlich die leere Liste sein wird. Dies passiert zwangsläufig, weil jeder folgende Rekursionsschritt mit einer um ein Element verkürzten Liste durchgeführt wird. Die leere Liste lässt sich nicht in Kopf und Rumpf spalten, also kann sie nicht weiterverarbeitet werden. Der Beweisversuch findet auf diese Weise sein Ende. In Prolog:

```
enthalten(X,[X|_]).  
enthalten(X,[_|R]) :- enthalten(X,R).
```

Beispiele

```
?- enthalten(2,[1,2,3]).
```

Yes

```
?- enthalten(2,[a,b,c,d,e,f]).
```

No

```
?- enthalten(a,[a,b,c,d,e,f]).
```

Yes

```
?- enthalten(X,[a,b,c,d,e,f]).
```

X = a ;

X = b ;

X = c ;

X = d ;

X = e ;

X = f ;

No

```
?- enthalten(objekt,X). % generieren...
```

X = [objekt|\_G300] ;

X = [\_G299, objekt|\_G303] ;

X = [\_G299, \_G302, objekt|\_G306] ;

X = [\_G299, \_G302, \_G305, objekt|\_G309]

...

Yes

```
?- enthalten(0,X). %So geht's auch, aber man muss sich fragen, ob
%man irgendetwas davon hat...
0 = _G222
X = [_G222|_G285] ;

0 = _G222
X = [_G284, _G222|_G288] ;

0 = _G222
X = [_G284, _G287, _G222|_G291]
```

Yes

Dieses Prädikat ist ein Beispiel für die typische Struktur von Prologprogrammen.

Natürlich gibt es ein Builtin für rekursive Suche in Listen. Es handelt sich dabei um `member/2`. Ist man nur an dem ersten Auftreten eines Terms in einer Liste interessiert, so bietet sich die Verwendung von `memberchk/2` an.

Im Abschnitt 7.2.4 auf Seite 103 wird eine Variante rekursiver Suche vorgestellt, die den Zugriff auf eine bestimmte Position in einer Liste ermöglicht. Dazu werden Arithmetikprädikate benötigt, die in Kapitel 7 vorgestellt werden. Ein entsprechendes Builtin heißt `nth1/3`.

#### 4.2.7 `write/1`, `nl/0` - Ausgabe

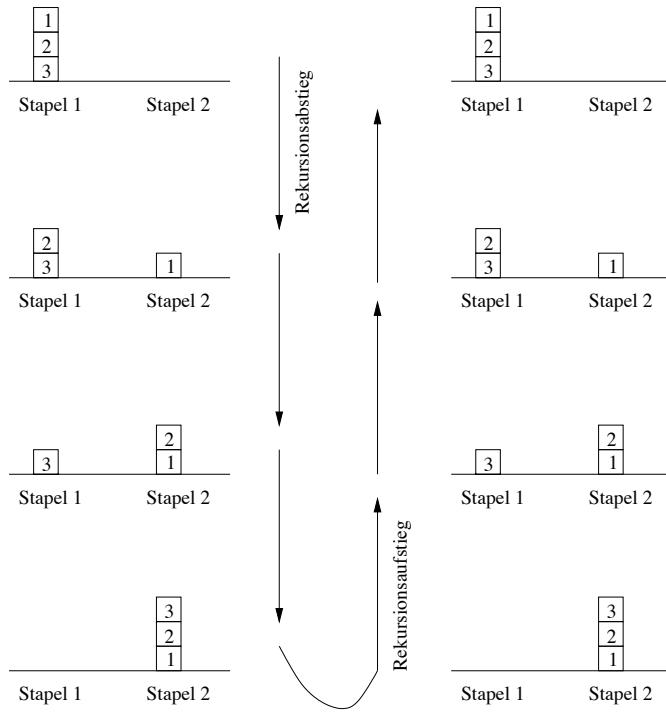
Um im folgenden Beweisversuchen zuschauen zu können, führen wir die eingebauten Prädikate `write/1` und `nl/0` aus dem Bereich der Ein- und Ausgabe an dieser Stelle ein. Erstes gibt den als Argument übergebenen Term aus, letzteres bewirkt einen Zeilenumbruch. Beide Prädikate gelingen immer, sie haben auf den Beweis keine Auswirkungen, jedoch den Nebeneffekt, dass sie Zeichen auf die Standardausgabe schreiben. Weder für `write/1` noch `nl/0` gibt es alternative Beweise.

#### 4.2.8 Umdrehen von Listen, Strukturaufbau beim Rekursionsabstieg

Die Liste `[3,2,1]` soll umgedreht werden in die Liste `[1,2,3]`.

Wir klappern die übergebene Liste Element für Element ab und konstruieren dabei eine neue Liste, an die wir das jeweilige erste Element der übergebenen Liste als Kopf anhängen.

Diesen Vorgang kann man sich als “Umstapeln von Kisten” vorstellen. Von einem ersten Stapel wird solange die oberste Kiste entfernt und auf den zweiten Stapel oben draufgelegt, bis der erste Stapel leer ist. Der zweite Stapel besitzt dann die umgekehrte Reihenfolge.



Der Zeichnung entsprechend werden dem folgenden Prädikat nun zwei Listen übergeben, ein voller Stapel und ein leerer Stapel.

```
u([0berstes1|Reststapel1], Stapel2bisher) :- u(R, [0berstes1|Stapel2bisher]).  
u([], Stapel2) :- write(Stapel2), nl.  
?- u([3,2,1], []).  
[1, 2, 3]
```

Yes

Das gewünschte Ergebnis wird per `write` ausgegeben, aber ansonsten nicht "zurückgeliefert". Wir führen ein drittes Argument ein, das diese Aufgabe übernehmen soll.

```
u([E|Reststapel1], Stapel2, Ergebnis) :- u(Reststapel1, [E|Stapel2], Ergebnis).  
u([], Stapel2, Ergebnis) :- Ergebnis = Stapel2.
```

Dem Prädikat werden nun der erste (volle) Stapel, der zweite (leere) Stapel und eine (uninstantiierte) Variable für das Ergebnis übergeben.

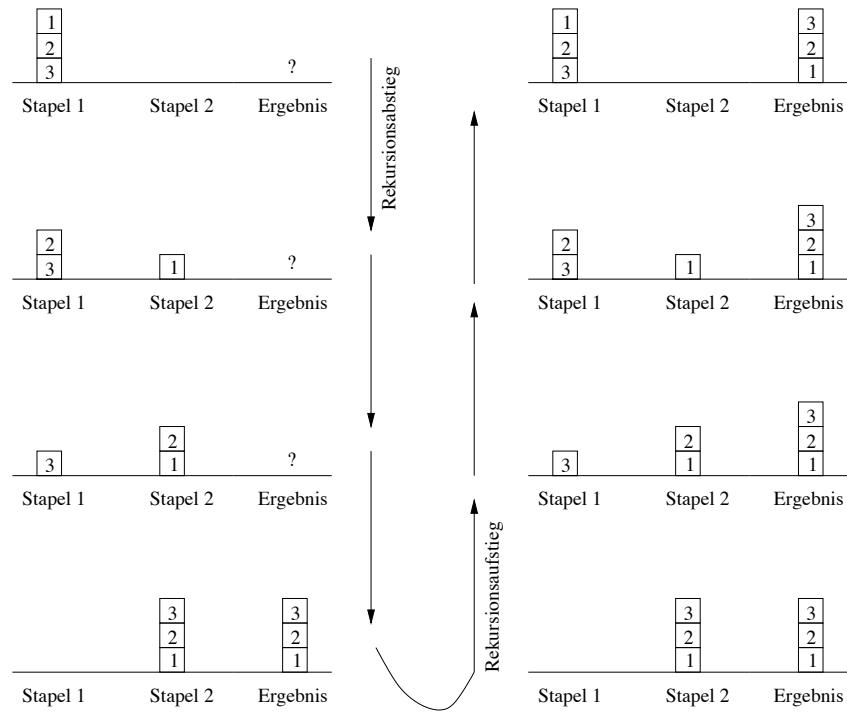
```
?- u([3,2,1], [], R).
```

```
R = [1, 2, 3]
```

Yes

Das Interessante dabei ist, dass das Ergebnis während des Rekursionsabstiegs konstruiert wird und dafür gesorgt werden muss, dass es auch oben “rauskommt”.

Der zweite Stapel wird jeweils um ein Element verlängert an die nächste Rekursionsebene übergeben. Beim Rekursionsaufstieg gilt für jede Ebene nur der Stapel, der übergeben wurde, der zweite Stapel “schrumpft” also wieder zurück, während der erste wieder “wächst”. Wir müssen also eine Möglichkeit suchen, das Ergebnis des Umdrehens nach oben zu schmuggeln. Dazu dient das dritte Argument, das uninstantiiert übergeben wird. Die aktuelle Rekursionsebene “wartet” auf das Ergebnis der nächsten Ebene. Dieses Ergebnis liefert sie dann brav an ihre Vorgängerebene ab. Ist die erste Liste (nach einigen Rekursionsschritten) leer, so wissen wir, dass die zweite Liste das Ergebnis darstellt. Wir instantiiieren das dritte Argument in der Abbruchbedingung mit dem zweiten Stapel. Während des Rekursionsaufstieges wird das dritte Argument von Ebene zu Ebene “durchgereicht”.



Den zweiten Stapel kann man verstecken, indem man ein neues Prädikat erstellt, das den eigentlichen Aufruf erledigt. Die Abbruchbedingung lässt sich ebenfalls vereinfachen. Vor allen Dingen sollte man sie als erste Klausel aufführen.

```
umdrehen(Liste, Ergebnis) :- u(Liste, [], Ergebnis).
```

```
u([], Ergebnis, Ergebnis).
```

```
u([E|Reststapel1], Stapel2, Ergebnis) :- u(Reststapel1, [E|Stapel2], Ergebnis).
```

```
% Beispiele
```

```
?- umdrehen([3,2,1],X).
X = [1, 2, 3]
Yes

?- umdrehen([a,b,c],[c,b,a]). 
Yes

?- umdrehen(X,[4,7,3]). 
X = [3, 7, 4]
Yes
```

Würde die Abbruchbedingung nicht an erster Stelle stehen, so würde die dritte Beispielanfrage zu einer Endlosschleife führen. Die rekursive Klausel matcht mit der Anfrage, das 1. Argument wird nie zur leeren Liste, da eine uninstantiierte Variable diesen Platz einnimmt ohne jemals instantiiert zu werden. Die Liste im zweiten Argument wird je rekursiven Aufruf länger. Da die Abbruchbedingung nie erreicht wird, geht dem System der Speicher aus. Mehr über die Vorgehensweise des Prologsystems findet sich im nächsten Kapitel.

Natürlich kann das Umdrehen auch auf andere Weise erledigt werden. Die hier vorgestellte Version gilt jedoch als elegant, da sie eine n-elementige Liste in  $n$  Schritten umdreht. Es gibt ein entsprechendes Builtin namens `reverse/2`.

#### 4.2.9 Restrekursion

Für jeden rekursiven “Aufruf” muss das Prologsystem ein Stück Speicher reservieren, in dem es die Variablenbelegungen dieser Ebene festhält. Je tiefer die Rekursion, desto mehr Speicher wird benötigt.

Steht der rekursive Aufruf in einer Regel an letzter Stelle, ergibt sich eine interessante Optimierungsmöglichkeit für das System: Es kann den zuletzt angelegten Speicherbereich für die Variablen wiederbenutzen, d.h. die alten Werte mit den Werten der neuen Ebene überschreiben, ohne dass dies das Ergebnis des Beweises beeinflussen würde. Dies wird als *Restrekursionsoptimierung*, bzw. *Tail Recursion Optimization* bezeichnet.

Restrekursion gilt als elegant. (Prinzipiell unterscheidet sich Restrekursion nicht von einer iterativen Schleife.)

#### 4.2.10 `verkette/3`- Listen verketten, Strukturaufbau beim Rekursionsaufstieg

Das Aneinanderhängen zweier Listen ist für viele Anwendungen eine grundlegende Operation. Das Prädikat `verkette` benötigt drei Argumente, zwei Teillisten, die verbunden werden sollen und einen Platz für die Ergebnisliste.

```
verkette([1,2,3],[a,b,c],[1,2,3,a,b,c]).
```

builtin: append verschiedene Instantiiierungen

- 4.2.11 Suffix, Präfix, Infix - Listen**
- 4.2.12 Löschen von Elementen aus Listen**
- 4.2.13 Menge, Teilmenge, Vereinigung, Schnitt**
- 4.2.14 Permutationen von Listen**
- 4.2.15 Sortieren von Listen**
- 4.2.16 Flachklopfen von verschachtelten Listen**
- 4.2.17 Umformungen von Listen**
- 4.2.18 Palindrome**

Palindrome ergeben von rechts nach links gelesen denselben Text wie von links nach rechts. Das Prädikat zum Umdrehen von Listen kann genutzt werden, um zu überprüfen, ob eine Liste diese Eigenschaft besitzt.

Die folgenden Palindrome müssen dazu allerdings erst in eine verdauliche Form gebracht werden. Großbuchstaben müssen in Kleinbuchstaben umgewandelt, Interpunktionszeichen und Leerstellen entfernt werden. Anschliessend muss eine Liste erzeugt werden, in der die Buchstaben einzeln aufgeführt werden. Diese kann an das Prädikat verfüttert werden.

Lived on Decaf, Faced no Devil. Plan no damn Madonna LP! Du, erfror Freud? I'm, alas, a salami. Gnudung! Never odd or even. A man, a plan, a canal - panama! I love Me - Vol. I Leg in eine so helle Hose nie 'n Igel! Are we not drawn onward, we few, drawn onward to new era?

## 4.3 Rekursive Programme auf Bäumen

### 4.3.1 Traversierung

Im Baum aus Abschnitt 3.1.1 auf Seite 40 sollen alle Knoten besucht werden.

```
bsp(
baum(
    baum(
        baum(          %
            baum(nil, a, nil),   %   /L   )
            b,                 %   W   > linker Teilbaum
            baum(nil, c, nil)  %   / \R   )
        ),                %
        %
        d,                  % Wurzel
        %
        baum(          %
            baum(nil, e, nil),   %   \ /L   )
            f,                 %   W   > rechter Teilbaum
            baum(nil, g, nil)  %   \R   )
        )
    )
).
```

Eine Inorder-Traverse besteht darin, zunächst den linken Teilbaum zu behandeln, danach die Wurzel und anschließend den rechten Teilbaum. Das Vorgehen wird auf die Teilbäume rekursiv angewendet, an Blättern wird die Rekursion abgebrochen.

Damit es dabei auch etwas zu sehen gibt, benötigen wir die Prädikate `write/1` und `nl/0`, die in Abschnitt 4.2.7 auf Seite 60 kurz eingeführt wurden.

In Prolog sieht die Traverse folgendermaßen aus:

```
inorder(baum(L,W,R)) :- inorder(L),    % L rekursiv verarbeiten
                           write(W), nl,   % W ausgeben
                           inorder(R).    % R rekursiv verarbeiten
```

An den Blättern des Baumes ist Schluss:

```
inorder(nil).                      % "nichts tun" und fertig
```

Das Prädikat in Aktion:

```
?- bsp(_X), inorder(_X).
a
b
c
d
e
f
g
```

Yes

### 4.3.2 Prolog, Logik und Ein-/Ausgabe

Man kann Ausgabeprädikate benutzen, um einem Beweis zuzuschauen. Prä- und Postorder-Traversen lassen sich durch Umordnen der Subgoals der ersten Klausel der Inorder-Traverse erreichen. Logisch gesehen hat die Reihenfolge der Subgoals keinerlei Bedeutung. Vom prozeduralen Standpunkt her jedoch schon: Prolog arbeitet die Klauseln gemäß ihrer Reihenfolge ab, damit ist garantiert, dass die Ausgabe an der Stelle geschieht, an der sie vorgesehen ist. Würde man Prolog gestatten, Klauseln in beliebiger Reihenfolge abzuarbeiten, könnte man diese Garantie nicht mehr geben — Ein-/Ausgabe würde an nicht vorhersehbaren und sinnlosen Zeitpunkten auftreten. Näheres zur Mechanik des Prologbeweisers findet sich im nächsten Kapitel.



# Kapitel 5

## Die Beweisprozedur von Prolog

Prolog geht beim Beantworten von Anfragen nach einem bestimmten Verfahren vor. Teile dieses Verfahrens haben wir schon kennen gelernt. Nun wollen wir unsere Vorstellungen vervollständigen. Der Teil des Prologsystems, der die Beweise durchführt, wird “Beweiser” genannt.

In diesem Kapitel soll es um das mechanische Vorgehen des Beweisers gehen, die logische Fundierung wird in dem Kapitel über Prädikatenlogik geliefert werden.

### 5.1 Klauselauswahl, Choicepoints

Bisher wissen wir, dass das Prologsystem Klauseln auswählt, die mit dem aktuellen Goal matchen (= zusammenpassen, unifizierbar sind). Die Suche nach einer matchenden Klausel in der Wissensbasis erfolgt von oben nach unten. Das bedeutet, dass die Reihenfolge der Klauseln in der Quelldatei eine wichtige Rolle beim Programmieren spielt. Das Prologsystem “entscheidet” sich immer zuerst für die oberste matchende Klausel. Die Quelldatei wird von oben nach unten durchsucht.

### 5.2 Backtracking

Die entsprechende Stelle in der Wissensbasis wird markiert, es wird ein sogenannter *Choicepoint* eingerichtet, der auf die nächste potentielle Alternative – also die nächste Klausel mit gleichem Funktor und gleicher Stelligkeit – zeigt. Außerdem enthält der Choicepoint die Belegungen der beteiligten Platzhalter, bevor das zum Choicepoint gehörende Prädikat ausgeführt worden ist. Diese Informationen ermöglichen das “Rückgängigmachen” von Platzhalterbelegungen und bereits erfolgten Teilbeweisen. Wir erinnern uns daran, dass Platzhalter nur einmal einen Wert aufnehmen, der anschließend unveränderlich bleibt. Um doch andere Belegungen zu ermöglichen (für alternative Beweise/Lösungen), müssen Teilbeweise und aus ihnen resultierende Instantiierungen von Platzhaltern zurückgenommen werden. Das Verfahren, in einem Beweis Schritte und

Instantiierungen zurückzunehmen, wird *Backtracking* genannt.

Ist ein solcher Rückwärtschritt getan, so befindet sich der Beweis in einem Zustand, in dem das System eine andere Klausel als zuvor wählen kann, die in anderen Instantiierungen resultiert, ohne dass die Bedingung verletzt ist, dass Platzhalter nur eine einmalige Wertzuweisung akzeptieren.

Wird also ein alternativer Beweis für ein Goal nötig, so probiert das System am letzten angelegten Choicepoint die nächste passende Alternative und benutzt dabei die ursprünglichen Belegungen der Platzhalter. Existiert keine passende Alternative, so scheitert der (Teil-)Beweis.

Gibt der aktuelle Choicepoint keine Alternativen mehr her, so wird erneutes Backtracking initiiert, das zum nächsthöheren Choicepoint führt. Ein alternativer Beweis ab dieser Ebene kann natürlich dazu führen, dass neue Choicepoints aufgebaut werden. Für das Backtracking kommt zunächst immer nur der zuletzt angelegte Choicepoint in Frage. Erst wenn dieser keine Alternativen mehr bietet, wird zu seinem Vorgänger zurückgesprungen.

### 5.3 Und-/Oder-Bäume

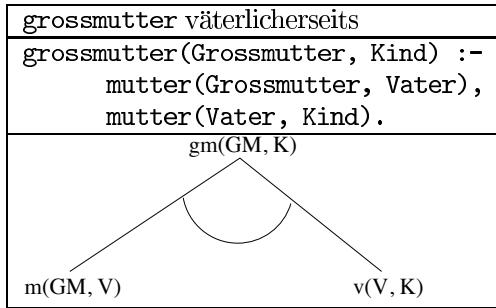
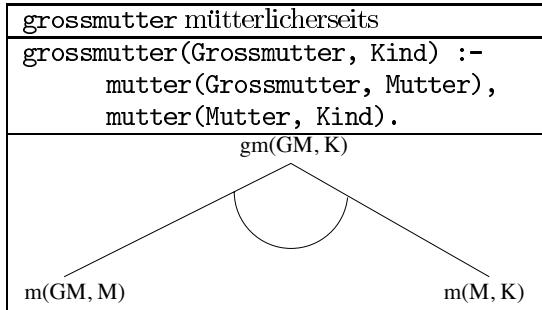
Prologprogramme und deren Abarbeitung lassen sich grafisch in Form sogenannter *Und-/Oder-Bäume* darstellen. Solche Bäume lassen sich auf folgende Weise erstellen:

- Die zu beweisende Aussage bildet die Wurzel des Baumes.
- Als Nachfolger der Wurzel werden alle Klauselköpfe des Prädikates aus der Anfrage eingetragen. Sie stellen Alternativen zueinander dar. Daher werden sie als “oder-verknüpft” angesehen. Auf dem Weg durch den Baum muss jeweils eine und nur eine der Alternativen gewählt werden.
- Zu jedem der alternativen Klauselköpfen müssen nun als Nachfolger die jeweiligen Klauselrumpfe eingetragen werden. Die Unteranfragen aus dem Klauselrumpf werden dabei auf einer Ebene von links nach rechts aufgeschrieben. Sie werden als “und-verknüpft” bezeichnet, da sie alle erfolgreich bewiesen werden müssen, um ihren Vater in dem Baum zu beweisen. Diese Verknüpfung wird in der grafischen Darstellung durch einen Bogen durch die Verbindungslien gekennzeichnet.
- Für jede dieser Unteranfragen ist das Verfahren zu wiederholen.

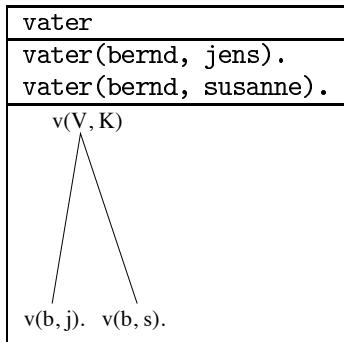
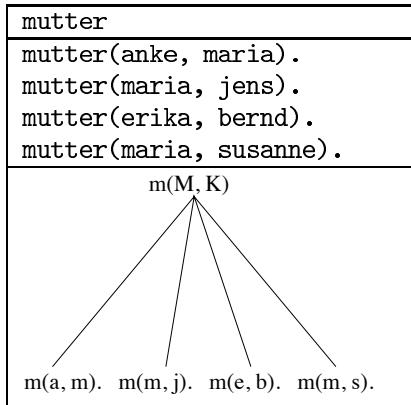
Die Blätter eines solchen Baumes bestehen aus Fakten. Ein Beweis kann als Traverse eines Und-/Oder-Baumes aufgefasst werden. Daher werden Und-/Oder-Bäume auch als *Beweisbäume* bezeichnet.

#### 5.3.1 Beispielbaum

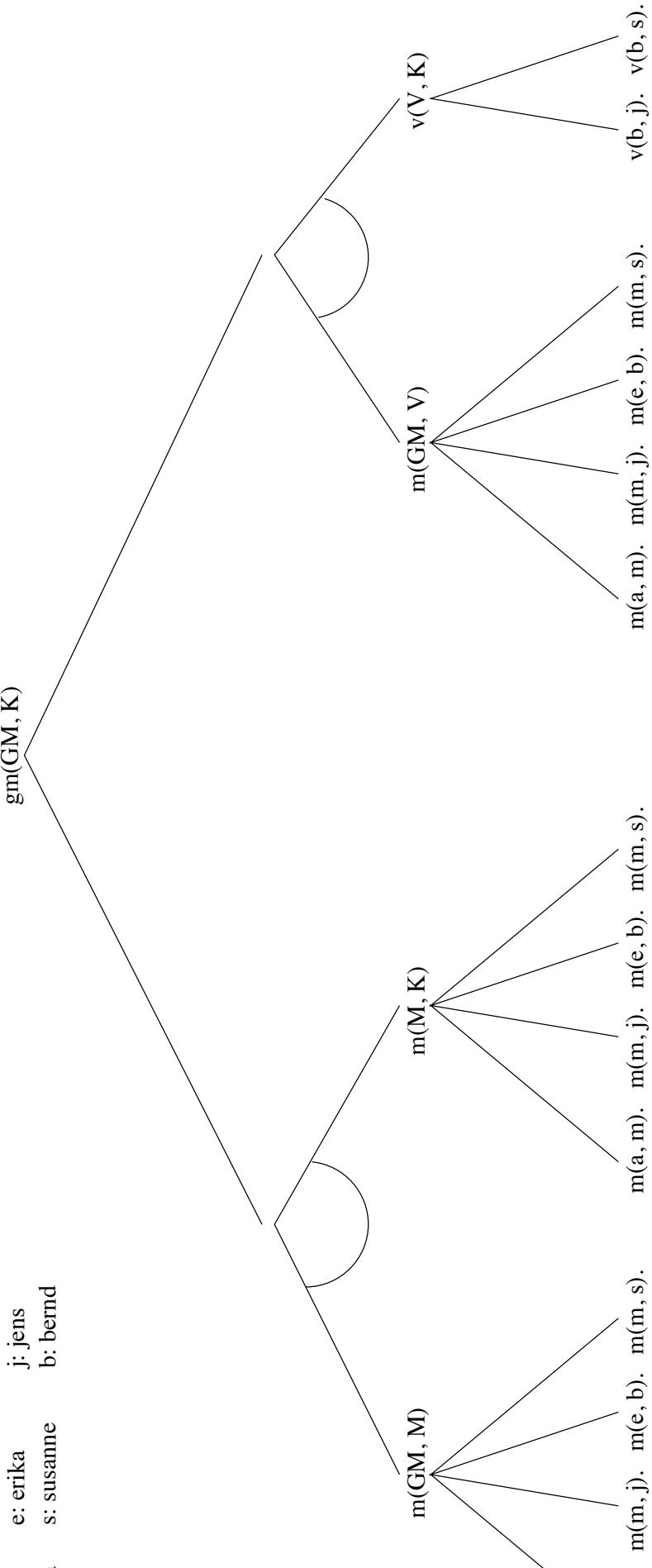
Aus dem Beispielprogramm 1.12 konstruieren wir einen Beweisbaum für das `grossmutter`-Prädikat. Es soll in diesem Fall keine konkrete Anfrage gestellt werden, so dass der resultierende Baum eine Schablone für alle `grossmutter`-Anfragen darstellen wird. Zunächst erzeugen wir Teilbäume, die wir anschließend zusammensetzen. Für die Grafiken verwenden wir aus Platzgründen Abkürzungen, die aber aus den Tabellen ersichtlich sein sollten.



Natürlich benötigen wir auch entsprechende Bäume für `mutter` und `vater`:



Setzen wir diese Teilbäume zusammen, erhalten wir den kompletten Beweisbaum:



Anmerkung: Rekursive Prädikate erzeugen Beweisbäume, die sehr, bzw. unendlich groß werden können.

Stellt man die Abbruchbedingung eins eines Prädikates hinter eine rekursiv Klausel, die *immer matcht*, wird der linke Ast des Beweisbaumes unendlich lang. Die Abbruchbedingung ist in einem solchen Fall unerreichbar, der Beweis findet kein Ende. Das System ist in einem endlosen Beweis verfangen. Gerade bei rekursiven Prädikaten spielt die Klauselreihenfolge eine entscheidende Rolle, generell sollte man die Abbruchbedingung als erste Klausel notieren.

Auf der Suche nach einem Beweis werden die Knoten des Und-/Oder-Baumes in systematischer Weise besucht:

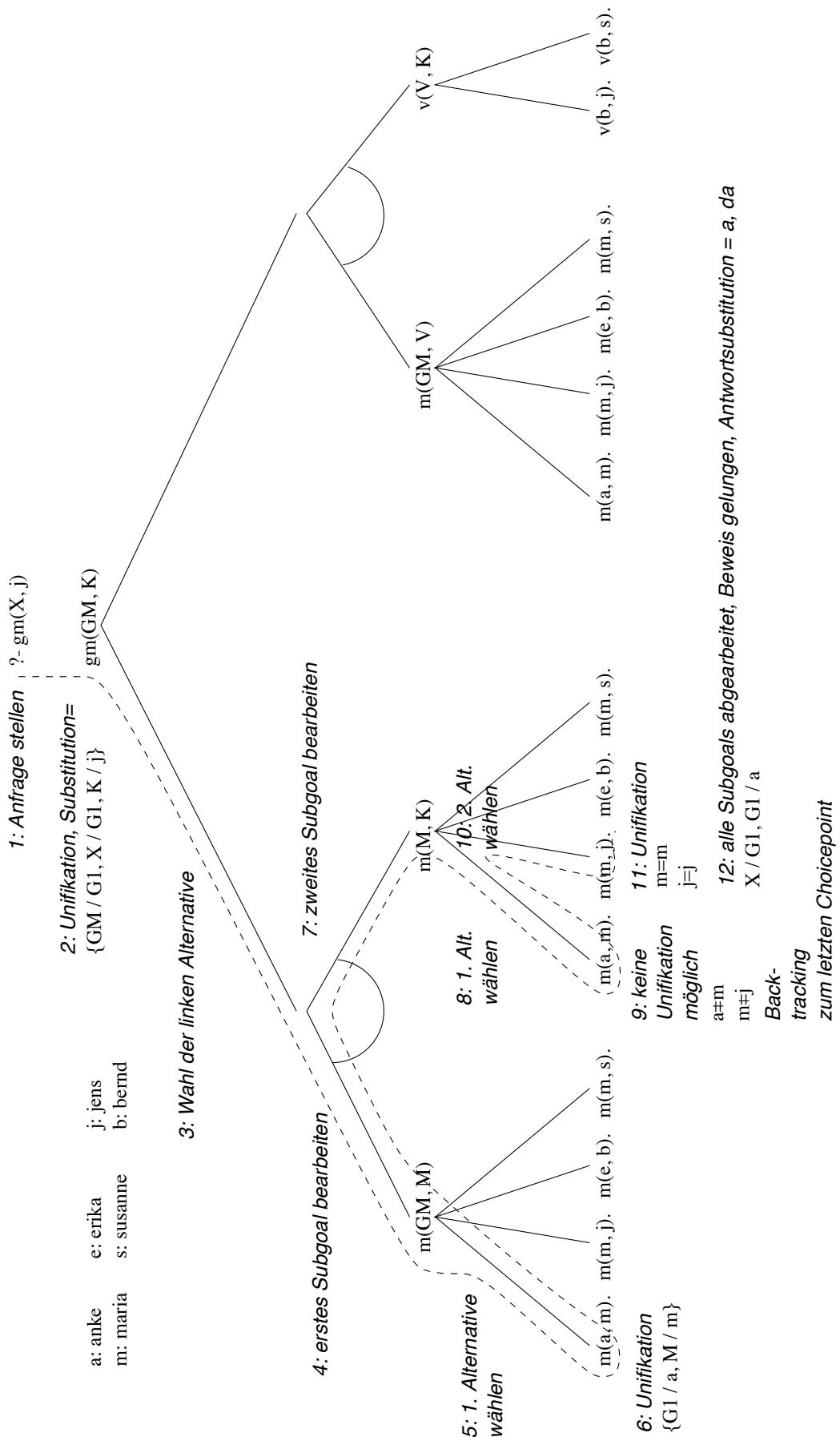
- Es wird an der Wurzel begonnen.
  
  
  
- Das System wählt aus mehreren Alternativen die jeweils am weitesten links stehenden aus, die im bisherigen Teilbeweis noch nicht besucht worden ist.
  
  
  
- Undverknüpfte Klauseln werden von links nach rechts abgearbeitet.

Dieses Verfahren lässt sich mit den Worten “von oben nach unten, von links nach rechts” charakterisieren. Es handelt sich dabei um eine sogenannte *Tiefensuchstrategie*. Im Falle von Prolog ist die Tiefensuche mit einem Backtracking-Mechanismus ausgestattet, der ein Rückgängigmachen von Variablenbelegungen ermöglicht. Oder-Knoten stellen dabei die Choicepoints dar.

Die Reihenfolge der Klauseln in der Wissensbasis legt die Reihenfolge der Alternativen im Beweisbaum von links nach rechts fest. Ebenso wird die Reihenfolge der Untieranfragen übertragen. Aus einer logischen Perspektive spielt die Reihenfolge von Klauseln für den Wahrheitswert keine Rolle. Vom prozeduralen Standpunkt her kann sie für einen erfolgreichen Beweis von entscheidender Bedeutung sein. Die Klauselreihenfolge hat sehr großen Einfluss auf den Verlauf des Prolog-Beweises.

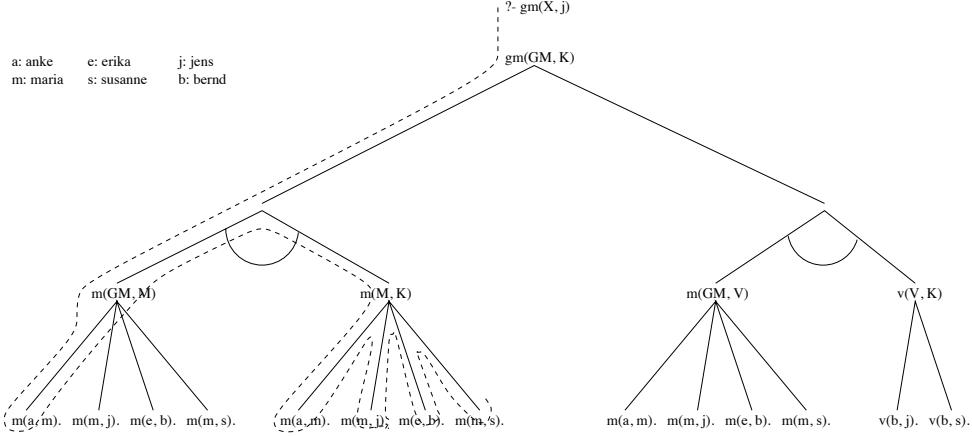
### 5.3.2 Beispielbeweis

Als illustrierendes Beispiel stellen wir die Anfrage `grossmutter(X, jens)..`  
Unser `grossmutter`-Beweisbaum dient dabei als Schablone.

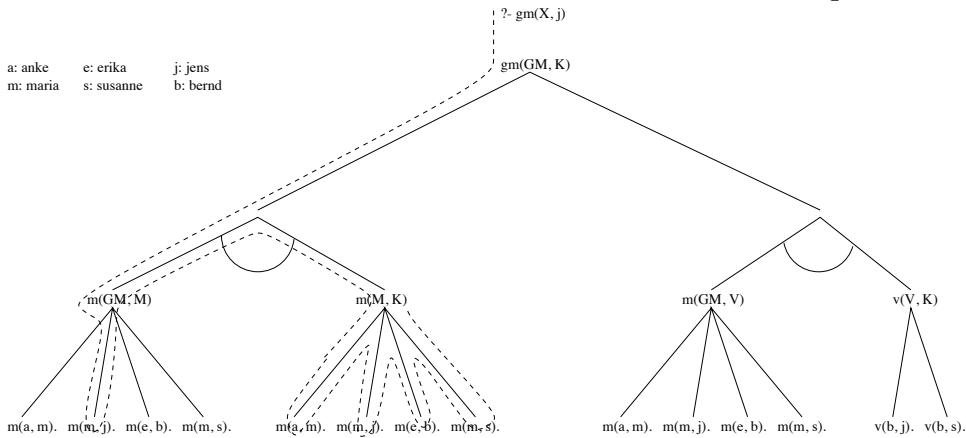


### Suche nach weiteren Lösungen

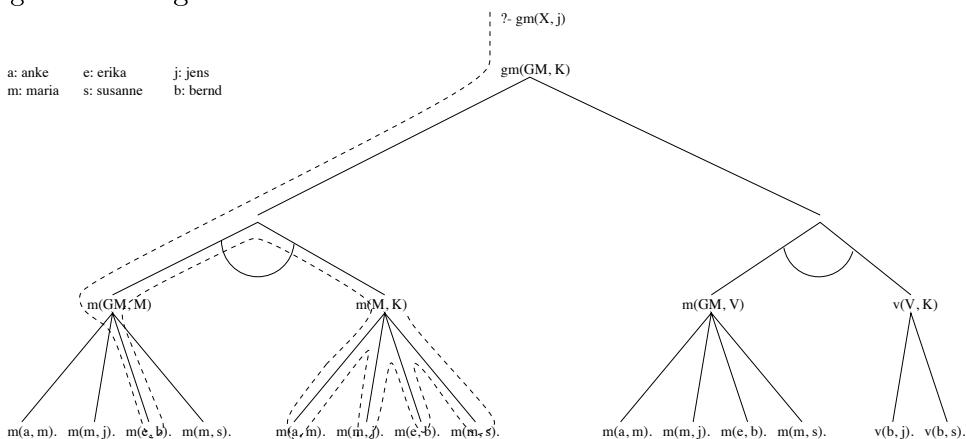
Erläuterungen jeweils unter den Grafiken.



Der Beweiser probiert nun nacheinander Alternativen für das zweite Subgoal.

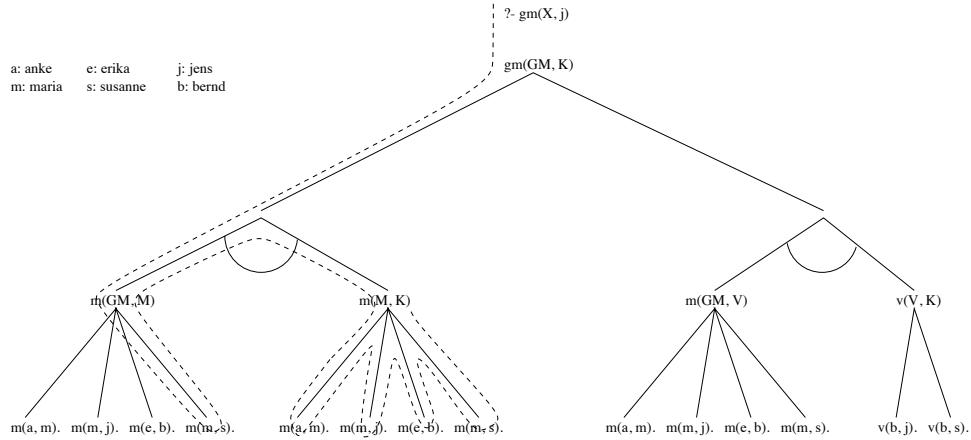


Der Beweiser wählt eine neue Alternative für das erste Subgoal und versucht wiederum das zweite Subgoal zu beweisen. Alle Alternativen für das zweite Subgoal sind erfolglos.

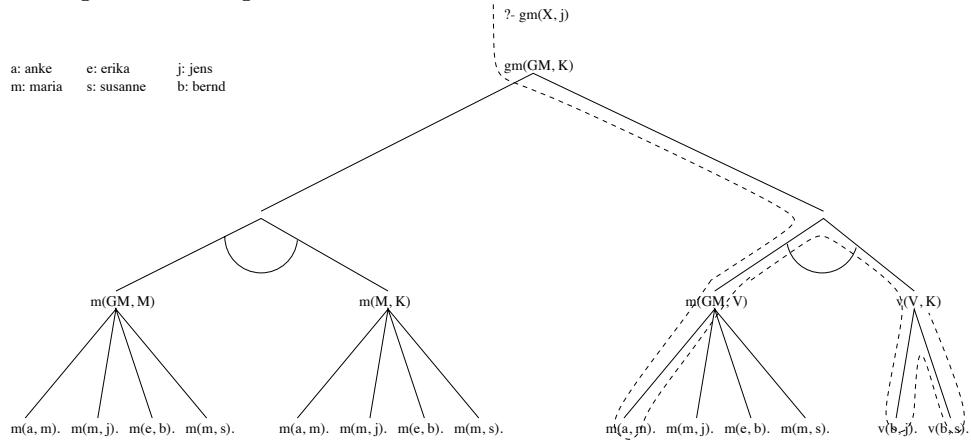


Analog zum obigen Schritt wird erneut eine Alternative für das erste Sub-

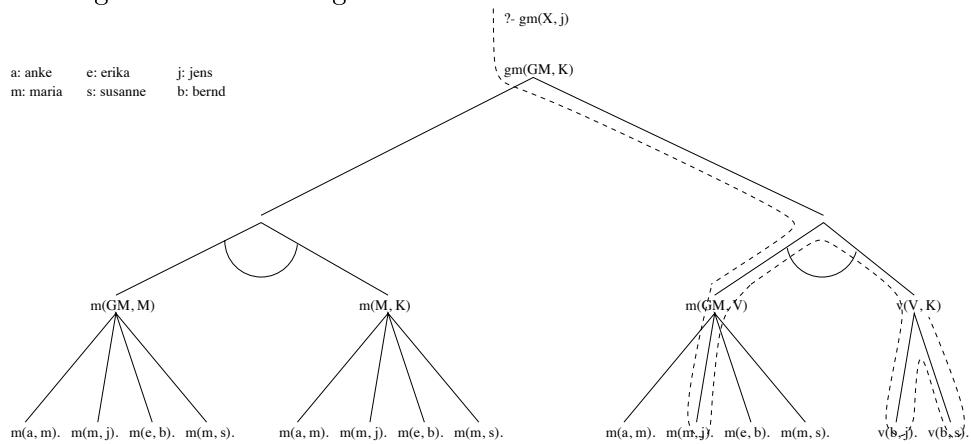
goal gewählt. Wieder lässt sich keine der Alternativen für das zweite Subgoal beweisen.



Analog zum vorherigen Schritt.

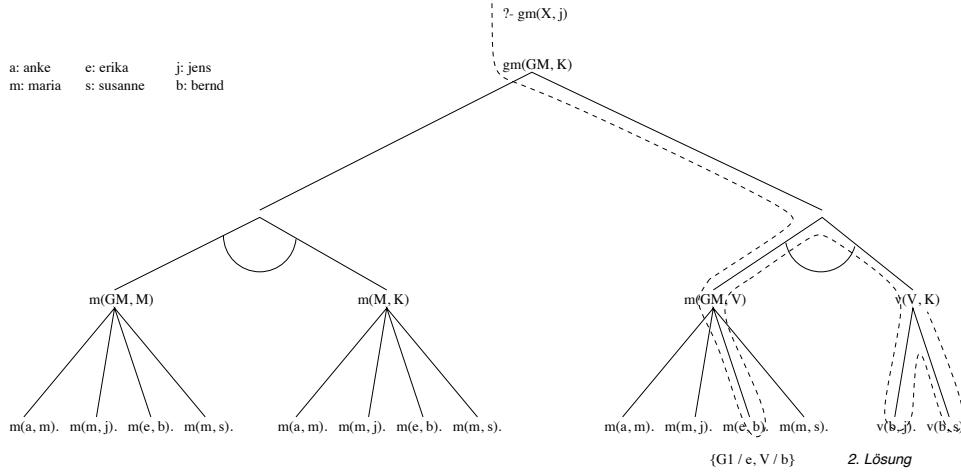


Schließlich gibt es keine Alternativen für das erste Subgoal mehr. Der Beweiser wählt eine Alternative für das Goal. Dieses hat zwei Subgoals. Für das erste Subgoal wird der erste Match genommen. Keine der Alternativen für das zweite Subgoal führt zum Erfolg.

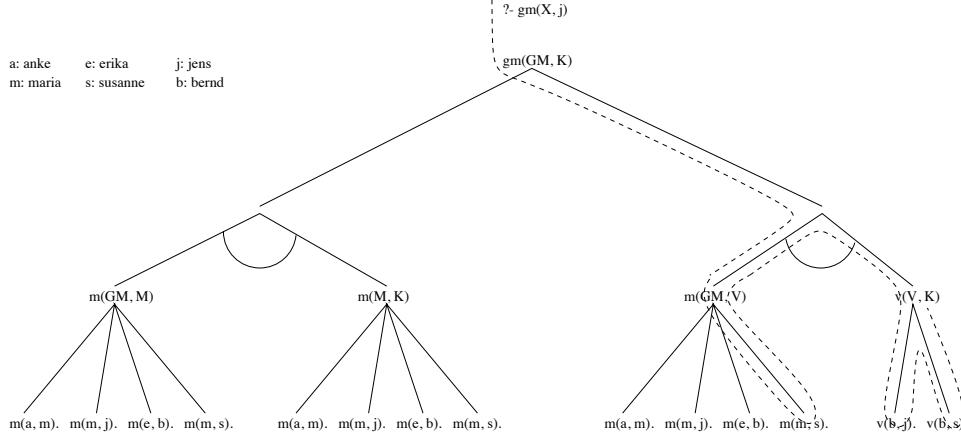


Die zweite Alternative für das erste Subgoal wird ausgesucht. Keine der

Alternativen für das zweite Subgoal ist erfolgreich.



Das System wählt die dritte Alternative für das erste Subgoal. Die erste Alternative des zweiten Subgoals führt zu einer Lösung. Die zweite Alternative des zweiten Subgoals führt nicht zu einer Lösung.



Die vierte Alternative des ersten Subgoals wird gewählt. Alle Alternativen des zweiten Subgoals scheitern. Alle Alternativen des ersten Subgoals sind gescheitert. Es gibt keine weiteren Alternativen für das Goal. Das System antwortet mit no.

### 5.3.3 Indexing

In den meisten Prologsystemen wird die Wissensbasis in Tabellen abgelegt, die den Zugriff auf Prädikate beschleunigen. So müssen auf der Suche nach einer matchenden **vater**-Klausel z.B. nur die **vater**-Klauseln, nicht aber die gesamte Wissensbasis durchsucht werden. Funktor und Stelligkeit fließen in den Suchschlüssel mit ein. Eine weitere Optimierung besteht darin, auch den Funktor des ersten Arguments, bzw. das erste Argument zur Indizierung heranzuziehen. Dadurch kann die Häufigkeit des Backtrackings reduziert werden, indem Alternativen, die zwar gleichen Funktor und gleiche Stelligkeit besitzen, aber *offensichtlich* nicht mit der Anfrage unifizierbar sind ausser Acht gelassen werden. Allerdings funktioniert dies nur wenn die ersten Argumente der Klauseln

und der Anfrage instantiiert sind.

Indexing kann nicht Vorhersagen, ob Klauseln scheitern oder gelingen. Auch kann Indexing nicht vorhersagen, ob die gesamte Unifikation gelingen wird.

## 5.4 Ablaufverfolgung / Trace

Die Trace des Prologsystems zu der oben verwendeten Anfrage `grossmutter(X, jens).` lautet:

```
?- spy(grossmutter).
% Spy point on grossmutter/2

Yes
[debug] ?- grossmutter(X, jens).

T Call: (6) grossmutter(_G378, jens)
    Call: (6) grossmutter(_G378, jens) ? creep
    Call: (7) mutter(_G378, _G430) ? creep
    Exit: (7) mutter(anke, maria) ? creep
    Call: (7) mutter(maria, jens) ? creep
    Exit: (7) mutter(maria, jens) ? creep
T Exit: (6) grossmutter(anke, jens)
    Exit: (6) grossmutter(anke, jens) ? creep

X = anke ;
Redo: (7) mutter(maria, jens) ? creep
    Fail: (7) mutter(maria, jens) ? creep
    Redo: (7) mutter(_G378, _G430) ? creep
    Exit: (7) mutter(maria, jens) ? creep
    Call: (7) mutter(jens, jens) ? creep
    Fail: (7) mutter(jens, jens) ? creep
    Redo: (7) mutter(_G378, _G430) ? creep
    Exit: (7) mutter(erika, bernd) ? creep
    Call: (7) mutter(bernd, jens) ? creep
    Fail: (7) mutter(bernd, jens) ? creep
    Redo: (7) mutter(_G378, _G430) ? creep
    Exit: (7) mutter(maria, susanne) ? creep
    Call: (7) mutter(susanne, jens) ? creep
    Fail: (7) mutter(susanne, jens) ? creep
T Redo: (6) grossmutter(_G378, jens)
    Redo: (6) grossmutter(_G378, jens) ? creep
    Call: (7) mutter(_G378, _G430) ? creep
    Exit: (7) mutter(anke, maria) ? creep
    Call: (7) vater(maria, jens) ? creep
    Fail: (7) vater(maria, jens) ? creep
    Redo: (7) mutter(_G378, _G430) ? creep
    Exit: (7) mutter(maria, jens) ? creep
    Call: (7) vater(jens, jens) ? creep
    Fail: (7) vater(jens, jens) ? creep
```

```

Redo: (7) mutter(_G378, _G430) ? creep
Exit: (7) mutter(erika, bernd) ? creep
Call: (7) vater(bernd, jens) ? creep
Exit: (7) vater(bernd, jens) ? creep
T Exit: (6) grossmutter(erika, jens)
Exit: (6) grossmutter(erika, jens) ? creep

X = erika ;
    Redo: (7) vater(bernd, jens) ? creep
    Fail: (7) vater(bernd, jens) ? creep
    Redo: (7) mutter(_G378, _G430) ? creep
    Exit: (7) mutter(maria, susanne) ? creep
    Call: (7) vater(susanne, jens) ? creep
    Fail: (7) vater(susanne, jens) ? creep
T Fail: (6) grossmutter(_G378, jens)
Fail: (6) grossmutter(_G378, jens) ? creep

```

No

Diese Ablaufverfolgung lässt sich als Protokoll der Traverse des Beweisbaumes auffassen. Natürlich werden in der Trace Optimierungen des Systems deutlich. Offensichtlich nicht matchende `mutter`-, und `vater`-Klauseln wurden vom System ausser Acht gelassen. Man mag die Trace neben den oben vorgestellten Und-/Oder-Baum legen und die beiden Darstellungsformen vergleichen.

#### 5.4.1 Das Vierportmodell

Die Ablaufverfolgung stützt sich auf das sogenannte “Vierportmodell”. Prädikate können von den vier folgenden Ereignissen betroffen sein:

CALL (Teil-)Beweis wird angestossen

EXIT (Teil-)Beweis erfolgreich beendet

FAIL (Teil-)Beweis ist gescheitert

REDO Wiederholung eines (Teil-)Beweises — dies führt im Allgemeinen zu einer alternativen Variablenbelegung

In der Trace wird zusätzlich zu aktuellem Ereignis und betroffenem Prädikat die sogenannte Aufrufebene ausgegeben. Ein Vergleich der Ereignisse CALL und EXIT gibt Aufschluss über geschehene Instantiierungen.

## 5.5 Prozedurales Programmieren in Prolog

In Abschnitt 4.3.2 auf Seite 65 wurde Ein-/Ausgabe in Prolog berührt. Man muss wissen, zu welchem Zeitpunkt Ein-/Ausgabe stattfindet. Aus diesem Grund benutzt Prolog die oben beschriebene Beweisprozedur. Sie macht den Kontrollfluss nachvollziehbar.

## 5.6 Implikationen des Prologbeweisverfahrens

To do ...

### 5.6.1 Endlosschleifen

aufgrund der Klauselreihenfolge

durch transitive Relationen

### 5.6.2 Reihenfolgeprobleme

(altes Skript)

```
? element(5, L),
  element(7, L),
  eq(L, [_, _, _, _, _]).
```

Listet *nicht* alle 4-elementigen Listen auf, die mindestens eine 5 und eine 7 enthalten!

### 5.6.3 Programmierprinzipien

(altes Skript)

1. Spezifischere Informationen vor allgemeineren Informationen! Dies führt zu kleineren Suchräumen. Einige Probleme mit unendlichen Lösungsräumen lassen sich so vermeiden.

## 5.7 alternative Beweisstrategien

Man kann sich alternative Beweisverfahren ausdenken, die in bezug auf die Arbeitsreihenfolge pfiffiger vorgehen. Damit könnten beispielsweise einige Situationen, die leicht zu Endlosschleifen führen entschärft werden. Allerdings ist der Preis dafür in der Regel der Verlust einer prozeduralen Interpretierbarkeit der Programme. Es ist nicht mehr vorhersehbar, wann die Auswertung bestimmter Prädikate stattfindet. Dies führt z.B. zu Problemen im Bereich der Ein- und Ausgabe.

### 5.7.1 Beispiele

Statt einer Tiefensuche könnte eine Breitensuche verwendet werden. Es wäre auch denkbar, Klauseln mit Wahrscheinlichkeiten oder anderen “Gütekriterien” zu belegen, die die Auswahl beeinflussen. Zum einen braucht man Informationen über Suchstrategien, zum anderen über sogenannte *Metainterpreter*. Metainterpreter sind Prologprogramme, die Prologprogramme verarbeiten und dabei z.B. eine alternative Beweisstrategie fahren können.

## 5.8 Das Prädikat **fail**

Mit dem Prädikat **fail** stellt Prolog ein Mittel zur Verfügung, mit dem Backtracking ausgelöst werden kann. **fail** bedeutet: “Dieser Beweis ist gescheitert – versuche eine Alternative!”

### 5.8.1 Backtrackingschleifen

Neben rekursiven Schleifen gibt es die Möglichkeit, Schleifen über Backtracking zu realisieren.

Eine Backtrackingschleife, die alle Lösungen einer Anfrage ausgibt, kann folgendermaßen realisiert werden:

```
?- member(_X, [1,2,3,4,5,6,7]), write(_X), nl, fail.  
1  
2  
3  
4  
5  
6  
7
```

No

Da **write** und **nl** keine Choicepoints darstellen, wirkt sich **fail** direkt auf **member(\_X, [1,2,3,4,5,6,7])** aus.

Die Prädikate **write** und **nl** wurden in Abschnitt 4.2.7 auf Seite 60 vorgestellt.

In Backtrackingschleifen ist mit den bisher vorgestellten Mitteln kein Informationstransport zwischen den Einzelnen “Aufrufen” möglich. Es gibt jedoch Prädikate zur Änderung der Wissensbasis, die zusammen mit Backtrackingschleifen eingesetzt werden können und in Kapitel 11 auf Seite 157 vorgestellt werden.

Das eingebaute Prädikat **findall/3**, das alle Lösungen eines Prädikates in einer Liste aufsammelt, arbeitet nach derselben Idee. (Es nimmt dynamische Änderungen an der Wissensbasis vor, um die Informationen nicht zu vergessen. Siehe Abschnitt 11.1.2 auf Seite 159.)

Wie oft diese Schleifenkonstruktion durchlaufen wird, hängt davon ab, wieviele Alternativen das betreffende Prädikat bietet. Das folgende Prädikat **wiederhole** stellt unendlich viele Choicepoints zur Verfügung. Es gibt ein entsprechendes Built-in namens **repeat**, das ebenfalls unendlich viele Choicepoints zur Verfügung stellt. Das Prädikat **bis** gelingt, wenn seine beiden Argumente uniformierbar sind, andernfalls scheitert es und löst damit Backtracking aus, es dient also als Bedingung für die Schleife.

```
wiederhole(_).  
wiederhole(_X) :- wiederhole(_X).  
  
bis(X,X).  
bis(X,Y) :- not(X=Y), fail.
```

Ein Verwendungsbeispiel:

```
?- wiederhole(_, nat_zahl(_X), write(_X), nl, bis(_X, s(s(s(s(0)))))).  
0  
s(0)  
s(s(0))  
s(s(s(0)))  
s(s(s(s(0))))
```

Yes

Fordert man weitere Lösungen an, so gerät der Beweiser in eine Endlosschleife, da die Abbruchbedingung in diesem Beispiel nicht geeignet formuliert ist. (Unter Verwendung eines sogenannten Cuts lässt sich das Problem beheben.)

## 5.9 Der Cut

Die einzige explizite Kontrollstruktur in Prolog ist **!** – der sogenannte *Cut*. Der Cut (**!**) schneidet alle verbleibende Alternativen ab für alle Goals, die vor dem Cut auftreten. Für die Goals hinter dem Cut bleibt alles normal. Damit werden ganze Teile des Beweisbaumes entfernt. Das Prädikat **!** ist immer wahr.

Ein Beispiel:

```
a :- b, c, !, d, e.  
a :- f.
```

Um **a** zu beweisen, gibt es zwei alternative Klauseln. Die erste enthält einen Cut nach den Subgoals **b** und **c**. Gelingen diese beiden Subgoals, so wird der Cut überschritten. Für den Beweiser gibt es kein Backtracking über diesen Punkt hinaus. Für **d** und **e** ist Backtracking möglich. Scheitern **d** und **e**, scheitert der gesamte Beweis. Die zweite Klausel wird nicht ausgewählt!

Scheitert jedoch **b** oder **c** so wird der Cut nicht überschritten, das System führt Backtracking durch und versucht die zweite Klausel für **a** zu beweisen.

### 5.9.1 “guarded gate”-Metapher

Dieses Verhalten führt zu der sogenannten “guarded gate”-Metapher: Im Klau-selrumpf stehen vor dem Cut Bedingungen, die erfüllt sein müssen, um den Cut zu überschreiten. Diese Bedingungen kann man sich als Torwächter vorstellen, der nicht jeden durch das Tor passieren lässt. Das Tor ist der Cut. Es fällt hinter dem, der es passiert, ins Schloss und verhindert jede Rückkehr. D.h. sobald der Beweiser einen Cut überschritten hat, führt kein Backtracking mehr vor den Cut zurück.

### 5.9.2 Beispiel

Wir wollen ein Prädikat definieren, das den Typ seines Argumentes ausgibt. Ein Prädikat, das wahr ist, wenn sein Argument eine Liste ist, haben wir im Kapitel über rekursive Programme erstellt.

## 5.10. **CUT**-**FAIL** KOMBINATIONEN ZUR SIMULATION VON NEGATION81

```
typ(X) :- is_list(X), !, write('Liste'), nl.  
typ(_) :- write('keine Liste...'), nl.
```

Wird **typ** mit einer Liste aufgerufen, so greift die Bedingung in der ersten Klausel, der Cut wird überschritten, eine Meldung ausgegeben und der Beweis ist gelungen. Alternativen gibt es nicht. Entfernt man den Cut, gibt es eine Alternative, das Ergebnis ist, dass zuerst behauptet wird, das Argument sei eine Liste, und als Alternative wird behauptet, dass es eben keine Liste sei. Der Cut verhindert dieses unerwünschte Verhalten. Ist das Argument tatsächlich keine Liste, so wird der Cut nicht überschritten. Es findet Backtracking statt und die zweite Klausel wird bewiesen. Diese ist immer beweisbar. Nach dem Vorbild der zweiten Klausel kann man den “default”-Fall aufschreiben, der in Kraft tritt wenn alle vorigen Klauseln scheitern.

Prolog verfügt über weitere eingebaute Prädikate aus diesem Aufgabenbereich. Siehe dazu Abschnitt 5.12 auf Seite 84.

So können wir das Prädikat beliebig erweitern, z.B.:

```
typ(X) :- is_list(X), !, write('Liste'), nl.  
typ(X) :- atom(X), !, write('Atom'), nl.  
typ(_) :- write('weder Atom noch Liste...'), nl.
```

### 5.9.3 “rote” und “grüne” Cuts

Pfade durch einen Beweisbaum können entweder zu Erfolg oder zu Scheitern führen.

- Schneidet man nur Teile eines Beweisbaumes ab, die keine Lösung liefern, ändert sich die Bedeutung (Semantik) des Programmes nicht.  
Cuts, die die Semantik eines Programmes nicht ändern, werden als *grüne Cuts* bezeichnet.  
Sie können dazu eingesetzt werden, die Effizienz von Prologprogrammen zu steigern, indem sie den Beweiser von überflüssiger Arbeit abhalten.
- Schneidet man Teile des Beweisbaumes ab, die zu möglichen Lösungen führen, so ändert man die Semantik des Programmes. Ein Cut, der mögliche weitere Lösungen verhindert, wird als *roter Cut* bezeichnet. Die Cuts im Beispiel in 5.9.2 sind rot.

## 5.10 **cut**-**fail** Kombinationen zur Simulation von Negation

Man kann sich Prädikate vorstellen, die unter bestimmten Bedingungen sofort und ohne Alternativen scheitern sollen. Solche Konstruktionen kann man mit **cut**-**fail** Kombinationen realisieren.

Ein Beispiel: Das Prädikat **schmeckt\_kindern\_gut(X)** soll Eltern bei der Auswahl von Speisen für ihre Kinder helfen. An dieser Stelle fliesst Expertenwissen ein:

Ein Rosenkohl hat keine Chance.

In Anbetracht dieser ‘‘Tatsache’’ soll das Prädikat alle Rosenkohle zurückweisen.

```
schmeckt_kindern_gut(X) :- rosenkohl(X), !, fail.
schmeckt_kindern_gut(X) :- suess(X), bunt(X), ungesund(X).
```

```
rosenkohl(rosenkohl).
```

```
suess(gummibaerchen).
bunt(gummibaerchen).
ungesund(gummibaerchen).
```

Ein Test mit konkreten Anfragen:

```
?- schmeckt_kindern_gut(rosenkohl).
No
```

```
?- schmeckt_kindern_gut(gummibaerchen).
Yes
```

An Stelle von `cut`-`fail` Kombinationen lässt sich das Builtin `not/1` verwenden:

```
schmeckt_kindern_gut2(X) :- not(rosenkohl(X)),
                           suess(X), bunt(X),
                           ungesund(X).
```

Beide Versionen des Prädikates verhalten sich gleich. Das Ergebnis der folgenden Anfrage macht deutlich, dass die (simulierte) Negation in Prolog nicht der logischen Negation entspricht. Sie ist mit Vorsicht zu genießen, da sie nicht unbedingt intuitiv ist. (Siehe ‘‘Closed-World-Assumption’’ im Kapitel über Logik, 6.5.3 auf Seite 90)

So liefert das System z.B keine Belegung für `X` in der Anfrage:

```
?- schmeckt_kindern_gut(X).
No
```

obwohl es in der Wissensbasis `gummibaerchen` gibt! Das liegt daran, dass der Beweiser in der ersten Klausel des Prädikates versucht das Subgoal `rosenkohl(X)` zu beweisen. Da es einen Rosenkohl gibt, wird `X` mit `rosenkohl` instantiiert. Die beiden folgenden Subgoals `!, fail.` bringen den Beweisversuch mit Misserfolg zu Ende. Der Beweiser kann kein Backtracking durchführen und so die zweite Klausel nicht zur Anwendung bringen, um damit die Bärchen rauszugeben. Uninstantiierte Variablen stellen in diesem Kontext ein Problem dar!

### 5.10.1 Das Prädikat `not`

Das Prädikat `not(Goal)` scheitert, wenn `Goal` als Anfrage gelingt. Gelingt `Goal` nicht, so gelingt `not(Goal)`. Es ist folgendermaßen definiert:

```
not(Goal) :- call(Goal), !, fail.
not(_).
```

## 5.10. [CUT]-[FAIL] KOMBINATIONEN ZUR SIMULATION VON NEGATION83

[call/1] interpretiert sein Argument als Anfrage. In der ersten Klausel wird also versucht, [Goal] zu beweisen. Gelingt dieser Beweis, wird der nachfolgende Cut überschritten. Das anschliessende [fail] bringt den Beweisversuch zu ende. Die zweite Klausel steht in diesem Fall nicht mehr als Alternative zur Verfügung. Damit scheitert [not], wenn [Goal] gelingt.

Scheitert allerdings [call(Goal)], wird der Cut nicht überschritten. Somit steht die zweite Klausel als Alternative zur Verfügung. Diese gelingt immer.

Bei diesem Verfahren handelt es sich um *Negation as Failure*. Dies entspricht nicht der logischen Negation und verhält sich nicht so wie wir es vielleicht intuitiv erwarten.

Probleme mit der Negation-as-Failure sind unter anderem:

- uninstantiierte (ungebundene) Variablen.  
(Beispiel: 5.10)
- Negation kann nicht im Kopf von Regeln stehen.  
(Siehe Hornlogik: 6.5.2, Seite 89)

Weitere Informationen finden sich im Kapitel über Prolog vs. PL1 auf Seite 85.  
(Insbesondere *Closed-World-Assumption* 6.5.3, Hornlogik 6.5.2)

Zum Abschluss eine kleine Erinnerung:

```
mensch(hans).  
?- not(mensch(maria)).  
Yes
```

Die Antwort [Yes] bedeutet nicht etwa, dass [maria] kein [mensch] ist, sondern lediglich, dass es dem Prologsystem nicht gelungen ist, die Aussage aus der Wissensbasis abzuleiten, dass [maria] ein [mensch] ist.

### 5.10.2 weitere Aspekte des Cuts

- Der Cut kann Programme effizienter aber auch schwerer nachvollziehbar machen.
- Von der Verwendung allzuvieler Cuts wird im Allgemeinen abgeraten.
- Aus logischer Sicht sind Cuts natürlich unangenehm. Prozedurale und deklarative Interpretationen können sich wesentlich unterscheiden.
- Cuts ermöglichen deterministische Prädikate, d. h. Prädikate, die nur eine Lösung liefern.
- [!] und [fail] ermöglichen Konstruktionen, die IF-THEN-ELSE aus anderen Sprachen entsprechen.

## 5.11 Prädikate kommentieren II – Konventionen + - ?

Logisch saubere Prädikate funktionieren mit allen erdenklichen Kombinationen instantiiierter und uninstantiiierter Argumente. Viele Prädikate verlassen diesen Rahmen und müssen diese Freiheit daher erheblich einschränken. In der Dokumentation solcher Prädikate wird jedes Argument hinsichtlich seiner “Freiheit” markiert:

- Ein + vor dem Argument bedeutet, dass das Argument instantiiert sein muss.
- Ein - vor dem Argument zeigt an, dass das Argument eine Variable (instantiiert / uninstantiiert) sein darf. Falls die Anfrage beweisbar ist, existiert genau eine Variablenbelegung. (deterministisch)
- Ein ? vor dem Argument erklärt, dass es sich bei dem Argument um eine (instantiierte/uninstantiierte) Variable handeln darf und dass es eine oder mehrere Variablenbelegungen geben kann, die einen erfolgreichen Beweis ermöglichen. (nichtdeterministisch)

Sollte es mehrere Möglichkeiten geben, ein solches Prädikat aufzurufen, sollten alle Möglichkeiten kommentiert werden. Im SWI-Manual wird diese Notation verwendet.

## 5.12 Typprädikate

Typprädikate erlauben es, herauszufinden, ob es sich bei einem Term z.B. etwa um ein Atom, eine Variable oder eine Zahl handelt. Unter anderem gibt es folgende Typprädikate:

- `var(+Term)` : `Term` ist zum Anfragezeitpunkt eine Variable.
- `nonvar(+Term)` : `Term` ist zum Anfragezeitpunkt keine uninstantiierte Variable.
- `atom(+Term)` : `Term` ist zum Anfragezeitpunkt ein Atom.
- `integer(+Term)` : `Term` ist zum Anfragezeitpunkt eine ganze Zahl.
- `float(+Term)` : `Term` ist zum Anfragezeitpunkt eine Fließkommazahl.
- `atomic(+Term)` : `Term` ist zum Anfragezeitpunkt Atom, ganze, oder Fließkommazahl.

Weitere Informationen finden sich im SWI-Manual. Diese Prädikate kann man beispielsweise dazu nutzen, sicherzustellen, dass eigene Prädikate mit instantiierten/uninstantiierten Argumenten aufgerufen werden.

# Kapitel 6

## Die Beziehung zwischen Prolog und der PL1

In diesem Kapitel werden wir die enge Verwandtschaft von mathematischer Logik und Prolog untersuchen. Dies ist zwar für das Programmieren nicht zwingend notwendig, gibt aber einen Eindruck darüber, wieso Prolog so funktioniert, wie wir es bisher angewendet haben.

So ist die Syntax von Prolog nicht willkürlich gewählt worden, sondern entspricht der Klauselform der Prädikatenlogik erster Stufe (PL1). Im folgenden werden wir daher die Syntax und Semantik von PL1 betrachten. \*[PL1, Klauselform]

### 6.1 Einführung

Die Prädikatenlogik geht davon aus, daß die Welt aus Objekten besteht, die bestimmte Eigenschaften haben und zueinander in Beziehungen oder Relationen stehen. All dies können wir in Prolog und PL1 auf dieselbe Weise ausdrücken: \*[Objekte]

- Objekte entsprechen Termen, also beispielsweise Atomen wie `maria` oder `john`
- Eigenschaften können wir als einstellige Prädikate darstellen, wie z.B. `frau(maria)`
- Relationen entsprechen mehrstelligen Prädikaten wie `vater(john, maria)`

### 6.2 Syntax der PL1

Die Syntax der Prädikatenlogik erster Stufe besteht aus:

- Konstanten. Sie werden anders als in Prolog in der PL1 meist groß geschrieben.
- Variablen. In diesem Kapitel werden wir sie mit x, y, z bezeichnen.
- Funktionen. Sie sind nicht wahrheitswertfähig und liefern zu jedem Argument genau einen Wert. Z.B. `vater(Maria) = John`

- Prädikaten. Sie sind wahrheitswertfähig, liefern also abhängig von ihren Argumenten wahr oder falsch. Ein Prädikat kann Terme - also Konstanten, Variablen und Funktionen - als Argumente besitzen.

\*[Quantoren]

\*[Junktoren, Formeln]

Quantoren und Junktoren schauen wir uns im folgenden noch etwas genauer an.

### 6.2.1 Quantoren

Schauen wir uns den Ausdruck `vater(x, Hans)` an. Obwohl `vater` ein Prädikat ist, können wir dem Ausdruck keinen Wahrheitswert zuweisen. Das liegt daran, daß die Variable `x` keinen Wert besitzt. Sie ist ungebunden oder auch frei. Ausdrücke werden zu wahrheitswertfähigen Formeln, indem wir alle Variablen durch Quantoren binden:

$$\begin{aligned} \exists x : vater(x, Hans) \\ \forall x : vater(x, Hans) \end{aligned}$$

Sobald jede Variable gebunden ist, können wir entscheiden, ob die Formel wahr ist oder nicht. Die erste Formel drückt aus, daß mindestens ein Objekt in unserem Universum existiert, das der Vater von demjenigen Individuum ist, das wir mit `Hans` bezeichnet haben. Im zweiten Fall ist jedes Objekt unseres Universums der Vater von `Hans`. Ob die Aussagen wahr sind, hängt davon ab, welche Objekte in unserem Universum sind, und welche Relationen zwischen ihnen bestehen.

In Prolog gibt es keine Quantoren. Wie wir bereits gelernt haben, sind in Prolog Variablen in Klauseln implizit allquantifiziert und in Anfragen implizit existenzquantifiziert. Es kommen also keine ungebundenen Variablen vor. Deshalb sind alle Klauseln und Anfragen automatisch Formeln.

### 6.2.2 Junktoren

Junktoren verknüpfen Formeln zu neuen komplizierteren Formeln. Wenn A und B Formeln sind, dann sind die folgenden Ausdrücke auch Formeln<sup>1</sup>:

Bezeichnung:	Symbol:	Entsprechung in Prolog:
A UND B	$A \wedge B$	A, B
A ODER B	$A \vee B$	A; B
A IMPLIZIERT B	$A \rightarrow B$	$B :- A$
A ist AEQUIVALENT zu B	$A \leftrightarrow B$	nicht möglich
NICHT A	$\neg A$	<code>not(A)</code>

Es ist wichtig, daß das `not/1` in Prolog nicht dem logischen NICHT entspricht, sondern als Negation-As-Failure realisiert ist. Für diesen Unterschied haben wir ein eigenes Kapitel reserviert. Wieso die logische Negation in Prolog nicht möglich ist, werden wir aber schon etwas weiter unten verstehen.

---

<sup>1</sup>Einige Bücher verlangen Klammern um zusammengesetzte Formeln. Wir verzichten zu gunsten der besseren Lesbarkeit auf diese Konvention.

## 6.3 Semantik der PL1

### 6.3.1 Junktoren

Die Semantik der Junktoren wollen wir der Einfachheit halber mit Mitteln der Aussagenlogik, nämlich durch Wahrheitstafeln, bestimmen. Eine Wahrheitstafel enthält für jede enthaltene Formel eine Spalte, also für jede einzelne atomare Formel und sukzessive für jede zusammengesetzte Formel. Die Spalten der atomaren Formeln werden so gefüllt, daß alle Kombinationen von wahr und falsch vorkommen:

A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$	$\neg A$
wahr	wahr	wahr	wahr	wahr	wahr	falsch
wahr	falsch	falsch	wahr	falsch	falsch	falsch
falsch	wahr	falsch	wahr	wahr	falsch	wahr
falsch	falsch	falsch	falsch	wahr	wahr	wahr

## 6.4 Transformationen

Sachverhalte lassen sich in der Prädikatenlogik auf viele verschiedene Weisen darstellen, ohne daß sich ihr Wahrheitsgehalt oder Aussage ändert. Beispielsweise gibt es die folgenden einfachen Transformationen, die eine Formel in eine äquivalente andere Formel überführen. Äquivalent soll hierbei bedeuten, daß die Formeln denselben Wahrheitsgehalt besitzen.

\*[Transformationen]

Formel 1	Formel 2
$A \leftrightarrow B$	$(A \rightarrow B) \wedge (B \rightarrow A)$
$A \rightarrow B$	$\neg A \vee B$
$A \wedge B$	$\neg(\neg A \vee \neg B)$
$A \vee B$	$\neg(\neg A \wedge \neg B)$
$A \vee (B \wedge C)$	$(A \vee B) \wedge (A \vee C)$
$A \wedge (B \vee C)$	$(A \wedge B) \vee (A \wedge C)$
$\forall x : A$	$\neg \exists x : \neg A$
$\exists x : A$	$\neg \forall x : \neg A$

Die zugehörigen Formeln sind also nur unterschiedliche Schreibweisen desselben Sachverhalts.

Eine weitere wichtige Transformation, die oft durchgeführt wird, ist die Skolemisierung. Sie wird benutzt, um den Existenzquantor zu ersetzen. Ihre Hauptidee basiert darauf, daß eine existenzquantifizierte Formel wahr ist, wenn mindestens ein Objekt existiert, daß die Formel erfüllt. Wenn wir also dieses Objekt kennen und in die Formel einsetzen, ist sie wahr, und wir können uns den Existenzquantor sparen.

\*[Skolemisierung]

Ein Beispiel:

Sei unsere Ausgangsformel die folgende:

$$\exists x : \forall y : \text{mann}(y) \rightarrow \text{lieben}(y, x)$$

Oder natürlichsprachlich ausgedrückt: Es existiert eine Frau, die alle Männer lieben. Angenommen, wir kennen diese Frau und sie sei Helena, der Grund für den Trojanischen Krieg. Dann können wir die Formel modifizieren:

$$\forall y : \text{mann}(y) \rightarrow \text{lieben}(y, \text{Helena})$$

\*[Skolemkonstante]

So haben wir den Existenzquantor eingespart. Im praktischen Leben wird man die existenzquantifizierte Variable oft nicht so genau ersetzen können. In diesen Fällen behilft man sich mit einer Skolemkonstante, die man einfach einführt und von der man annimmt, daß sie in unserem Universum auf das gewünschte Objekt verweist.

Was passiert aber, wenn wir diese Vorgangsweise auf die folgende Formel anwenden:

$$\forall y : \exists x : \text{mann}(y) \rightarrow \text{lieben}(y, x)$$

Oder natürlichsprachlich ausgedrückt: Jeder Mann liebt irgendeine Frau. Nach der Skolemisierung, so wie wir sie bisher kennen, erhalten wir:

$$\forall y : \text{mann}(y) \rightarrow \text{lieben}(y, \text{Helena})$$

Dies drückt aber nicht mehr denselben Sachverhalt aus, den wir ursprünglich gemeint haben! Der Grund ist der, daß der Wert der Variable  $x$  vom jeweiligen Mann abhängt. Hier wird statt der Skolemkonstante also eine Skolemfunktion benötigt, die abhängig vom Mann die zugehörige geliebte Frau liefert:  $f_s(y)$

Die entsprechende komplette Formel sieht dann so aus:

$$\forall y : \text{mann}(y) \rightarrow \text{lieben}(y, f_s(y))$$

Wie man entscheiden kann, ob man eine Skolemkonstante oder -funktion benutzen muß, und wieviele Argumente letztere benötigt, werden wir etwas weiter unten betrachten. Dafür benötigen wir nämlich die Konjunktive Normalenform (KNF). Nebenbei werden wir dabei natürlich noch etwas über die Verwandtschaft von Prolog und PL1 lernen.

\*[Skolemfunktion]\*[Konjunktive Normalenform]

## 6.5 Konjunktive Normalenform

Die KNF ist eine Schreibweise für prädikaten- und aussagenlogische Formeln, die einfach darin besteht, daß mehrere Teilformeln durch UND verbunden werden (daher der Konjunktiv-Teil des Namens, analog dazu gibt es auch die Disjunktive Normalenform). Die Teilformeln bestehen wiederum aus optional negierten Literalen, die durch ODER verbunden sind. Ein Beispiel:

$$(A \vee \neg B \vee C) \wedge (B \vee D)$$

Oder formal:

$$\bigwedge_{i=1}^n \bigvee_{j=1}^{m_i} l_{i,j}$$

### 6.5.1 Umwandlung von Formeln in die KNF

Man kann zeigen, daß sich jede Formel in die KNF transformieren läßt. Den Beweis sparen wir uns, denn er ist nicht nötig, um die Transformationen durchzuführen. Sie laufen nach diesem Schema ab:

- Äquivalenzen auflösen
- Implikationen auflösen
- Negationen möglichst weit nach innen
- Quantoren nach außen

- Disjunktionen nach innen
- Konjunktionen nach außen
- Existenzquantoren durch Skolemisieren eliminieren
- Allquantoren weglassen

An dieser Stelle löst sich die Frage auf, wieviele Argumente die Skolemfunktionen haben müssen (Skolemkonstanten können wir als nullstellige Skolemfunktion ansehen). Rein pragmatisch kann man sich daran halten, daß die Stelligkeit der Anzahl der Allquantoren vor dem Existenzquantor entspricht. Deswegen ist es wichtig, die Reihenfolge der Quantoren nicht zu verändern, wenn man sie im Schritt 4 nach vorn zieht.

Das ganze wollen wir im nächsten Abschnitt an einem einfachen Beispiel verdeutlichen.

### 6.5.2 KNF und Prolog

Eine Prolog-Klausel wie

```
p(X) :-  
q(X),  
r(X).
```

entspricht in PL1:

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

Dies wollen wir in die KNF transformieren. Zunächst eliminieren wir mittels der oben vorgestellten Transformationen die Implikation:

$$\forall x : \neg(q(x) \wedge r(x)) \vee p(x)$$

Nun ziehen wir die Negation in die Klammer.

$$\forall x : \neg q(x) \vee \neg r(x) \vee p(x)$$

Nun können wir den Allquantor weglassen und haben eine Formel in KNF.

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

Wenn wir uns klarmachen, daß Prolog-Klauseln maximal einen Term im Regelkopf haben, ist es offensichtlich, daß die korrespondierende KNF-Formel maximal ein positives Literal enthalten kann. Dies ist nicht ohne Grund so, sondern entspringt der Tatsache, daß Prolog nur aus Hornklauseln (Hornklauseln) besteht. Dies sind eine Untergruppe der PL1 und zeichnen sich eben gerade dadurch aus, daß sie in der KNF-Schreibweise höchstens ein positives Literal enthalten. \*1/1

Wie sieht dies nun für Fakten und Anfragen/Queries aus? Erinnern wir uns, daß Fakten Regeln ohne Rumpf sind. Da die Terme im Rumpf negiert werden und der Term im Kopf der einzige positive Term in der KNF ist, ergibt sich, daß Fakten einfach genau ein positives Literal sind. Anfragen sind Regeln ohne Kopf. Folglich bestehen Queries komplett aus negierten Termen.

### 6.5.3 Closed-world-assumption

Bei der Übersetzung von PL1 nach Prolog kommt uns die Closed-World-Assumption zur Hilfe. Diese besagt, daß alles, was nicht explizit als wahr bewiesen werden kann, falsch ist. Dies scheint auf den ersten Blick in einer zweiwertigen Logik nicht überraschend zu sein. Wenn wir uns aber erinnern, daß man ein Prolog-Programm auch als Datenbank ansehen kann, ist diese Annahme eine sehr starke, da alles, was nicht in unserer Datenbank repräsentiert ist, nicht existiert. Diesen Umstand kann man bei der Repräsentation von Fakten ausnutzen. Nehmen wir an, wir wollen formulieren, daß es nur genau einen Kilimandscharo gibt. In PL1 sähe das so aus:

$$\exists x : (\text{kilimandscharo}(x) \wedge (\forall y : \text{kilimandscharo}(y) \rightarrow y = x))$$

Es reicht nicht zu sagen, daß es ein Objekt gibt, daß der Kilimandscharo ist. Man muß auch ausdrücken, daß nur dieses eine Objekt die Anforderung erfüllt, ein Kilimandscharo zu sein.

Durch die Closed-World-Assumption wird diese Aussage in Prolog sehr viel einfacher:

```
kilimandscharo(kili).
```

Eine Anfrage mit irgendeiner anderen Konstanten als kili wird nicht mat-chen und daher scheitern.

## 6.6 Resolution

Wir haben bisher gelernt, wie wir Formeln durch Transformationen äquivalent ausdrücken können. Aber erst das Schließen, oder auch Schlußfolgern oder Inferenz, macht Logik zu einem nützlichen Werkzeug. Hornklauseln machen die Inferenz, in Prolog also das Beweisen von Anfragen, sehr einfach und elegant. Zunächst wollen wir uns ein Inferenzverfahren der Prädikatenlogik anschauen, bevor wir dieses Verfahren in Prolog wiederfinden.

### 6.6.1 Logische Resolution für Aussagenlogik

\*[Inferenz]

\*[Resolution]

Die Resolution ist ein Verfahren, um zu zeigen, daß eine Formel aus anderen ableitbar ist. Sie beruht auf der Tatsache, daß in einer Formel wie  $(A \vee B) \wedge (\neg B \vee C)$  nicht sowohl  $B$  als auch  $\neg B$  wahr sein kann. Folglich muß  $A \vee C$  wahr sein. Somit haben wir eine neue Formel abgeleitet. Man kann sich diesen Vorgang auch klar machen, indem man die Klauseln in Implikationen transformiert. Daraus erhält man  $(\neg A \rightarrow B) \wedge (B \rightarrow C)$ . Durch die Transitivität der Implikation kann man nun  $\neg A \rightarrow C$  ableiten, was natürlich äquivalent zu  $A \vee C$  ist.

Statt Resolution sagt man auch, man schneidet zwei Formeln, da als Ergebnis eine Klausel entsteht, aus der die sich widersprechenden Literale herausgeschnitten sind.

Zwei grundlegende Begriffe in der Logik sind Kontradiktion und Tautologie. Erstere ist eine Formel, die niemals erfüllt werden kann, egal wie man die Aussagenprimitive wählt. Das einfachste Beispiel ist:  $A \wedge \neg A$ . Egal wie man A wählt (wahr oder falsch), die Formel ist stets falsch. Eine Tautologie dagegen ist immer erfüllbar, wie zum Beispiel  $A \vee \neg A$ .

\*[Kontradiktion]

\*[Tautologie]

Was passiert, wenn man diese Methode auf Kontradiktionen anwendet? Nehmen wir an, wir haben zwei Klauseln, die sich widersprechen, z.B.  $(\neg A \vee B) \wedge (A \vee \neg B)$ . Egal, wie wir  $A$  und  $B$  belegen, die Gesamtaussage ist falsch. Wenn wir die Resolution darauf an, d.h. schneiden wir die sich widersprechenden Literale, bleiben keine Literale übrig. Solch eine leere Formel zeigt an, daß sich die Ausgangsformeln widersprechen.

Mit dieser Methode kann man nun überprüfen, ob eine Formel aus einem Axiomensystem ableitbar ist. Axiome entsprechen Klauseln. Wenn die neue Formel mit unseren Axiomen übereinstimmt, dann muß ihre Negation zwangsläufig falsch sein. Dadurch ergibt sich ein einfacher Test. Fügen wir nämlich die Negation der Formel ein, muß sie nun im Widerspruch zu den Axiomen stehen. Das heißt, wir müssen die leere Formel ableiten können. Wenn wir nach einer Anzahl von Resolutionsschritten also die leere Formel erhalten, heißt dies, daß die Negation der eingefügten Formel im Widerspruch zu unseren Axiomen steht, die originale Formel also wahr ist.

### 6.6.2 Logische Resolution für Prädikatenlogik

In der Prädikatenlogik resolvieren man prinzipiell genauso wie in der Aussagenlogik. Es ist allerdings ein wenig komplizierter, da wir nun nicht nur einfache Aussagen miteinander schneiden, sondern Prädikate. Und diese können Variablen enthalten. Bisher war es einfach zu erkennen, daß wir beispielsweise  $A$  mit  $\neg A$  schneiden können. Aber wie sieht es mit  $\text{vater}(\text{Hans})$  und  $\text{vater}(x)$  aus? Hier hilft uns ein Verfahren weiter, das wir bereits kennengelernt haben, die Unifikation.

?[Unifikation]

Auch in der Logik gibt es die Unifikation. Um einen Term wie  $\text{vater}(\text{Hans})$  und  $\text{vater}(x)$  in denselben Ausdruck zu überführen, müssen wir eine sogenannte Substitution anwenden. Diese ersetzt eine Variable durch einen anderen Term, also beispielsweise eine Konstante oder auch eine andere Variable. In unserem Beispiel müssen wir  $x$  durch  $\text{Hans}$  ersetzen.

\*[Substitution]

Etwas formaler schreibt sich die Unifikation folgendermaßen:

Zwei Terme  $\phi$  und  $\psi$  sind dann unifizierbar, wenn es einen Unifikator  $\theta$  gibt, so daß  $\phi\theta = \psi\theta$ .

$\phi\theta$  bedeutet, daß  $\theta$  auf  $\phi$  angewendet wird. Ein Unifikator besteht aus Substitutionen, also ist beispielsweise  $\{x\backslash \text{Hans}\}$  der Unifikator von  $\text{vater}(\text{Hans})$  und  $\text{vater}(x)$ , und  $\{x\backslash \text{Hans}\}$  angewendet auf  $\text{vater}(x)$  ergibt  $\text{vater}(\text{Hans})$ .

Natürlich ist der Unifikator nicht immer eindeutig. Zum Beispiel kann man  $\text{vater}(x)$  und  $\text{vater}(y)$  durch  $\{x\backslash \text{Hans}, y\backslash \text{Hans}\}$ ,  $\{x\backslash \text{Peter}, y\backslash \text{Peter}\}$  oder  $\{x\backslash y\}$  unifizieren. Welcher Unifikator ist nun der geeignetste? Nehmen wir an, wir möchten nach der Unifikation von  $\text{vater}(x)$  und  $\text{vater}(y)$  das Ergebnis mit  $\text{vater}(\text{Paul})$  unifizieren. Wenn wir nun den ersten von den drei vorgeschlagenen Unifikatoren benutzt haben, können wir das Ergebnis nicht mehr mit dem neuen Term unifizieren, da Paul und Hans nicht matchen. Um so etwas zu vermeiden, benutzt man immer den allgemeinsten Unifikator, den sogenannten Most General Unifier, oder auch MGU. Das wäre in diesem Fall  $\{x\backslash y\}$ .

\*[MGU]

Wir sparen uns hier eine formale Definition des MGU, da ein intuitives Verständnis dieses Konzeptes für das weitere Vorgehen ausreicht.

### 6.6.3 Resolution in Prolog

\*[SLD-Resolution]

In Prolog wird eine besondere Form der Resolution benutzt, nämlich die SLD-Resolution. Diese ist notwendig, da wir von unserem Prolog-System nicht nur eine Ja/Nein-Antwort erwarten, sondern ggf. auch wissen möchten, wie die Variablen unserer Anfrage belegt sein müssen, damit sie erfüllt werden kann.

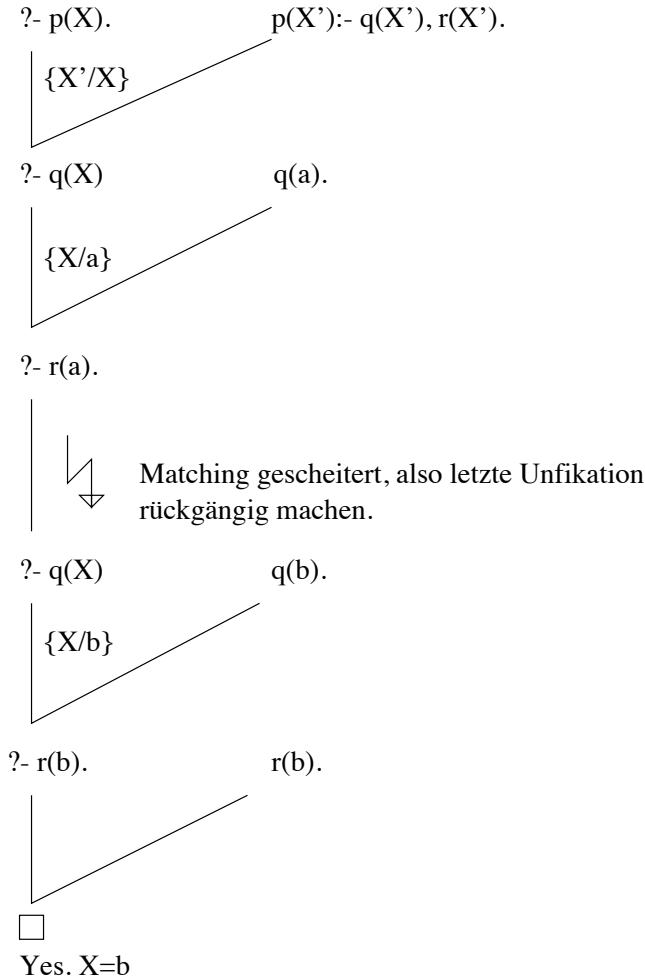
Eine Anfrage besteht aus einer Reihe UND-verknüpfter Formeln. Die logische Schreibweise  $Q_1 \wedge Q_2 \wedge \dots \wedge Q_n$  entspricht in Prolog  $Q_1, Q_2, \dots, Q_n$

Erinnern wir uns nun, daß dies einfach  $\leftarrow Q_1 \wedge Q_2 \wedge \dots \wedge Q_n$  entspricht. Was uns interessiert ist, ob diese Bedingung mit unserem Programm, also dessen Regeln und Fakten, bewiesen werden kann. Bei der Standard-Resolution kann man prinzipiell mit jeder Klausel anfangen zu resolvieren. Die SLD-Resolution resolviert jedoch nach einem festen Schema. Der erste Schritt besteht darin, die erste Klausel der Anfrage mit einer Klausel aus der Wissensbasis zu matchen, und zwar - wie wir natürlich schon wissen - in der Reihenfolge von oben nach unten. Dank der Tatsache, daß Prolog nur aus Hornklauseln besteht, ist das Finden von zugehörigen Literalpaaren sehr einfach. Die Anfrageklauseln sind durchweg negiert, und der Kopf der Wissensbasisklauseln ist immer positiv.

Wie bereits erwähnt, kann man die Gültigkeit einer Anfrage überprüfen, indem man die Anfrage negiert und der Wissensbasis hinzufügt. Wenn man dann die leere Klausel ableiten kann, ist der Beweis gelungen und die Anfrage erfüllt. In diesem Fall soll unser Prolog-System die Variablenbelegung ausgeben, und dabei hilft die SLD-Resolution. SLD steht für Linear Resolution with Selected Function for Definite Clauses. Anders als bei der normalen Resolution wird in jedem Schritt nur genau ein Schnittvorgang durchgeführt, also nur genau ein Literal und dessen Negation resolviert. Dadurch ist es möglich, im Falle des Backtracking die Resolution rückgängig zu machen. Wir wissen, daß Backtracking auch Variablenbelegungen aufhebt. Deswegen müssen wir in jedem Schritt die Variablenbelegung speichern. Und Variablenbelegung heißt in diesem Fall nichts anderes als Unifikation. Das Prolog-System merkt sich also für jeden Resolutionsschritt den Unifikator. Dieser entsteht dadurch, daß für ein Literal und dessen Negation der MGU ermittelt wird. Am Ende entsteht die Variablenbelegung einfach durch die Konkatenation aller Unifikatoren. Am anschaulichsten wird dies an einem Beispiel. Wir haben das folgende Prolog-Programm:

```
p(X) :-  
    q(X),  
    r(X).  
q(a).  
q(b).  
r(b).
```

Nun wollen wir verfolgen, wie man mit SLD-Resolution die Anfrage  $p(X)$  stellt:



Wie gehabt matcht die Anfrage  $p(X)$  mit dem Kopf der Klausel. Man beachte, daß die Variablen in Anfrage und Regel zwar gleich heißen können, aber trotzdem nichts miteinander zu tun haben. Deswegen sind die Variablen der Regel hier mit einem Strich (') benannt. Es wird eine Unifikation durchgeführt und danach ein weiteres Subgoal verfolgt, in diesem Fall  $q(X)$ . Nach Matching mit dem ersten Fakt erfolgt eine weitere Unifikation, so daß das neue Subgoal  $r(a)$  ist. Da das erste Subgoal nun erfüllt ist, wird nun das neue Subgoal verfolgt. Dies matcht aber mit keiner Klausel in unserer Wissensbasis, so daß wir diese Unifikation rückgängig machen müssen, um für  $q(X)$  eine andere Klausel zum Matching auszuprobieren. Diese führt zu der Unifikation von  $X$  mit  $b$ . Mit dieser Belegung matcht nun das letzte Subgoal mit dem entsprechenden Fakt in unserer Wissensbasis, und die Anfrage ist gelungen. Die Variablenbelegung besteht zwar formell aus  $\{X' \setminus X, X \setminus b\}$ , dies entspricht aber natürlich dem kürzeren  $\{X \setminus b\}$ .

Mit der SLD-Resolution haben wir also ein geeignetes Werkzeug für die Beweisführung in Prolog zur Verfügung. Allerdings handeln wir uns damit auch einen gewaltigen Nachteil ein. Die SLD-Resolution in Prolog ist nämlich nicht mehr vollständig, das heißt, wir können nicht alles beweisen, was prinzipiell

durch Standard-Resolution beweisbar ist. Beispiel:

```
p(X) :-  
        q(X).  
q(X) :-  
        p(X).  
q(a).
```

Stellen wir an dieses Programm die Anfrage  $p(X)$ , so geraten wir in eine Endlosschleife. Bei der normalen Resolution könnten wir uns aussuchen, welche Klausel wir matchen, aber die SLD-Resolution schreibt vor, die erste matchende Klausel zu benutzen.

## Kapitel 7

# Arithmetik und Operatoren

Aus Effizienzgründen handelt es sich bei der eingebauten Arithmetik der meisten Prologsysteme nicht um die logisch saubere Sukzessor-Arithmetik. Stattdessen werden die in die Prozessoren eingebauten Rechenkünste in Form des Prädikates `[is]` an das Prologsystem verkauft. Dieses Prädikat wertet arithmetische Ausdrücke aus und bindet das Ergebnis an eine Variable.

Alle Daten werden in Prolog in Form von Strukturen abgelegt, also in der Form `[funktor(arg1, arg2, ..., argn)]`. Es gibt Fälle, in denen uns diese Schreibweise unnatürlich vorkommt. `[3 + 5 * 6]` scheint leichter zu verstehen zu sein als `[(+,(3,*,(5,6)))]`. Es ist auch denkbar, statt `[liebt(hans, maria).]` schreiben zu wollen: `[hans liebt maria.]`. Sogenannte Operatordefinitionen machen es möglich, intuitivere Schreibweisen einzuführen und zu benutzen. Diese Definitionen erlauben es dem System, die neue Schreibweise auf die alte Schreibweise abzubilden und sie anschliessend korrekt zu verarbeiten.

## 7.1 Operatoren

Operatoren ermöglichen es, die Syntax von Prologprogrammen zu verändern, um z.B. die Lesbarkeit zu erhöhen.

Prolog erlaubt unäre und binäre Operatoren, d. h. Operatoren mit einem, bzw. zwei Argumenten.

### 7.1.1 Operatoren und Argumente, In-, Pre- und Postfixnotation

Mathematische Ausdrücke werden im Allgemeinen in sogenannter Infixnotation aufgeschrieben:

$5 + 3$

$6 * 7$

$8 * 4 - 2$

Die Zahlen sind die Argumente für die sogenannten Operatoren (+,-,\*,/,...). Die Operatoren legen fest, was mit ihren Argumenten geschehen soll, wenn der

Ausdruck ausgewertet wird. Prolog wertet arithmetischen Ausdrücke nicht automatisch aus! Zu diesem Zweck muss das Prädikat `[is]` bemüht werden (siehe 7.2.1). Infixnotation bedeutet, dass die Operatoren zwischen ihren Argumenten stehen. Ein Argument kann natürlich wiederum ein Ausdruck sein.

In Prolog können solche arithmetischen Ausdrücke in Strukturen (zusammengesetzte Terme, 2.2) abgelegt werden:

```
+ (5, 3)
*(6, 7)
-(*(8, 4), 2)
```

Die Operatoren sind hierbei die Funktoren, die Argumente die Komponenten der Strukturen. Bei dieser Schreibweise handelt es sich um die sogenannte Prefixnotation (erst der Operator, dann die Argumente).

Neben Pre- und Infixnotation existiert mit der Postfixnotation die Möglichkeit, erst die Argumente und dann den Operator anzugeben.

Die verschiedenen Notationen lassen sich ineinander überführen.

Operatoren kommen nicht nur in arithmetischen Ausdrücken vor. Beispielsweise sind auch `:-` und `,` (vordefinierte) Operatoren.

### 7.1.2 Umwandlung in Prefixnotation, Vorrang und Assoziativität von Operatoren

Da es möglich ist, die verschiedenen Notationen ineinander zu überführen, kann der Benutzer in seinen Programmen Infixschreibweise benutzen. Das Prologsystem kann daraus Prefixnotation generieren. Allerdings sind dazu bestimmte Informationen über die verwendeten Operatoren nötig.

Bei der Umwandlung eines Ausdrucks sind folgende Phänomene zu beachten:

- Der Ausdruck  $8 * 4 - 2$  ist zunächst einmal *ambig* (mehrdeutig). Zum einen könnte  $(8 * 4) - 2$  gemeint sein, zum anderen aber auch  $8 * (4 - 2)$ .
- Der Ausdruck  $8 - 4 - 2$  ist zunächst ebenfalls ambig. Zum einen könnte  $(8 - 4) - 2$  gemeint sein, zum anderen aber auch  $8 - (4 - 2)$ .

Um dafür sorgen zu können, dass zu jedem beliebigen Ausdruck nur ein eindeutiger Prefixausdruck gefunden werden kann, müssen den Operatoren zwei Eigenschaften gegeben werden. Zum einen ein sogenannter *Vorrang* (erstes Beispiel) und zum anderen Informationen über die sogenannte *Assoziativität* (zweites Beispiel).

Die arithmetischen Operatoren sind bereits vordefiniert, so dass man arithmetische Ausdrücke jederzeit infix schreiben kann.

#### Vorrang

Der Vorrang eines Operators wird in den meisten Prologsystemen als eine Zahl zwischen 1 und 1200 angegeben. Je kleiner diese Zahl für einen Operator ist, desto stärker bindet dieser, d.H. desto höher ist sein Vorrang. Der Operator `*` hat *zahlenmäßig* einen kleineren *Vorrangswert* als der Operator `+`. Das bedeutet, dass `*` Vorrang vor `+` hat.

Der Operator mit dem höchsten zahlenmässigen Vorrangswert — also der Operator mit der schwächsten Bindung — wird im Allgemeinen zum Hauptfunktoren der Prologstruktur.

### Assoziativitt, Position des Funktors zu den Argumenten

Falls zwei Operatoren denselben Vorrang besitzen, muss das Prologsystem entscheiden, ob es den Ausdruck von links nach rechts oder von rechts nach links auswertet. Dies ist die Aufgabe der Assoziativitt von Operatoren. Rechtsassoziativ bedeutet ‘von rechts nach links’. Linksassoziativ: ‘von links nach rechts’. Operatoren drfen auch nicht-assoziativ sein, d.h. zwei dieser Operatoren drfen nicht miteinander verkettet werden. Prolog erlaubt die Definition von Prefix-, Infix- und Postfixoperatoren mit unterschiedlicher Assoziativitt. Diese Eigenschaften eines Operators werden mit Zeichenketten dargestellt. Mit ‘x’ und ‘y’ werden Argumente bezeichnet, ‘f’ steht fr den Operator. Ein ‘x’ bedeutet, dass das Argument nur Operatoren einer echt kleineren Vorrangsklasse enthalten darf. (Argumente knnen beliebige Ausdrcke sein. Mehrere Operatoren knnen den gleichen zahlenmssigen Vorrang besitzen – siehe vordefinierte Operatoren 7.1.3.) Ein ‘y’ hingegen bedeutet, dass das Argument Operatoren mit gleichen oder kleineren zahlenmssigen Vorrangswerten enthalten darf.

In der Infixschreibweise steht der Operator zwischen seinen Argumenten. Folgende Mglichkeiten gibt es:

- $x f x$  nicht-assoziativ – z.B.: `:-`
- $x f y$  rechtsassoziativ – z.B.: `,`
- $y f x$  linksassoziativ – z.B.: `*`

In der Prefixschreibweise steht der Operator vor seinem Argument:

- $f x$  nicht-assoziativ – z.B.: `-`
- $f y$  linksassoziativ – z.B.: `[not]`

In der Postfixschreibweise steht der Operator hinter seinem Argument:

- $x f$  nicht-assoziativ
- $y f$  rechtsassoziativ

### Klammern

Mit Klammern lassen sich Vorrang und Assoziativitt wie gewohnt nach Wunsch umbiegen.

#### 7.1.3 vordefinierte Operatoren

Folgende Tabelle zeigt einige standardmssig vordefinierte Operatoren in Prolog. Eine komplette Liste lsst sich dem SWI-Manual entnehmen.

Vorrang	Assoziativität	Operator	Bedeutung
1200	xfx	<code>:</code> <code>-</code>	Implikationszeichen
1100	xfx	<code>;</code>	‘oder’
1000	xfy	<code>,</code>	‘und’
900	fy	<code>not</code> , <code>\+</code>	Negation als Scheitern
700	xfx	<code>=</code>	Unifikation
700	xfx	<code>is</code>	arithmetische Auswertung
700	xfx	<code>=..</code>	‘univ’
700	xfx	<code>\=</code>	<code>not Term1 = Term2</code>
700	xfx	<code>==</code>	beide Argumente haben gleiche Referenz
700	xfx	<code>\==</code>	<code>not Term1 == Term2</code>
700	xfx	<code>&lt;</code>	kleiner (Arithmetik)
700	xfx	<code>&gt;</code>	grösser (Arithmetik)
700	xfx	<code>=&lt;</code>	kleiner gleich (Arithmetik)
700	xfx	<code>&gt;=</code>	grösser gleich (Arithmetik)
500	fx	<code>+</code>	pos. Vorzeichen (Arithmetik)
500	fx	<code>-</code>	neg. Vorzeichen (Arithmetik)
500	yfx	<code>+</code>	Addition (Arithmetik)
500	yfx	<code>-</code>	Subtraktion (Arithmetik)
400	yfx	<code>*</code>	Multiplikation (Arithmetik)
400	yfx	<code>/</code>	Division (Arithmetik)
300	xfx	<code>mod</code>	Modulo (Arithmetik)

Ohne diese Operatordefinitionen müsste man z.B. `a :- b, c, d.` als `:(a, ,(b, ,(c, d)))` schreiben!

#### 7.1.4 eigene Operatordefinitionen

Eigene Operatoren werden mit dem Prädikat `op/3` definiert. `op(Vorrang, Assoziativität, Name).` macht `Name` zu einem Operator mit dem Vorrang `Vorrang` und der Assoziativität `Assoziativität`.

Das Prädikat `display/1` zeigt die interne Schreibweise eines Terms an, also die entstandene Präfixnotation. Gleiches leistet `write_canonical/1`. Beispiele:

```
%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%/%
% Assoziativität

?- op(100,xfy,cons). % cons ist rechtsassoziativ
Yes

?- _X = a cons b cons c cons nil, display(_X).
cons(a, cons(b, cons(c, nil)))
Yes

?- op(100,yfx,cons). % cons ist nun linksassoziativ
Yes

?- _X = a cons b cons c cons nil, display(_X).
```

```

cons(cons(cons(a, b), c), nil)
Yes

%%%%%%%%%%%%%
% Vorrang

?- op(150, xfy, ist_im), op(100, fx, essbarer).
Yes

?- _X = essbarer apfel ist_im korb, display(_X).
ist_im(essbarer(apfel), korb)
Yes

?- op(50, xfy, ist_im).                                % Vorrang geändert
Yes

?- _X = essbarer pilz ist_im schrank, display(_X).
essbarer(ist_im(pilz, schrank))                      % Au!
Yes

```

Operatordefinitionen in eigenen Programmtexten müssen als sogenannte *Direktiven* aufgeschrieben werden. Eine Direktive beginnt mit dem Implikationszeichen `[:-]`, es folgt eine Anfrage, die unmittelbar nach dem Konsultieren (Laden) der Wissensbasis ausgeführt wird:

```

% Operatordefinition richtig aufgeschrieben:
:- op(100,xfy, /\ ).
:- op(100,xfy, \ / ).

% Erzeugt Fehlermeldung, da hier kein Operator definiert wird, sondern
% versucht wird, das Prädikat op/3 neu zu definieren.
op(100,xfy, falsch ).
```

### 7.1.5 Ausführliche Beispiele zur Umwandlung von Ausdrücken in Prefixnotation

In diesem Abschnitt wollen wir verdeutlichen, wie das Prologsystem unter Verwendung der Operatordefinitionen aus Ausdrücken Präfixnotation erzeugt.

\*\*To do!\*\* -Algorithmus besorgen, erklären

### 7.1.6 Semantik von Operatoren

Operatorendefinitionen stellen neue Schreibweisen für Strukturen zur Verfügung. Operatoren erlauben eine neue Syntax, sorgen jedoch nicht dafür, dass die Ausdrücke auch eine Semantik(Bedeutung) erhalten. Daher werden Operatoren oft als "syntaktischer Zucker" bezeichnet.

```
% neue Syntax erlauben
:- op(100,xfx,ist_grossmutter_von).
```

```

:- op(100,xf,ist_eine_frau).
:- op(100,xf,ist_ein_mann).
:- op(100,xfx,ist_mutter_von).
:- op(100,xfx,ist_vater_von).

```

Die Prädikate, die dank der Operatoren nun ‘schöner’ aufgeschrieben werden dürfen, müssen natürlich auch definiert werden. Erst damit erhält die neue Syntax auch eine Semantik.

```

% Prädikate in neuer Syntax definieren
X ist_grossmutter_von Y :-
    X ist_mutter_von A,
    A ist_mutter_von Y.

X ist_grossmutter_von Y :-
    X ist_mutter_von A,
    A ist_vater_von Y.

maria    ist_eine_frau.
michalea ist_eine_frau.
susanne  ist_eine_frau.
anke     ist_eine_frau.
ist_eine_frau(erika).      % Die alte Schreibweise ist nach
                            % wie vor für alle Prädikate
                            % gültig!

jens    ist_ein_mann.
bernd   ist_ein_mann.

anke   ist_mutter_von maria.
maria ist_mutter_von jens.
erika ist_mutter_von bernd.
maria ist_mutter_von susanne.

bernd ist_vater_von jens.
bernd ist_vater_von susanne.

```

Es lassen sich nun Anfragen stellen wie z.B.:

```

?- Wer ist_vater_von susanne.
Wer = bernd
Yes

```

\*\*\* Weiteres Beispiel: X in Liste soll auf member abgebildet werden.

## 7.2 Arithmetik

Die Sukzessornotation (4.1.1) ist zwar logisch sauber, aber ineffizient im Vergleich zur ‘herkömmlichen’ auf Binärzahlen basierenden Arithmetik, die von Computerprozessoren geleistet wird. Die Geschwindigkeit von Arithmetik in der

Sukzessornotation ist abhängig von der Grösse der Zahlen, während Arithmetik mit Binärzahlen in konstanter Zeit abläuft. Prolog bietet Prädikate, die auf diese Funktionen zugreifen.

### 7.2.1 Arithmetische Ausdrücke, `[is]`

Wie wir in dem Abschnitt über Operatoren (7.1.1) bereits erfahren haben, werden arithmetische Ausdrücke nicht automatisch ausgewertet:

```
?- X = 123 + 45, display(X).
+(123, 45)
X = 123+45
```

Yes

Im obigen Beispiel ist der Term `[123+45]` an die Variable `[X]` gebunden worden. (`[X]` wurde mit der Struktur `[123+45]` instantiiert.) Das ist in etwa das gleiche als hätten wir

```
?- X = flutschi(a,b).
X = flutschi(a,b)
Yes
```

geschrieben. Oder mit entsprechender Operatordefinition:

```
?- X = a flutschi b.
X = a flutschi b
Yes
```

In Prolog braucht es ein spezielles Prädikat – `[is]` – um arithmetische Ausdrücke auszuwerten. `[is]` darf infix geschrieben werden und verlangt auf seiner linken Seite eine Variable oder eine Zahl und auf seiner rechten Seite einen arithmetischen Ausdruck. `[is]` interpretiert den Ausdruck und versucht, das Ergebnis mit seinem linken Argument zu unifizieren. `[is]` misst den Operatoren in dem Ausdruck also eine bestimmte Bedeutung bei. Was `[flutschi]` in `[X is a flutschi b.]` bedeuten soll, kann `[is]` natürlich nicht ahnen. Dafür kennt es aber `[+, -, *, /]` und noch einige mehr. `[is]` akzeptiert auf der rechten Seite außerdem nur Zahlen und Variablen, die mit Zahlen instantiiert sind. Uninstantiierte Variablen und Atome wie `[a]`, `[b]` oder `[maria]` kann `[is]` nicht interpretieren.

```
?- X is 123 + 45.
X = 168
Yes
```

```
?- X is 1 * 2 * 3 * 4 * 5 * 6.
X = 720
Yes
```

```
?- 42 is 7 * 6.
Yes
```

Prolog enthält keinen Gleichungslöser! Auf der linken Seite von **[is]** dürfen keine arithmetischen Ausdrücke auftreten.

```
?- 7 * 6 is 6 * 7.
No
```

Auf der linken Seite von **[is]** dürfen auftreten: Zahlen, uninstantiierte Variablen und mit Zahlen instantiierte Variablen. **[is]** versucht, das Ergebnis der Auswertung der rechten Seite mit der linken Seite zu unifizieren. Ist auf der linken Seite eine uninstantiierte Variable zu finden, gelingt die Unifikation natürlich sofort. Tritt eine Zahl oder eine mit einer Zahl instantiierte Variable auf, so gelingt die Unifikation nur, wenn die Zahlen übereinstimmen.

### 7.2.2 eigene Arithmetikprädikate

Mit Hilfe des Prädikats **[arithmetic\_function/1]** können eigene Prädikate als Arithmetikprädikate benutzbar gemacht werden. Das jeweils letzte Argument eines Arithmetikprädikates wird zur Ergebnisübergabe genutzt. Eigene Arithmetikprädikate benötigen also ein zusätzliches Argument.

```
% minimum(A,B, Ergebnis)

minimum(A,B, A) :- A < B.
minimum(A,B, B) :- A >= B.
```

```
?- arithmetic_function(minimum/3).
```

Yes

```
?- X is minimum(27,15).
X = 15
Yes
```

### 7.2.3 Länge einer Liste

Unter Verwendung der Arithmetik können wir ein Prädikat schreiben, das die Länge einer Liste ermittelt:

```
laenge1([],0).

laenge1([_|R], L) :-
    laenge1(R, RL),
    L is 1+ RL.

% Restrekursive Version
laenge2([],Erg,Erg).

laenge2([_|R],Bisher, Erg) :-
    Bisher2 is Bisher + 1,
    laenge2(R, Bisher2, Erg).
```

```
% Beispielenfragen
?- laenge1([a,b,c],Laenge).
Laenge = 3
Yes

?- laenge2([a,b,c],0,Laenge).
Laenge = 3
Yes
```

#### 7.2.4 Zugriff auf n-tes Element

Das folgende (restrekursive) Prädikat erlaubt den Zugriff auf das n-te Element einer Liste:

```
enthalten_an_position(Element, Liste, Position) :-  
    e(Element, Liste, 0, Position).

e(E, [E|_], Bisher, Pos) :-  
    Pos is Bisher + 1.

e(E, [_|R], Bisher, Pos) :-  
    Bisher2 is Bisher + 1,  
    e(E,R,Bisher2, Pos).

% Beispielenfragen
?- enthalten_an_position(r,[l,a,g,e,r,r,e,g,a,l],6).
Yes

?- enthalten_an_position(e,[l,a,g,e,r,r,e,g,a,l],Pos).
Pos = 4 ;
Pos = 7 ;
No

?- enthalten_an_position(x,Liste,6).
Liste = [_G376, _G382, _G388, _G394, _G400, x|_G407]
Yes
```



# Kapitel 8

## Graphen und Suche

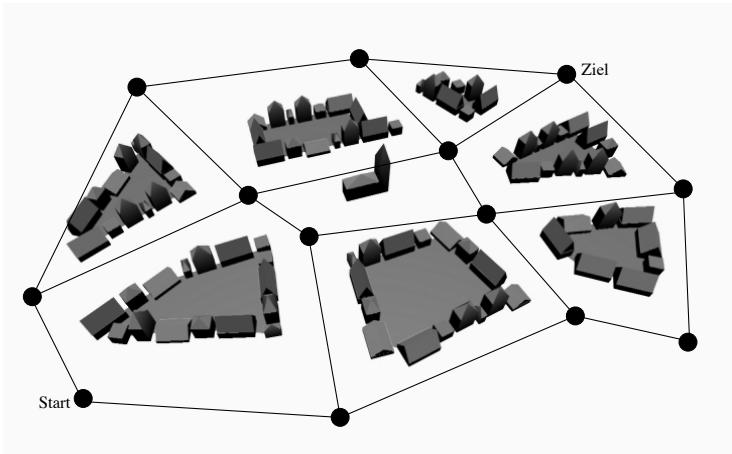
Suche ist ein wesentlicher Bestandteil der Künstlichen Intelligenz und kann zur Lösung einer Fülle (scheinbar) verschiedener Probleme herangezogen werden.

In diesem Kapitel werden Graphen und Suche behandelt. Zunächst werden die allgemeinen Ideen relativ umfassend dargestellt. Anschliessend werden Graphen und Suche in Prolog Schritt für Schritt von ganz einfachen Programmen zu komplexeren entwickelt.

### 8.1 Einleitung

Die folgende Grafik soll einen Stadtplan symbolisieren. An Strassenkreuzungen sind schwarze Kreise eingetragen, die über Linien verbunden sind. Diese Linien liegen auf den Straßen. Ab jetzt werden wir die Kreise als *Knoten* bezeichnen und die Linien *Kanten* nennen.

Stellen wir uns folgendes vor: Ein Fußgänger steht auf dem Knoten, der mit ‘Start’ beschriftet ist und möchte zu dem Knoten namens ‘Ziel’ gehen. Er wandert auf seinem Weg von Start zu Ziel von einem Knoten zum nächsten. Dabei geht er entlang der Kanten, schließlich kann er weder fliegen noch durch Mauern hindurchgehen.



Einige Knoten haben mehrere Kanten, es gibt also mehrere Möglichkeiten weiterzugehen. Unser Fußgänger muss sich jeweils für einen Nachfolgeknoten entscheiden, zu dem er sich bewegt. Es gibt eine ganze Reihe verschiedener Wege, die vom Start zum Ziel führen, einen davon wird der Fußgänger gegangen sein, wenn er am Ziel ankommt.

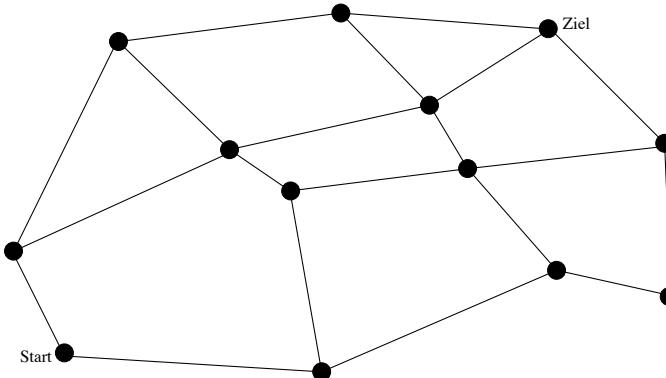
Die möglichen Wege unterscheiden sich in der Anzahl der Knoten, die besucht wurden und in der metrmässig zurückgelegten Strecke, bzw. der benötigten Zeit. Wenn man bedenkt, dass der Fußgänger beliebig oft im Kreis oder zwischen zwei Knoten hin- und herlaufen kann, wird schnell klar, dass es in dem obigen Beispiel unendlich viele Wege vom Start zum Ziel geben muss.

Die Anzahl der Knoten, die ein bestimmter Weg enthält, wird die *Länge* des Weges genannt. Benötigte Zeit, metrmässige Strecke oder ähnliche Eigenschaften werden als *Kosten* des Weges bezeichnet. Läuft der Fußgänger im Kreis oder zwischen zwei Knoten hin- und her, spricht der Fachmann von einem *Zyklus*.

Die Aktion, die der Fußgänger ausführen muss, um von einem Knoten zu einem Nachfolgeknoten zu gelangen ist ‘gehen’. Es sind Szenarien denkbar, in denen verschiedene Verkehrsmittel zur Verfügung stehen. Z.B. könnten einige Knoten nur mit einer Straßenbahn, andere nur per Schiff erreichbar sein. Man könnte sich vorstellen, dass ein kurzes aber steil nach oben steigendes Wegstück anstrengender zu bewältigen ist, als ein langes flaches. Auch Einbahnstrassen sind denkbar. Der Fantasie sind kaum Grenzen gesetzt.

## 8.2 Graphen

Wenn wir den Stadtplan aus dem obigen Beispiel wegwerfen und nur die Knoten und Kanten sowie die anderen Ideen behalten, so erhalten wir einen Graphen:

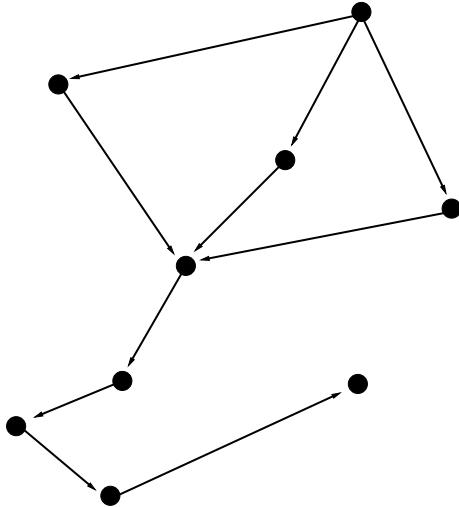


In diesem Graphen kann man jede Kante in beide Richtungen abwandern, der Graph wird daher *ungerichtet* genannt. Ein Graph kann auch *gerichtet* sein. In einem gerichteten Graphen fügt man an die Kanten kleine Pfeilspitzen an, die die erlaubte Gehrichtung anzeigen. Graphen können Zyklen enthalten, man spricht in diesem Zusammenhang von *zyklischen* und *azyklischen* Graphen.

In einem Graphen kann ein Weg von einem Startknoten zu einem Zielknoten gesucht werden.

### 8.2.1 DAGs

Gerichtete Graphen ohne Zyklen haben den Spitznamen DAG (directed acyclic graph) erhalten. Neben dem folgenden Beispiel sind auch Bäume DAGs.



### 8.2.2 Anwendungen von Graphen und Suche

Graphen lassen sich nicht nur nutzen, um in Stadtplänen einen Weg von A nach B zu finden. Auch scheinbar ganz andere Probleme lassen sich mit Graphen repräsentieren und mit Suche lösen.

Solche ‘Probleme’ sind z.B.: Theorembeweisen (Prolog), Parsen von Sätzen oder Spielzüge finden, wobei es für jedes spezialisierte Algorithmen gibt.

#### Beispiel: eine Umfüllaufgabe

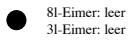
Umfüllaufgaben lauten etwa folgendermassen:

Sie haben zwei Eimer. Der eine Eimer fasst 8 Liter, der andere 3 Liter. Im Traum erscheint Ihnen Elvis und befiehlt Ihnen, nur unter Zuhilfenahme der beiden Eimer (und unbegrenzt viel Wasser) 4 Liter Wasser in den großen Eimer zu füllen. Sie wollen dem King eine Freude machen und die Aufgabe lösen.

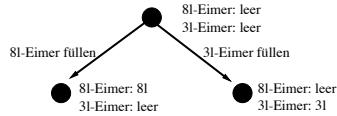
Am Anfang sind beide Eimer leer. Diesen *Zustand* nennen wir den *Startzustand*. Vom Startzustand ausgehend wollen wir einen Zustand erreichen, in dem der grösitere Eimer 4 Liter Wasser enthält. Der Inhalt des kleineren Eimers ist egal. Es gibt also möglicherweise mehr als nur einen einzigen *Zielzustand*.

Es gibt einige Operationen, die mit den Eimern durchgeführt werden können. Die Eimer können (an einem Wasserhahn) mit Wasser aufgefüllt werden. Wasser kann aus einem Eimer in den anderen Eimer umgefüllt werden, bis dieser voll, bzw. der Ausgangseimer leer ist. Zu guter letzt kann jeder der Eimer ausgeschüttet werden.

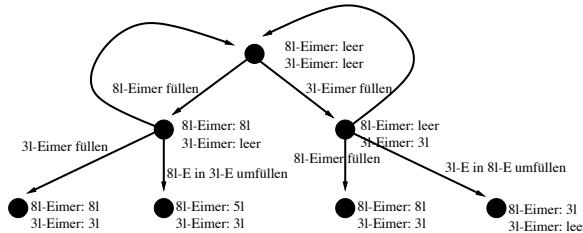
Der Startzustand kann als Knoten in einem Graphen repräsentiert werden:



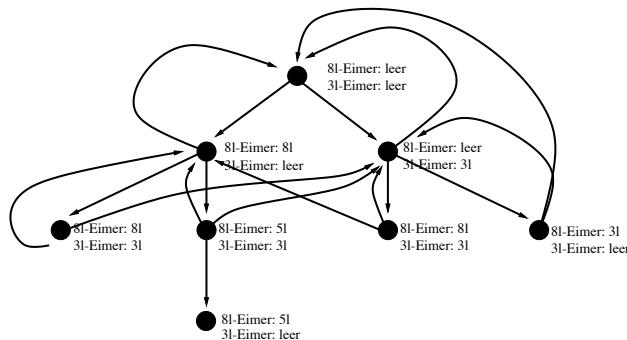
Die Nachfolgeknoten stellen die Folgezustände dar, die durch das Anwenden je einer Aktion (Füllen vs. Umfüllen vs. Leeren) ergeben:



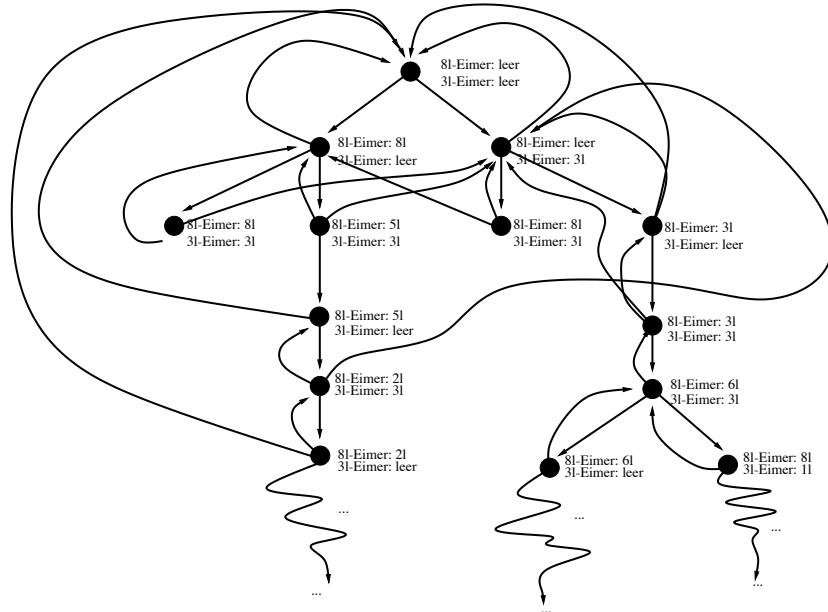
Diese Knoten haben entsprechend den ausführbaren Operationen wiederum Nachfolger:



Noch mehr Nachfolger:



Und noch mehr:

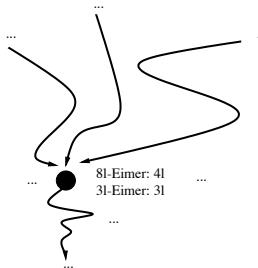


Die Analogie aus der Einleitung kann auch auf diesen Graphen angewendet werden: Gehen wird in Auffüllen, Umfüllen und Auskippen untergliedert, Knoten stellen statt Straßenkreuzungen Eimerzustände dar. Mit diesen Ersetzungen läuft unser Fussgänger in einer seltsamen Stadt herum, bis er eine Straßenkreuzung erreicht, auf der ein mit 4l Wasser gefüllter 8l-Eimer und ein 3l-Eimer mit beliebigem Inhalt herumstehen.

Der Graph ist von Abbildung zu Abbildung gewachsen, in dem die zuletzt dazugekommenen Knoten *expandiert* worden sind. Einen Knoten *expandieren* bedeutet, die Nachfolgeknoten des Knotens zur ermitteln. Die Nachfolgeknoten zu einem Knoten können sich bereits in der Wissensbasis befinden (z.B. Stadtplan), oder dynamisch bei Bedarf erzeugt werden. Letzteres bietet sich für das Umfüllproblem an. Man baut den Graphen nicht vorher von Hand auf, sondern überlässt diese Arbeit dem Prologsystem. Dazu müssen Regeln formuliert werden, die beschreiben, wie die Nachfolgeknoten aus einem Knoten zu gewinnen sind.

Der auf diese Weise für unsere Umfüllaufgabe entstandene Graph enthält Zyklen, die beliebig lange durchwandert werden können, ohne dass ein Zielzustand erreicht wird. Suchverfahren in Prolog müssen sich daher um die Vermeidung von Zyklen kümmern.

Irgendwo in dem Graphen befindet sich der folgende Knoten:

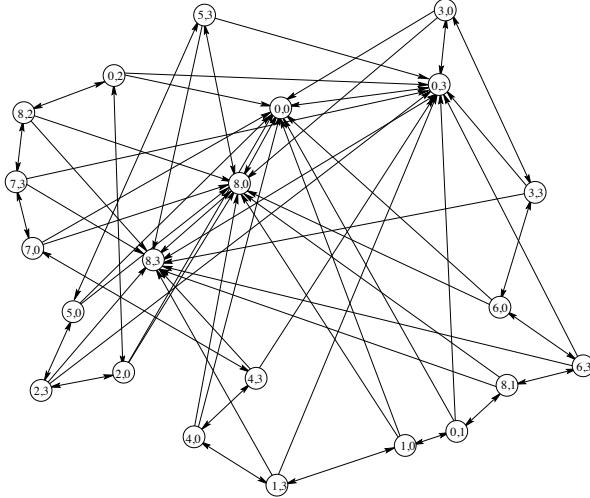


Ein Weg dorthin lässt sich als Abfolge von Zuständen und Aktionen beschreiben. Bei dem folgenden Weg handelt es sich weder um den besten noch den einzigen Weg.

- Beide Eimer leer. Fülle 8l-Eimer.
- (8l-E:8l, 3l-E:0l) Fülle 8l-Eimer in 3l-Eimer um.
- (8l-E:5l, 3l-E:3l) Leere 3l-Eimer.
- (8l-E:5l, 3l-E:0l) Fülle 8l-Eimer in 3l-Eimer um.
- (8l-E:2l, 3l-E:3l) Leere 3l-Eimer.
- (8l-E:2l, 3l-E:0l) Fülle 8l-Eimer in 3l-Eimer um.
- (8l-E:0l, 3l-E:2l) Fülle 8l-Eimer.
- (8l-E:8l, 3l-E:2l) Fülle 8l-Eimer in 3l-Eimer um.
- (8l-E:7l, 3l-E:3l) Leere 3l-Eimer.
- (8l-E:7l, 3l-E:0l) Fülle 8l-Eimer in 3l-Eimer um.

- (8l-E:4l, 3l-E:3l) Zielzustand erreicht.

Dieses Problem lässt sich also ebenso wie das Problem mit dem Stadtplan durch einen Graphen repräsentieren und mit Suche im Graphen lösen. Der komplette Graph dieses Umfüllproblems kann z.B. folgendermaßen dargestellt werden:



Andere Probleme können wesentlich komplexere Graphen erzeugen. Es kann passieren, dass der Graph unendlich gross wird. Der Graph stellt den sogenannten *Suchraum* dar.

### 8.2.3 Anmerkungen

Wichtig in einem Graphen sind die Verbindungen von einem Knoten zu seinen Nachfolgern. Ob man die Linien später krumm oder gerade zeichnet, ist egal. Die Koordinaten der Punkte in den obigen Beispielen sind willkürlich gewählt worden. In der Regel gibt man jedem Knoten einen eindeutigen Namen und legt alle relevanten Informationen unter diesem Namen ab. Je nach Problemstellung können dies beliebige Zustandsinformationen, z.B. Koordinaten oder Füllstände, sein. Die Verbindungen zu den Nachfolgern können um beliebige Kosten angereichert werden. Ein Synonym für Weg ist Pfad.

Im Beispiel der Umfüllaufgabe bestehen die Namen der Knoten nur aus den Füllständen der Eimer. Dadurch sind die Knoten eindeutig identifizierbar.

## 8.3 Graphen und Suche in Prolog

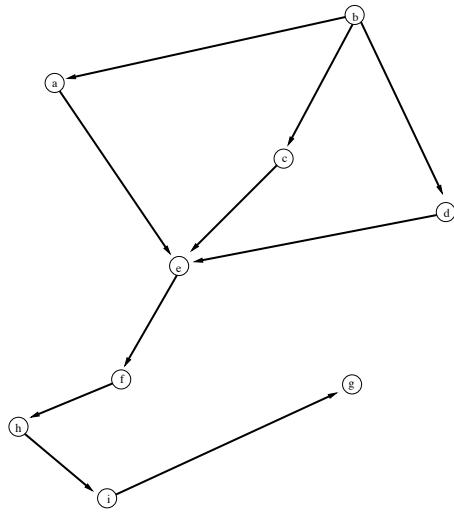
In den folgenden Abschnitten soll geklärt werden, wie die Suche in einem Graphen nun eigentlich von statthaften gehen kann. Es gibt eine ganze Reihe verschiedener Suchverfahren, die unterschiedliche Eigenschaften besitzen. Einige dieser Verfahren werden wir in diesem Kapitel kennen lernen.

Zunächst wird mit einfachsten Mitteln vorgegangen. Anschließend wird eine Tiefensuche entwickelt, die die Hauptarbeit dem Prologbeweiser überlässt. Diese Tiefensuche wird in einem nächsten Abschnitt um ein eigenes Gedächtnis – eine so genannte *Agenda* – erweitert und ist damit unabhängiger vom Beweiser. Mit einem einfachen Handgriff wird diese Tiefensuche schließlich in eine

Breitensuche umgewandelt. Von dort aus ist es nur ein kleiner Schritt zu einer Best-First Suche. Es folgt ein einfaches heuristisches Verfahren namens ‘Nearest Neighbour’. Abschließend wird ein kleiner Ausblick auf weitere Verfahren gegeben.

### 8.3.1 mit einfachsten Mitteln

Der folgende DAG soll für unsere ersten Gehversuche herhalten.



Die einfachste Art, den Graphen zu repräsentieren ist als Fakten und Regeln:

```

a :- e.      % von a geht es nach e
b :- a.      % von b geht es nach a
b :- c.      % von b geht es nach c
b :- d.      % von b geht es nach d
c :- e.      % von c geht es nach e
d :- e.      % von d geht es nach e
e :- f.      % von e geht es nach f
f :- h.      % von f geht es nach h
h :- i.      % von h geht es nach i
i :- g.      % von i geht es nach g
  
```

Die Regel `a :- e.` beispielsweise kann in diesem Kontext als “von `a` geht ein direkter Weg nach `e`” gelesen werden, statt als “um `a` zu beweisen, muss `e` bewiesen werden”. Gibt es zu einem Knoten mehrere Nachfolger, so werden diese dank des Backtrackings zu entsprechender Zeit berücksichtigt (alternative Regelrümpfe). Die Zeile `i :- g.` enthält im Rumpf das Prädikat `g`, das nicht definiert ist. Dies führt zu einer Fehlermeldung: `ERROR: Undefined procedure: g/0 ....` Dieser Fehler kann umgangen werden, in dem `g :- fail.` als Definition von `g` eingetragen wird. Auf diese Weise lassen sich alle Knoten eintragen, die keine Nachfolger haben.

Diese Art, Graphen zu repräsentieren eignet sich nur für DAGs. Wäre die Kante zwischen `a` und `b` ungerichtet, lautete die Repräsentation `a :- b. b :- a.` Der Prologbeweiser würde in eine Endlosschleife gelangen.

Um mit diesem Prologprogramm eine Suche anzustossen, müssen Start- und Zielknoten festgelegt werden. Den Startknoten wählt man über die Anfrage. Der Zielknoten muss in der Wissensbasis markiert werden. Es muss dafür gesorgt werden, dass der Beweis des Zielknotens gelingt, damit der gesamte Suchvorgang ein Ende findet. Die Wissensbasis muss also je nach Zielknoten entsprechend verändert werden.

Das Ziel der Suche soll im ersten Beispiel der Knoten **[g]** sein. Es muss also dafür gesorgt werden, dass der Beweis von **[g]** gelingt. Die Zeile **[g :- fail.]** wird ersetzt durch **[g.]**.

Nun muss nur noch eine entsprechende Anfrage gestellt werden, um die Suche in Gang zu bringen. Der Startknoten soll **[b]** sein. Die Anfrage lautet also:

**?- b.**

**Yes**

Das System hat bewiesen, dass es einen Weg von **[b]** nach **[g]** gibt.

Das nächste Ziel soll **[e]** sein. Die Zeile **[e :- f.]** wird dazu durch **[e.]** ersetzt, während statt **[g.]** erneut **[g :- fail.]** eingetragen werden muss. Einige Anfragen:

**?- a.** % Gibt es einen Weg von a nach e?  
**Yes**

**?- d.** % Gibt es einen Weg von d nach e?  
**Yes**

**?- g.** % Gibt es einen Weg von g nach e?  
**No**

**?- h.** % Gibt es einen Weg von h nach e?  
**No**

**?- i.** % Gibt es einen Weg von i nach e?  
**No**

Die Ergebnisse können mit Blick auf den Graphen bestätigt werden. Das System kann dazu gebracht werden, den Weg von Start zu Ziel als Ergebnis abzuliefern. Dazu muss die Repräsentation der Kanten ein wenig geändert werden:

```
a([a|X]) :- e(X).
b([b|X]) :- a(X).
b([b|X]) :- c(X).
b([b|X]) :- d(X).
c([c|X]) :- e(X).
d([d|X]) :- e(X).
e([e|X]) :- f(X).
f([f|X]) :- h(X).
g([g|X]) :- fail.
h([h|X]) :- i(X).
```

```
%i([i|X]) :- g(X).
i([i]).
```

```
% Suche Weg von b nach i:
?- b(Weg).
```

```
Weg = [b, a, e, f, h, i] ;
```

```
Weg = [b, c, e, f, h, i] ;
```

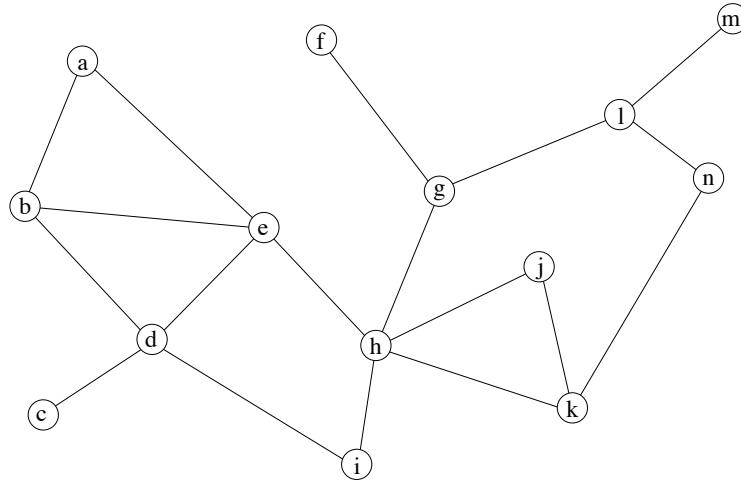
```
Weg = [b, d, e, f, h, i] ;
```

No

### 8.3.2 verbessertes Vorgehen

Das Programm aus dem vorherigen Abschnitt ist auf DAGs beschränkt. Um diese Beschränkung aufheben zu können, muss der zu durchsuchende Graph in einer anderen Form repräsentiert werden.

Die Relation `kante/2` besteht zwischen zwei Knoten. Mit ihr wird aufgeschrieben, dass eine Kante vom ersten Argument zum zweiten Argument führt.



```
kante(a,b).    kante(a,e).    kante(b,e).
kante(b,d).    kante(c,d).    kante(d,e).
kante(d,i).    kante(e,h).    kante(f,g).
kante(g,h).    kante(g,l).    kante(h,i).
kante(h,j).    kante(h,k).    kante(j,k).
kante(k,n).    kante(l,n).    kante(l,m).
```

Auf die angegebene Weise ist der Graph noch nicht komplett abgebildet. Die oben aufgeführten Kanten sind gerichtet, während es sich eigentlich um einen ungerichteten Graphen handelt. Statt die Umkehrungen — also z.B. `kante(b,a).` — extra aufzuführen, können wir ein Prädikat definieren, das für Symmetrie sorgt.

Wenn der Graph sowohl ungerichtete als auch gerichtete Kanten enthält, muss das natürlich berücksichtigt werden.

### Wege und direkte Wege, Prolog

Wir schreiben auf, was wir unter einem Weg verstehen wollen. Es gibt einen Weg von **Start** nach **Ziel**, wenn

- **Start** gleich **Ziel** ist, oder
- es einen direkten Weg von **Start** zu einem **Zwischenknoten** gibt und es einen Weg von diesem **Zwischenknoten** zum **Ziel** gibt.

Einen direkten Weg von **A** nach **B**, gibt es in diesem Kontext, wenn

- es eine Kante von **A** nach **B** oder
- es eine Kante von **B** nach **A** gibt.

Diese Definitionen lassen sich leicht in Prolog umsetzen:

**kante/2** Die Kanten sind als Fakten repräsentiert (siehe oben).

**direkter\_weg/2** Kanten sind direkte Wege. In diesem Beispiel ist der Graph ungerichtet. **direkter\_weg/2** macht **kante/2** symmetrisch.

```
direkter_weg(A,B) :- kante(A,B).
direkter_weg(A,B) :- kante(B,A).      % Symmetrie
```

**weg/2** Es gibt einen Weg zwischen Start und Ziel, wenn Start = Ziel ist.

```
weg(Start, Start).
```

Es gibt einen Weg von Start nach Ziel, wenn es einen direkten Weg von Start zu einem Zwischenknoten gibt, sowie einen Weg vom Zwischenknoten zum Ziel.

```
weg(Start, Ziel):- direkter_weg(Start, Zwischenknoten),
                  weg(Zwischenknoten, Ziel).
```

Es ist wichtig, dass die Prädikate **weg** und **direkter\_weg** unterschiedliche Namen, bzw. Stelligkeiten besitzen. Hießen beide **weg**, würde die 2. Klausel des eigentlichen **weg** folgendermassen aussehen:

```
% falsch!
% weg(Start, Ziel):-   weg(Start, Zwischenknoten),  % endlose Rekursion!
%                      weg(Zwischenknoten, Ziel).
```

Damit würde der Beweiser sofort in eine Endlosschleife abtauchen. Es handelt sich um einen beliebten Fehler.

Die ‘richtige’ Version von **weg/2** bringt den Beweiser jedoch auch in Endlosschleifen, da sie nichts unternimmt, um Zyklen zu vermeiden:

```
?- weg(a,a).
Yes
```

```
?- weg(a,b).
Yes
```

```
?- weg(a,c).
ERROR: Out of local stack
```

Um Zyklen zu vermeiden, muss sichergestellt werden, dass nicht zu einem Knoten gegangen wird, der schon einmal besucht worden ist. Kommt man da an, wo man losgegangen ist, ist man im Kreis gegangen. Aufgrund seiner Art Klauseln auszuwählen, verlässt der Beweiser Zyklen nicht. Um dies vermeiden zu können, muss sich das Prädikat `weg` merken, welche Knoten es schon besucht hat, also welchen Weg es bisher gegangen ist. Dazu wird ein weiteres Argument benötigt. `weg/3`:

```
weg(Start, Start, _BisherigerWeg). % Ziel erreicht!
```

```
weg(Start, Ziel, BisherigerWeg) :-  
    direkter_weg(Start, Zwischenknoten), % zu ZK gehen  
    not(member(Zwischenknoten, BisherigerWeg)), % Zykluscheck!  
    weg(Zwischenknoten, Ziel, [Start|BisherigerWeg]). % von ZK zu Ziel  
                                         % weitersuchen
```

Beim Aufruf wird im dritten Argument eine leere Liste übergeben, da ja noch kein Knoten besucht wurde.

```
?- weg(a,a,[]).
Yes
```

```
?- weg(a,b,[]).
Yes
```

```
?- weg(a,c,[]).
Yes
```

Die erste Klausel von `weg/3` greift, wenn das Ziel erreicht worden ist. Die zweite Klausel wählt einen direkten Weg von `Start` zu einem `Zwischenknoten` und prüft, ob `Zwischenknoten` nicht ein Element der Liste `BisherigerWeg` ist. Scheitert das Subgoal `not(member(Zwischenknoten, BisherigerWeg))`, wird Backtracking initiiert. Das führt dazu, dass ein neuer Nachfolger des Knotens `Start` gewählt wird, indem eine Alternative für `direkter_weg(Start, Zwischenknoten)` gewählt wird. Wenn es keine Alternative gibt, die funktioniert, erfolgt Backtracking zurück zum Vorgängerknoten. Auf diese Weise können ganze Teilstücke des Weges rückgängig gemacht werden. All dies leistet der Prologbeweiser! Wurde nun ein `Zwischenknoten` gewählt, der noch nicht besucht worden ist, wird `weg/3` rekursiv aufgerufen: `weg(Zwischenknoten, Ziel, [Start|BisherigerWeg]).`

Für diese neue Instanz von `weg/3` wird `Zwischenknotenalt` zu `Startneu`

, `Zielalt` zu `Zielneu` und `BisherigerWegalt` um `Startalt` erweitert zu `BisherigerWegneu`.

Irgendwann wird entweder der Zielknoten erreicht, oder alle erreichbaren Knoten sind erfolglos besucht worden. Im letzteren Fall gibt es keinen Weg. Ein Suchverfahren, das erst scheitert, nachdem alle erreichbaren Knoten besucht wurden, wird als *erschöpfend* bezeichnet.

Nachdem wir nun die Zyklen aus dem Weg geräumt haben, wüssten wir natürlich auch gerne, wie der zurückgelegte Weg eigentlich aussieht.

Der Liste `BisherigerWeg` wird der aktuelle Knoten vorangestellt, um eine neue Liste zu erhalten, die den bisherigen Weg enthält. Auf diese Weise kann ein ‘teurer’ Aufruf von `append/3` gespart werden. Diese Liste enthält den zurückgelegten Weg also in umgekehrter Reihenfolge. Wir müssen nur dafür sorgen, dass der Zielknoten auch mitanhängt wird und eine Kopie der Liste aus der untersten Rekursionsebene nach oben durchgereicht wird. Dazu brauchen wir ein weiteres Argument. `weg/4` lautet:

```
% weg(Start, Ziel, BisherigerWeg, ErgebnisWeg).

weg(Ziel, Ziel, BisherigerWeg, [Ziel|BisherigerWeg]).

weg(Start, Ziel, BisherigerWeg, ErgebnisWeg) :-
    direkter_weg(Start, Zwischenknoten),
    not(member(Zwischenknoten, BisherigerWeg)),
    weg(Zwischenknoten, Ziel, [Start|BisherigerWeg], ErgebnisWeg).
```

Beispielaufruf:

```
?- weg(a,c,[],X). % Suche Weg X von a nach c
X = [c, d, i, h, e, b, a] ;
X = [c, d, e, b, a] ;
X = [c, d, b, a] ;
X = [c, d, i, h, e, a] ;
X = [c, d, b, e, a] ;
X = [c, d, e, a] ;
```

No

Es ist auffällig, dass der erste gefundene Weg kein besonders guter Weg ist. Das hier vorgestellte Programm kann nicht garantieren, dass es auf Anhieb einen guten Weg findet, aber es garantiert, dass es alle Wege findet, falls es welche gibt. Es handelt sich um ein erschöpfendes Verfahren, das als *Tiefensuche* bekannt ist. Das das Programm eine Tiefensuche ergibt, liegt daran, dass der Prologbeweiser den Großteil der Arbeit erledigt: Auswahl des jeweiligen Nachfolgeknotens (= Klauselauswahl) und Backtracking. Die Wissensbasis wird von

oben nach unten durchsucht. Backtracking findet immer am zuletzt angelegten Choicempoint statt. `weg/4` hat genau einen Weg im ‘Gedächtnis’, nämlich den, den es aktuell verfolgt. Außerdem leistet `weg/4` einen Zykluscheck, der den Beweiser davon abhält, in Endlosschleifen zu laufen. Alles weitere wird vom Beweiser erbracht, der ja bekanntlich ebenfalls eine Tiefensuche einsetzt (siehe Abschnitt 5.3.1 auf Seite 71). Die Tiefensuche des Beweisers wurde in Abschnitt 5.3.2 auf Seite 71 grafisch veranschaulicht. Tiefensuche wird in Abschnitt 8.3.8 grafisch veranschaulicht und einem weiteren Suchverfahren – der Breitensuche – gegenübergestellt.

### Listing dieses Abschnitts

Es folgt das komplette Listing dieses Abschnitts einschließlich einem Prädikat `tief1/3`, das `weg/4` entsprechend aufruft und den Ergebnisweg richtigerum liefert.

```
% Kanten
```

```
kante(a,b).    kante(a,e).    kante(b,e).
kante(b,d).    kante(c,d).    kante(d,e).
kante(d,i).    kante(e,h).    kante(f,g).
kante(g,h).    kante(g,l).    kante(h,i).
kante(h,j).    kante(h,k).    kante(j,k).
kante(k,n).    kante(l,n).    kante(l,m).
```

```
% direkte Wege
```

```
direkter_weg(A,B) :- kante(A,B).
direkter_weg(A,B) :- kante(B,A).      % Symmetrie
```

```
% Weg/2
```

```
weg(Start, Start).
weg(Start, Ziel) :- direkter_weg(Start, Zwischenknoten),
                  weg(Zwischenknoten, Ziel).
```

```
% Weg/3
```

```
weg(Start, Start, BisherigerWeg).

weg(Start, Ziel, BisherigerWeg) :-
    direkter_weg(Start, Zwischenknoten),
    not(member(Zwischenknoten, BisherigerWeg)),      % Zykluscheck!
    weg(Zwischenknoten, Ziel, [Start|BisherigerWeg]).
```

```
% Weg/4
```

```
% weg(Start, Ziel, BisherigerWeg, ErgebnisWeg).

weg(Ziel, Ziel, BisherigerWeg, [Ziel|BisherigerWeg]). 

weg(Start, Ziel, BisherigerWeg, ErgebnisWeg) :- 
    direkter_weg(Start, Zwischenknoten),
    not(member(Zwischenknoten, BisherigerWeg)),
    weg(Zwischenknoten, Ziel, [Start|BisherigerWeg], ErgebnisWeg).

% Bequemer Aufruf von weg/4, gelieferter Weg ist in der richtigen
% Reihenfolge

tief1(Start, Ziel, WEG) :- 
    weg(Start, Ziel, [], GEW),
    reverse(GEW, WEG).
```

### 8.3.3 Tiefensuche

Die Tiefensuche (Depth-First Search) wird so genannt, weil sie *einen* Weg immer weiter expandiert. Der Abstand des letzten Knoten des Weges zum Startknoten wird dabei immer größer, die Suche geht in die Tiefe. Durch Backtracking schließlich werden wenn nötig Stücke des Weges zurückgenommen und alternative Abzweigungen können verfolgt werden. Ein Baum würde mit Tiefensuche astweise (von oben nach unten) durchsucht werden. In Abschnitt 8.3.8 werden Tiefensuche (und Breitensuche) grafisch dargestellt.

Die Programme in den vorherigen Abschnitten ergeben Tiefensuche, da der Prologbeweiser selbst eine Tiefensuche fährt und die Programme diese Eigenschaft erben.

In diesem Abschnitt werden wir eine Tiefensuche implementieren, die unabhängiger vom Prologbeweiser ist, indem sie alternative Wege in einer so-nannten *Agenda* speichert. Dieses ‘Gedächtnis’ macht die Suche unabhängiger, da sie ohne Backtracking an Alternativen kommen kann und die Auswahl von Nachfolgewegen unabhängiger von der Klauselauswahl des Beweisers treffen kann.

Die Agenda dient als Speicher für Wege, die früher oder später einmal verfolgt werden müssen. Die Agenda ist eine Liste von Wegen. Der Tiefensuchalgorithmus entnimmt der Agenda den ersten Weg und prüft, ob dieser im Zielknoten endet. Ist das der Fall, ist die Suche erfolgreich abgeschlossen. Führt der Weg nicht zum Ziel, so werden alle Nachfolgewege bestimmt, also alle Möglichkeiten, vom aktuellen Weg noch einen Knoten weiter zu gehen. Dabei werden alle Alternativen, die zu Zyklen führen eliminiert. Die verbleibenden Wege werden an den Anfang der Agenda gestellt. Der Algorithmus wird erneut durchlaufen, d.H. es wird wieder der erste weg aus der Agenda entfernt, expandiert und überprüft. Da es vorkommt, dass Wege nicht weiter expandiert werden können, leert sich die Agenda mit der Zeit wieder. Ist die Agenda komplett leer, wurde der Graph komplett durchsucht, ohne das ein Weg vom Start- zum Zielknoten gefunden werden konnte. Initialisiert wird die Agenda mit dem Startknoten der Suche als Weg der Länge 1. Die Agenda verwaltet also alternative (Teil-) Wege. Diese Arbeit wurde in den bisherigen Programmen vom Beweiser geleistet.

### 8.3.4 Tiefensuchalgorithmus

Der Tiefensuchalgorithmus kann wie folgt charakterisiert werden. Die Agenda wird zu Beginn der Suche mit dem Startknoten als Weg der Länge 1 initialisiert.

- Ist die Agenda leer, so ist die Suche erschöpft. Alle möglichen Wege wurden erkundet.
- Ist die Agenda nicht leer, entnehme der Agenda den ersten Weg.
  - Führt dieser Weg zum Ziel:
    - \* liefere ihn als Ergebnis, terminiere
  - Führt dieser Weg nicht zum Ziel, bzw. weitere Lösung gewünscht:
    - \* erzeuge alle seine Nachfolgewege, die keine Zyklen enthalten,
    - \* füge diese Nachfolgewege vorne in die Agenda ein,
    - \* suche mit der neuen Agenda weiter

### 8.3.5 Tiefensuchalgorithmus in Prolog

In diesem Abschnitt wollen wir zunächst die spezielleren Prädikate und anschließend das darauf aufbauende Suchprädikat erstellen, also einmal ‘bottom-up’ vorgehen.

Das Tiefensuchprädikat stützt sich auf folgende Dinge:

**Kanten** Kanten zwischen zwei Knoten werden wir wie bisher notieren.

```
kante(a,b).    kante(a,e).    kante(b,e).
kante(b,d).    kante(c,d).    kante(d,e).
kante(d,i).    kante(e,h).    kante(f,g).
kante(g,h).    kante(g,l).    kante(h,i).
kante(h,j).    kante(h,k).    kante(j,k).
kante(k,n).    kante(l,n).    kante(l,m).
```

**direkte Wege** Da wir noch immer von einem ungerichteten Graphen ausgehen, sorgen wir für die Symmetrie der Kanten:

```
direkter_weg(A,B) :- kante(A,B).
direkter_weg(A,B) :- kante(B,A).      % Symmetrie
```

**Knoten** Wenn jeder Knoten mindestens einen Nachfolgeknoten besitzt, also in zumindest einer Kantenrelation auftritt, ist die Menge der Knoten implizit durch die Kanten gegeben. Daher müssen die Knoten nicht extra notiert werden.

**Weg** Ein Weg soll wie bisher eine Liste von Knoten sein. Aus Effizienzgründen werden Wege rückwärts gespeichert, d.h. der erste Knoten in der Liste ist der letzte Knoten des Weges. Ein Weg von **[d]** nach **[g]** kann also beispielsweise so aussehen:

**[g, h, e, b, d]**

**Agenda** Die Agenda ist eine Liste von Wegen. In dieser Liste herrscht die ‘normale’ Reihenfolge.

Diese Beispielagenda enthält einige Wege, die am Knoten `[d]` beginnen und Alternativen zueinander darstellen, da sie unterschiedlich abbiegen.

```
[[a, b, d], [e, b, d], [i, d], [e, d]]
```

**Nachfolgewege erzeugen** Um die Nachfolgewege eines Weges zu erzeugen, nehme man den letzten Knoten des Weges – also das erste Element der Liste, die den Weg repräsentiert – und besorge nun alle Nachfolgeknoten dieses Knotens. Für jeden gefundenen Knoten, der nicht in dem zu expandierenden Weg vorkommt, erzeuge man eine Kopie des zu expandierenden Weges und hänge den gefundenen Nachfolgeknoten an.

Beispiel: Der erste Weg aus der obigen Agenda lautet `[a, b, d]`. Der letzte Knoten dieses Weges ist der erste in der Liste, also `[a]`. Die Nachfolger von `[a]` sind `[b]` und `[e]`. Es ergeben sich zwei Nachfolgewege, `[b, a, b, d]` und `[e, a, b, d]`, wobei ersterer ausscheidet, weil er einen Zyklus enthält.

Ein weiteres Beispiel: Der Weg `[h]` soll expandiert werden. Die Nachfolgeknoten von `[h]` lauten: `[g]`, `[e]`, `[k]`, `[j]` und `[i]`. (Reihenfolge ergibt sich durch Klauselauswahl und Reihenfolge der `kante`n.) Die Nachfolgewege lauten also: `[g, h]`, `[e, h]`, `[k, h]`, `[j, h]` und `[i, h]`.

Um alle Nachfolger eines Knotens zu besorgen, benutzen wir das Built-in Prädikat `findall(Var, Goal, Ergebnisliste)`, das alle Instanziierungen der Variable `Var`, die in dem Ziel `Goal` möglich sind, in der Liste `Ergebnisliste` aufsammelt.

```
?- findall(X,kante(h,X),Liste).
```

```
X = _G301
Liste = [i, j, k]
```

Yes

Die Funktionsweise von `findall/3` wird in Abschnitt 11.1.2 erklärt.

Schliesslich wird noch ein Prädikat benötigt, das aus einem Weg und einer Liste von Nachfolgeknoten des letzten Knotens des Weges eine Liste von

Nachfolgewegen erzeugt. `e(Weg, Nachfolgeknoten, Akkumulator, Ergebnis)` soll so eine Liste erzeugen. `e/4` verarbeitet die Liste der Nachfolgeknoten rekursiv. Es gibt drei Fälle:

- Die Liste der Nachfolgeknoten ist leer. Unifiziere Ergebnis mit dem Akkumulator.

```
e(_, [], X, X).
```

- Der erste der Nachfolgeknoten  $E$  ist nicht Bestandteil von  $\text{Weg}$ . Erweitere  $\text{Weg}$  um  $E$  und stelle diesen neuen Weg den bisher schon erstellten Wegen in der Akkumulatorliste  $\text{Bisher}$  voran und bilde so einen neuen Akkumulator. Verarbeite die restlichen Nachfolgeknoten  $R$  mit dem neuen Akkumulator  $[[E|\text{Weg}]|\text{Bisher}]$  rekursiv weiter, um  $\text{Ergebnis}$  zu erhalten.

```
e(Weg, [E|R], Bisher, Ergebnis) :-  
    not(memberchk(E,Weg)),  
    e(Weg, R, [[E|Weg]|Bisher], Ergebnis).
```

- Der erste Nachfolgeknoten  $E$  ist Bestandteil von  $\text{Weg}$ . Überspringe  $E$  und verarbeite die übrigen Nachfolger  $R$  rekursiv weiter. Auf diese Weise werden Zyklen vermieden.

```
e(Weg, [E|R], Bisher, Ergebnis) :-  
    memberchk(E,Weg),  
    e(Weg, R, Bisher, Ergebnis).
```

Aus diesen Teilen kann nun ein Prädikat **expandiere/2** zusammengesetzt werden, das eine Liste von Nachfolgewegen zu einem gegebenen Weg liefert.

```
expandiere([E|R], Wege) :-  
    findall(X,direkter_weg(E,X), Nachfolger),  
    e([E|R], Nachfolger, [], Wege).
```

Beispiele:

```
?- expandiere([i],X).  
X = [[h, i], [d, i]]  
Yes  
  
?- expandiere([d,i],X).  
X = [[c, d, i], [b, d, i], [e, d, i]]  
Yes  
  
?- expandiere([e,d,i],X).  
X = [[b, e, d, i], [a, e, d, i], [h, e, d, i]]  
Yes  
  
?- expandiere([h,e,d,i],X).  
X = [[g, h, e, d, i], [k, h, e, d, i], [j, h, e, d, i]]  
Yes
```

Nun haben wir alle Bausteine beieinander, die zur Formulierung des Tiefensuchtprädictates mit Agenda notwendig sind. Zwei Klauseln reichen aus:

```
% ts(Agenda, Ziel, Ergebnisweg)
ts([[Ziel|R]|_RestWege], Ziel, [Ziel|R]). 

ts([Weg1 | RestWege], Ziel, Weg) :- 
    expandiere(Weg1, NachfolgeWege),
    append(NachfolgeWege, RestWege, NeueAgenda),
    ts(NeueAgenda, Ziel, Weg).
```

Die erste Klausel versucht, den letzten Knoten des ersten Weges in der Agenda mit dem Zielknoten zu unifizieren, wenn das gelingt, ist das Ziel erreicht und der erste Weg der Agenda wird als Ergebnis geliefert. Die Agenda ist eine Liste von Wegen:  $\text{Agenda} = [\text{Weg}_1, \text{Weg}_2, \dots, \text{Weg}_n]$ . Wege werden durch Listen repräsentiert:  $\text{Weg}_i = [\text{Knoten}_n, \text{Knoten}_{n-1}, \dots, \text{Knoten}_1]$ . (Die Liste, die einen Weg repräsentiert enthält dessen Knoten in umgekehrter Reihenfolge.) Die Agenda ist also eine Liste von Listen:  $\text{Agenda} = [[\text{Knoten}_{w1n}, \dots, \text{Knoten}_{w11}], \dots, [\text{Knoten}_{wmk}, \dots, \text{Knoten}_{wm1}]]$ .

An den ersten Weg der Agenda kommt man mit folgender Unifikation:  $[\text{E} | \text{Restwege}] = \text{Agenda}$ , an den letzten Knoten des ersten Weges — also den ersten Knoten der Liste die an  $\text{E}$  gebunden ist — kommt man mit dieser Unifikation:  $[\text{K1} | \text{Restknoten}]$ .

Beide Schritte lassen sich zusammenfassen zu  $[[\text{K1} | \text{Restknoten}] | \text{Restwege}] = \text{Agenda}$ .

Wenn  $\text{K1}$  sich mit dem Zielknoten unifizieren lässt, ist  $[\text{K1} | \text{Restknoten}]$  ein Ergebnisweg. Das ist alles, was in der ersten Klausel passiert.

Die zweite Klausel besorgt sich per Unifikation den ersten Weg  $\text{Weg1}$  aus der Agenda und erzeugt alle Nachfolgewege. An die Nachfolgewege werden die restlichen Wege der Agenda mit  $\text{append/3}$  angehängt, um so eine neue Agenda ( $\text{neueAgenda}$ ) zu erzeugen. Man sagt, dass die Nachfolgewege vorne an die Agenda angehängt werden.  $\text{neueAgenda}$  enthält nicht  $\text{Weg1}$ ! Anschließend wird mit der neuen Agenda rekursiv weitergesucht.

Ist die Agenda leer, matcht der rekursive Aufruf der Suche mit keiner der Klauseln des Suchprädikates, damit scheitert der Beweis.

Die Agenda muss mit dem Startknoten als Weg initialisiert werden. Ist der Startknoten beispielsweise  $\text{k}$ , so lautet die initiale Agenda  $[[\text{k}]]$ .

Das folgende Prädikat ermöglicht einen bequemen Aufruf der Tiefensuche, in dem es die Agenda automatisch initialisiert:

```
tiefensuche(Start, Ziel, Weg) :- 
    ts([[Start]], Ziel, Gew),
    reverse(Gew, Weg).
```

Beispielaufruf:

```
tiefensuche(d,g,Weg).
```

```
Weg = [d, b, a, e, h, g] ;
```

```
Weg = [d, b, a, e, h, k, n, l, g] ;
```

```
Weg = [d, b, a, e, h, j, k, n, l, g] ;
```

```

Weg = [d, b, e, h, g] ;
Weg = [d, b, e, h, k, n, l, g]
Yes

```

**tiefensuche(a,c,Weg) . protokolliert**

Tiefensuche von a nach c. Initialisiere Agenda: [[a]]

- Suchschritt 1
  - Die Agenda lautet: [ [a] ]
  - Nehme ersten Weg aus der Agenda: [a].
  - Die Nachfolgewege dieses Weges lauten: [ [e, a], [b, a] ]
  - Füge die Nachfolgewege vorne an die Agenda an.
  - Die neue Agenda lautet: [ [e, a], [b, a] ]
  - Suche mit neuer Agenda weiter.
- Suchschritt 2
  - Die Agenda lautet: [ [e, a], [b, a] ]
  - Nehme ersten Weg aus der Agenda: [e, a].
  - Die Nachfolgewege dieses Weges lauten: [ [d, e, a], [b, e, a], [h, e, a] ]
  - Füge die Nachfolgewege vorne an die Agenda an.
  - Die neue Agenda lautet: [ [d, e, a], [b, e, a], [h, e, a], [b, a] ]
  - Suche mit neuer Agenda weiter.
- Suchschritt 3
  - Die Agenda lautet: [ [d, e, a], [b, e, a], [h, e, a], [b, a] ]
  - Nehme ersten Weg aus der Agenda: [d, e, a].
  - Die Nachfolgewege dieses Weges lauten: [ [c, d, e, a], [b, d, e, a], [i, d, e, a] ]
  - Füge die Nachfolgewege vorne an die Agenda an.
  - Die neue Agenda lautet: [ [c, d, e, a], [b, d, e, a], [i, d, e, a], [b, e, a], [h, e, a], [b, a] ]
  - Suche mit neuer Agenda weiter.
- Suchschritt 4
  - Der Weg [c, d, e, a] endet im Zielknoten c.
- Suchschritt 5
  - Die Agenda lautet: [ [c, d, e, a], [b, d, e, a], [i, d, e, a], [b, e, a], [h, e, a], [b, a] ]

- Nehme ersten Weg aus der Agenda: [c, d, e, a].
  - Die Nachfolgewege dieses Weges lauten: [ ]
  - Füge die Nachfolgewege vorne an die Agenda an.
  - Die neue Agenda lautet: [ [b, d, e, a], [i, d, e, a], [b, e, a], [h, e, a], [b, a] ]
  - Suche mit neuer Agenda weiter.
- ...

### das komplette Programm

```
% Kanten
kante(a,b).    kante(a,e).    kante(b,e).
kante(b,d).    kante(c,d).    kante(d,e).
kante(d,i).    kante(e,h).    kante(f,g).
kante(g,h).    kante(g,l).    kante(h,i).
kante(h,j).    kante(h,k).    kante(j,k).
kante(k,n).    kante(l,n).    kante(l,m).

% direkte Wege
direkter_weg(A,B) :- kante(A,B).
direkter_weg(A,B) :- kante(B,A).      % Symmetrie

% Wege in Nachfolgewege expandieren
expandiere([E|R], Wege) :-
    findall(X,direkter_weg(E,X), Nachfolger),
    e([E|R], Nachfolger, [], Wege).

% Hilfsprädikat für expandiere
e(_, [], X, X).

e(Weg, [E|R], Bisher, Ergebnis) :-
    not(memberchk(E,Weg)),
    e(Weg, R, [[E|Weg]|Bisher], Ergebnis).

e(Weg, [E|R], Bisher, Ergebnis) :-
    memberchk(E,Weg),
    e(Weg, R, Bisher, Ergebnis).

% Tiefensuche
% ts(Agenda, Ziel, Ergebnisweg)

ts([[Ziel|R]|_RestWege], Ziel, [Ziel|R]). 

ts([Weg1 | RestWege], Ziel, Weg) :-
    expandiere(Weg1, NachfolgeWege),
```

```
append(NachfolgeWege, RestWege, NeueAgenda),
ts(NeueAgenda, Ziel, Weg).
```

```
% bequemer Aufruf
tiefensuche(Start, Ziel, Weg) :-
    ts([[Start]], Ziel, Gew),
    reverse(Gew, Weg).
```

### 8.3.6 Breitensuche

Die Breitensuche (Breadth-First Search) untersucht zuerst alle vom Startknoten ausgehende Wege der Länge 0, dann alle Wege der Länge 1, ..., und schließlich alle Wege der Länge n. Ein Baum würde also ebenenweise (in der Breite) durchsucht werden. Daher der Name und darüberhinaus die Eigenschaft, den — gemessen an der Anzahl der Knoten — kürzesten Weg als ersten zu finden. In Abschnitt 8.3.8 werden Breiten- und Tiefensuche einander grafisch gegenübergestellt.

Die Tiefensuche aus Abschnitt 8.3.3 lässt sich auf denkbar einfache Weise in eine Breitensuche umwandeln. Die Reihenfolge der Wege in der Agenda muss so geändert werden, dass die Wege nach Länge sortiert sind. Die kürzesten Wege stehen dabei vorne. Das ist insofern sehr einfach, weil diese Sortierung bereits dadurch erreicht wird, dass expandierte Wege hinten in die Agenda aufgenommen werden. Neue Wege müssen also am Ende der Agenda eingefügt werden, statt wie bei der Tiefensuche am Anfang der Agenda.

Das Programm zur Breitensuche lautet analog zu dem der Tiefensuche, mit dem Unterschied, dass die ersten beiden Argumente des `append`-Aufrufes vertauscht werden müssen:

```
breitensuche(Start, Ziel, Weg) :-
    bs([[Start]], Ziel, Gew),
    reverse(Gew, Weg).

bs([[Ziel|R] | _RestWege], Ziel, [Ziel|R]).
```

```
bs([Weg1 | RestWege], Ziel, Weg) :-
    expandiere(Weg1, NachfolgeWege),
    append(RestWege, NachfolgeWege, NeueAgenda),
    bs(NeueAgenda, Ziel, Weg).
```

Beispielaufruf:

```
?- breitensuche(d,g,Weg).
```

```
Weg = [d, i, h, g] ;
```

```
Weg = [d, e, h, g] ;
```

```
Weg = [d, b, e, h, g] ;
```

```
Weg = [d, b, a, e, h, g] ;
```

Weg = [d, i, h, k, n, l, g] ;

Weg = [d, e, h, k, n, l, g]

Yes

**breitensuche(a,c,Weg). protokolliert**

Breitensuche von a nach c. Initialisiere Agenda: [[a]]

- Suchschritt 1

- Die Agenda lautet: [ [a] ]
- Nehme ersten Weg aus der Agenda: [a].
- Die Nachfolgewege dieses Weges lauten: [ [e, a], [b, a] ]
- Füge die Nachfolgewege hinten an die Agenda an.
- Die neue Agenda lautet: [ [e, a], [b, a] ]
- Suche mit neuer Agenda weiter.

- Suchschritt 2

- Die Agenda lautet: [ [e, a], [b, a] ]
- Nehme ersten Weg aus der Agenda: [e, a].
- Die Nachfolgewege dieses Weges lauten: [ [d, e, a], [b, e, a], [h, e, a] ]
- Füge die Nachfolgewege hinten an die Agenda an.
- Die neue Agenda lautet: [ [b, a], [d, e, a], [b, e, a], [h, e, a] ]
- Suche mit neuer Agenda weiter.

- Suchschritt 3

- Die Agenda lautet: [ [b, a], [d, e, a], [b, e, a], [h, e, a] ]
- Nehme ersten Weg aus der Agenda: [b, a].
- Die Nachfolgewege dieses Weges lauten: [ [d, b, a], [e, b, a] ]
- Füge die Nachfolgewege hinten an die Agenda an.
- Die neue Agenda lautet: [ [d, e, a], [b, e, a], [h, e, a], [d, b, a], [e, b, a] ]
- Suche mit neuer Agenda weiter.

- Suchschritt 4

- Die Agenda lautet: [ [d, e, a], [b, e, a], [h, e, a], [d, b, a], [e, b, a] ]
- Nehme ersten Weg aus der Agenda: [d, e, a].
- Die Nachfolgewege dieses Weges lauten: [ [c, d, e, a], [b, d, e, a], [i, d, e, a] ]
- Füge die Nachfolgewege hinten an die Agenda an.

- Die neue Agenda lautet: [ [b, e, a], [h, e, a], [d, b, a], [e, b, a], [c, d, e, a], [b, d, e, a], [i, d, e, a] ]
  - Suche mit neuer Agenda weiter.
- Suchschritt 5
  - Die Agenda lautet: [ [b, e, a], [h, e, a], [d, b, a], [e, b, a], [c, d, e, a], [b, d, e, a], [i, d, e, a] ]
    - Nehme ersten Weg aus der Agenda: [b, e, a].
    - Die Nachfolgewege dieses Weges lauten: [ [d, b, e, a] ]
    - Füge die Nachfolgewege hinten an die Agenda an.
  - Die neue Agenda lautet: [ [h, e, a], [d, b, a], [e, b, a], [c, d, e, a], [b, d, e, a], [i, d, e, a], [d, b, e, a] ]
    - Suche mit neuer Agenda weiter.
- Suchschritt 6
  - Die Agenda lautet: [ [h, e, a], [d, b, a], [e, b, a], [c, d, e, a], [b, d, e, a], [i, d, e, a], [d, b, e, a] ]
    - Nehme ersten Weg aus der Agenda: [h, e, a].
    - Die Nachfolgewege dieses Weges lauten: [ [g, h, e, a], [k, h, e, a], [j, h, e, a], [i, h, e, a] ]
      - Füge die Nachfolgewege hinten an die Agenda an.
    - Die neue Agenda lautet: [ [d, b, a], [e, b, a], [c, d, e, a], [b, d, e, a], [i, d, e, a], [d, b, e, a], [g, h, e, a], [k, h, e, a], [j, h, e, a], [i, h, e, a] ]
      - Suche mit neuer Agenda weiter.
- Suchschritt 7
  - Die Agenda lautet: [ [d, b, a], [e, b, a], [c, d, e, a], [b, d, e, a], [i, d, e, a], [d, b, e, a], [g, h, e, a], [k, h, e, a], [j, h, e, a], [i, h, e, a] ]
    - Nehme ersten Weg aus der Agenda: [d, b, a].
    - Die Nachfolgewege dieses Weges lauten: [ [c, d, b, a], [i, d, b, a], [e, d, b, a] ]
      - Füge die Nachfolgewege hinten an die Agenda an.
    - Die neue Agenda lautet: [ [e, b, a], [c, d, e, a], [b, d, e, a], [i, d, e, a], [d, b, e, a], [g, h, e, a], [k, h, e, a], [j, h, e, a], [i, h, e, a], [c, d, b, a], [i, d, b, a], [e, d, b, a] ]
      - Suche mit neuer Agenda weiter.
- Suchschritt 8
  - Die Agenda lautet: [ [e, b, a], [c, d, e, a], [b, d, e, a], [i, d, e, a], [d, b, e, a], [g, h, e, a], [k, h, e, a], [j, h, e, a], [i, h, e, a], [c, d, b, a], [i, d, b, a], [e, d, b, a] ]
    - Nehme ersten Weg aus der Agenda: [e, b, a].
    - Die Nachfolgewege dieses Weges lauten: [ [d, e, b, a], [h, e, b, a] ]
      - Suche mit neuer Agenda weiter.

- Füge die Nachfolgewege hinten an die Agenda an.
- Die neue Agenda lautet: [ [c, d, e, a], [b, d, e, a], [i, d, e, a], [d, b, e, a], [g, h, e, a], [k, h, e, a], [j, h, e, a], [i, h, e, a], [c, d, b, a], [i, d, b, a], [e, d, b, a], [d, e, b, a], [h, e, b, a] ]
- Suche mit neuer Agenda weiter.
- Suchschritt 9
  - Der Weg [c, d, e, a] endet im Zielknoten c.
- Suchschritt 10
  - Die Agenda lautet: [ [c, d, e, a], [b, d, e, a], [i, d, e, a], [d, b, e, a], [g, h, e, a], [k, h, e, a], [j, h, e, a], [i, h, e, a], [c, d, b, a], [i, d, b, a], [e, d, b, a], [d, e, b, a], [h, e, b, a] ]
  - Nehme ersten Weg aus der Agenda: [c, d, e, a].
  - Die Nachfolgewege dieses Weges lauten: [ ]
  - Füge die Nachfolgewege hinten an die Agenda an.
  - Die neue Agenda lautet: [ [b, d, e, a], [i, d, e, a], [d, b, e, a], [g, h, e, a], [k, h, e, a], [j, h, e, a], [i, h, e, a], [c, d, b, a], [i, d, b, a], [e, d, b, a], [d, e, b, a], [h, e, b, a] ]
  - Suche mit neuer Agenda weiter.
- ...
  - ...

### 8.3.7 Anmerkungen

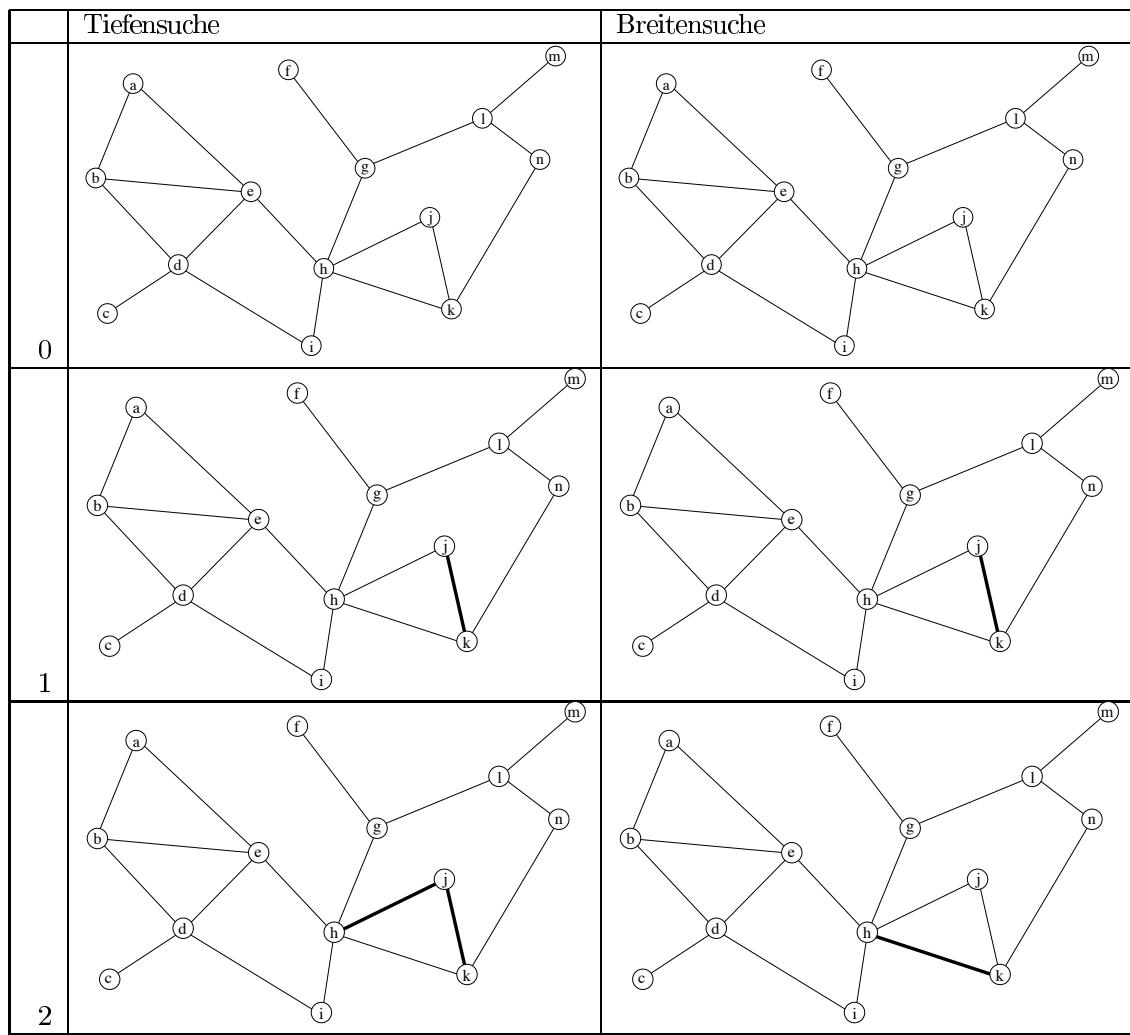
Bei aufmerksamer Betrachtung fällt auf, dass die hier vorgestellten Versionen der Suchalgorithmen eine Schwäche haben. Nachdem die erste Klausel der Tiefen- oder auch der Breitensuche erfolgreich gewählt und damit ein Weg gefunden wurde, bleibt der erfolgreiche Weg in der Agenda. D.h. fordert man eine weitere Lösung an, wird der erfolgreiche Weg von eben aus der Agenda genommen und expandiert. Diese neuen Wege haben natürlich keine Chance, da sie den Zielknoten ja bereits enthalten und dank des Zyklenchecks kein zweites Mal erreichen können. Gerade die Tiefensuche verfolgt in einem solchen Fall eine ganze Menge aussichtsloser Wege. Dieses Problem könnte gelöst werden durch einen weiteren Check im **expandiere**-Prädikat, das als zusätzliches Argument den Zielknoten übergeben bekommen müsste und keine Expansionen zulassen dürfte, die den Zielknoten bereits enthalten. Im Abschnitt 8.3.8 werden solche Wege übergangen.

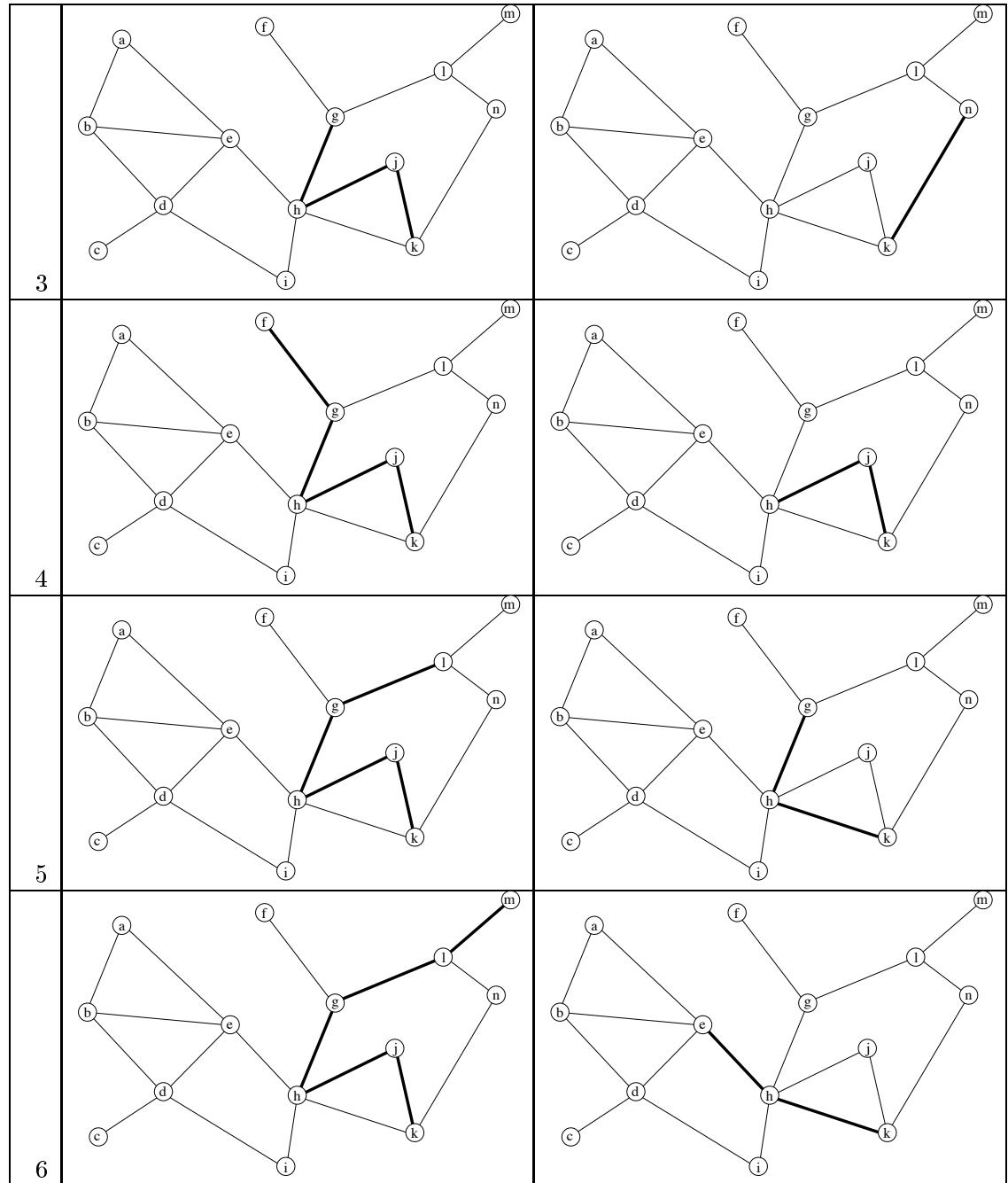
Allerdings kann es auch durchaus der Fall sein, dass es eine ganze Reihe verschiedener Zielknoten gibt. Es könnte passieren, dass ein besonders guter Weg durch einen ‘mittelmässigen’ Zielknoten läuft und in einem sehr guten Zielknoten endet. Mehrere Zielknoten können ermöglicht werden, indem die erste Klausel der Suche entsprechend angepasst wird.

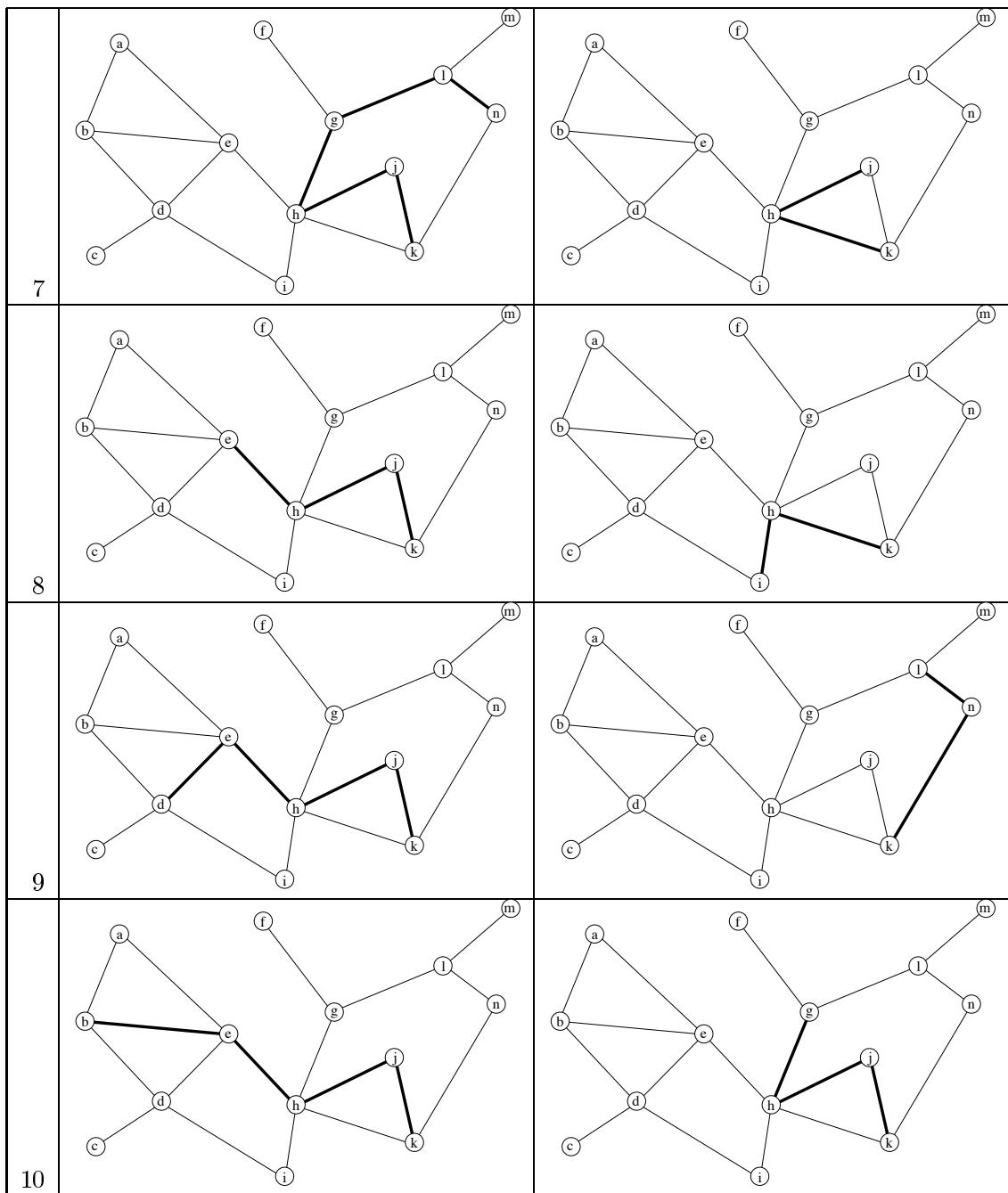
### 8.3.8 Veranschaulichung von Tiefen- und Breitensuche

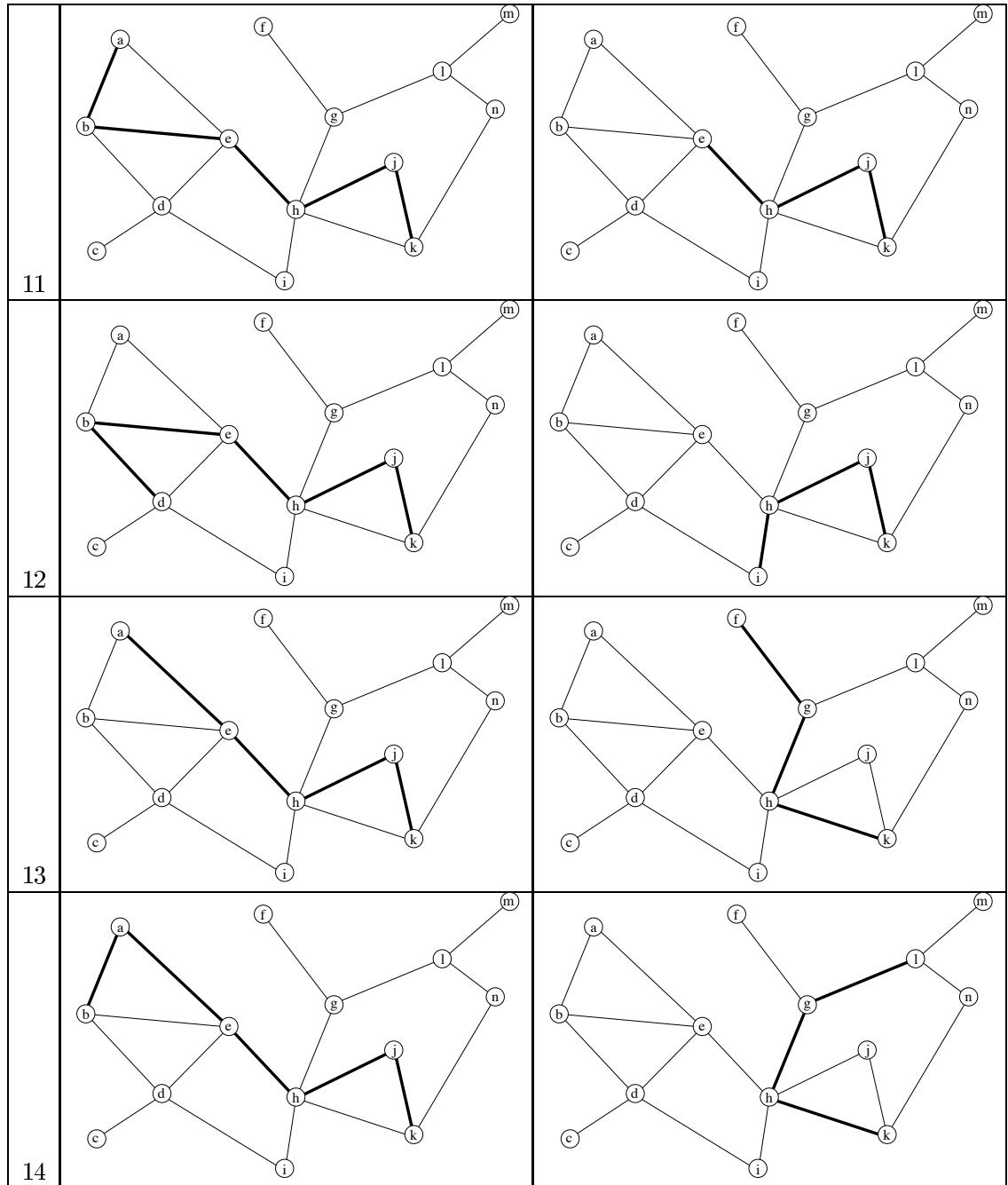
In dem Graphen aus Abschnitt 8.3.2 soll von Knoten k nach Knoten d gesucht werden. Die Tabelle stellt den Verlauf einer Breitensuche dem einer Tiefensuche gegenüber. Die Breitensuche betrachtet zunächst alle Wege der Länge 1, dann alle Wege der Länge 2, usw., während die Tiefensuche den aktuellen Weg immer weiter expandiert und gegebenenfalls Backtracking betreibt. In welcher Reihenfolge die Nachfolger eines Knotens ausgewählt werden, hängt von der Klauselreihenfolge ab. Da die Tiefensuche einen Weg solange expandiert, bis es nicht mehr weiter geht, wird sie von der Klauselreihenfolge stärker beeinflusst. Die Klauselreihenfolge für dieses Beispiel ist willkürlich gewählt.

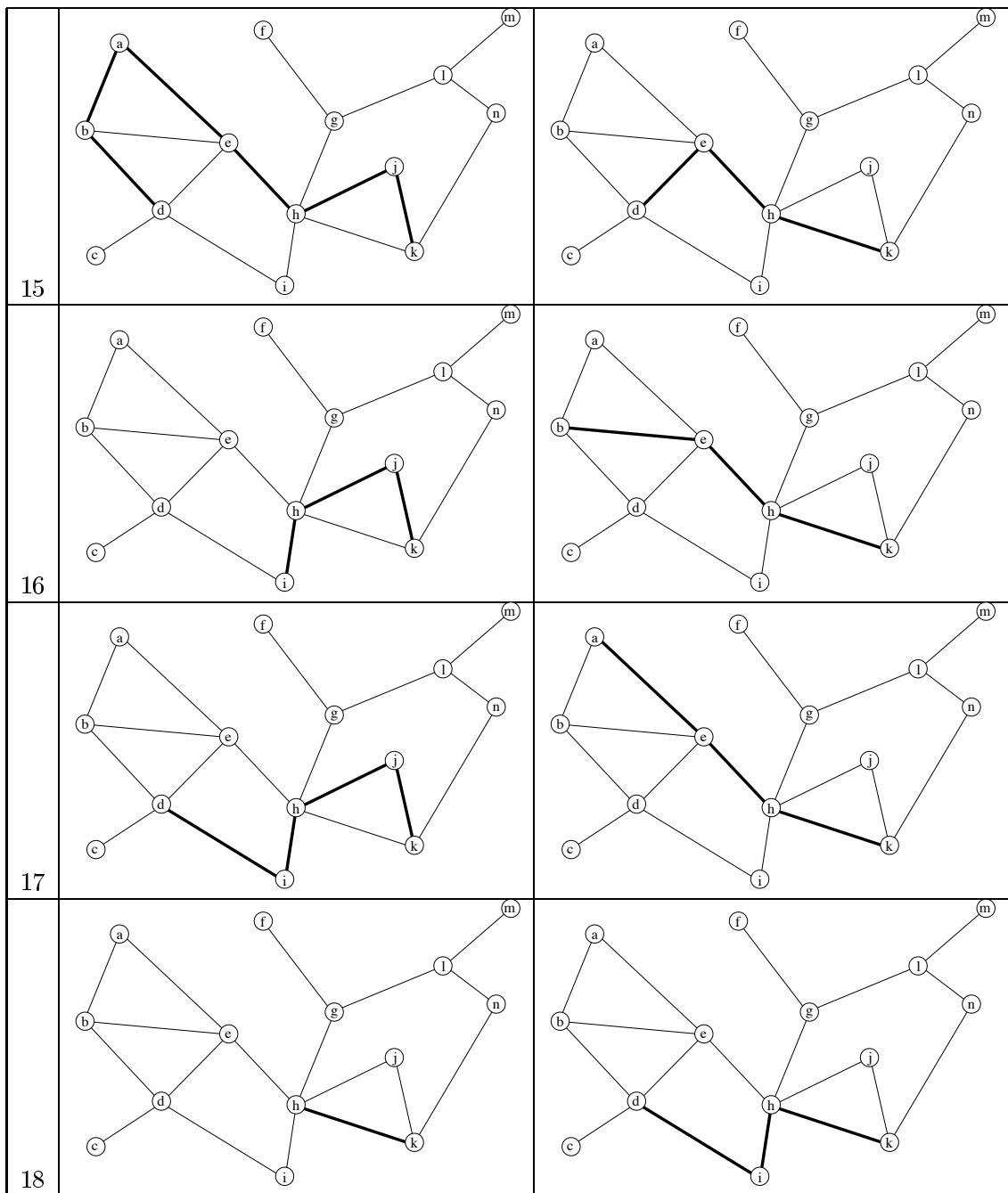
Die Tabelle zeigt nicht den gesamten Verlauf der Suche, d.h. nach Tabellemente sind die Agenden der Suchen noch nicht leer. Ich hoffe aber, dass es trotzdem genug zu sehen gibt.

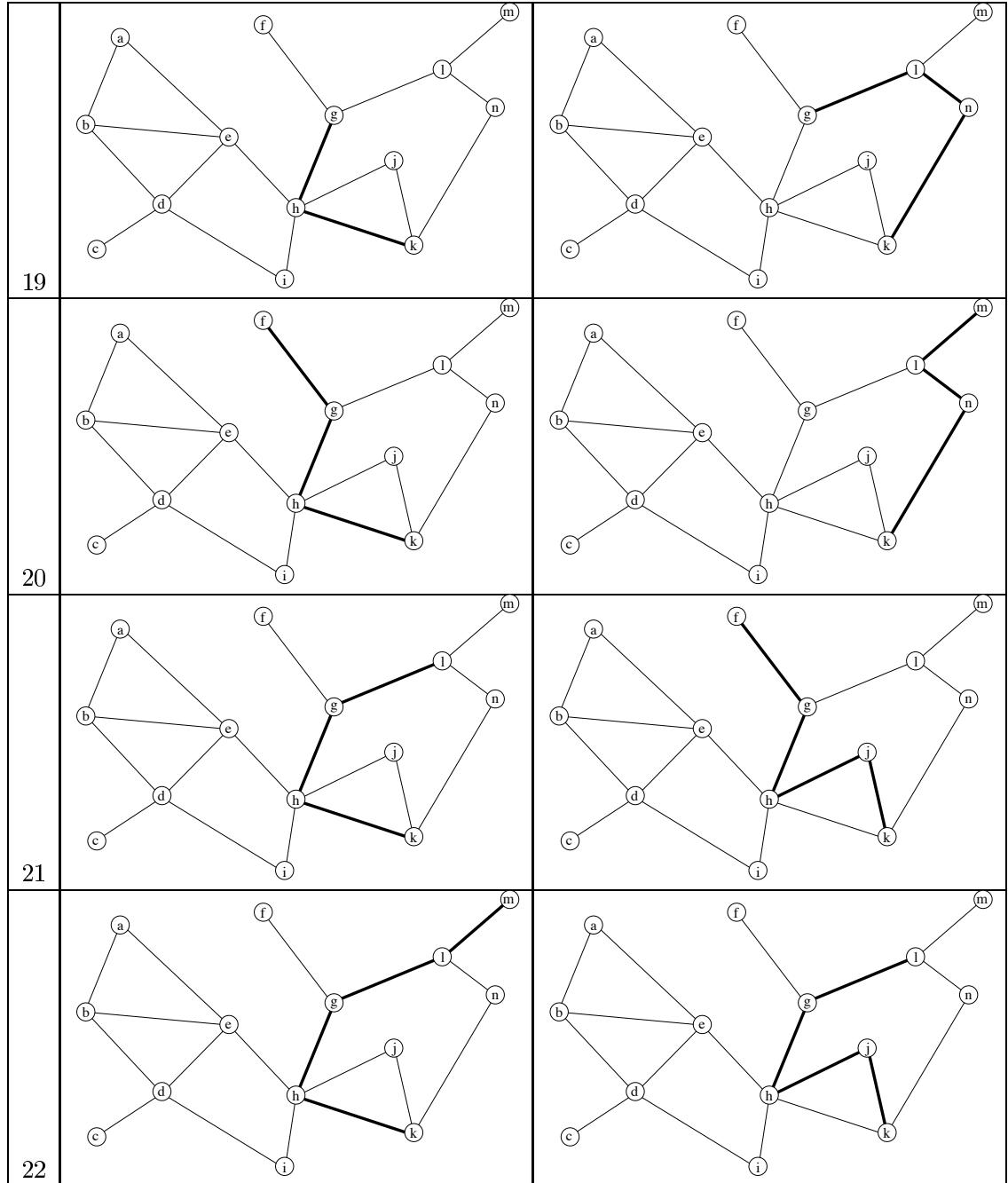


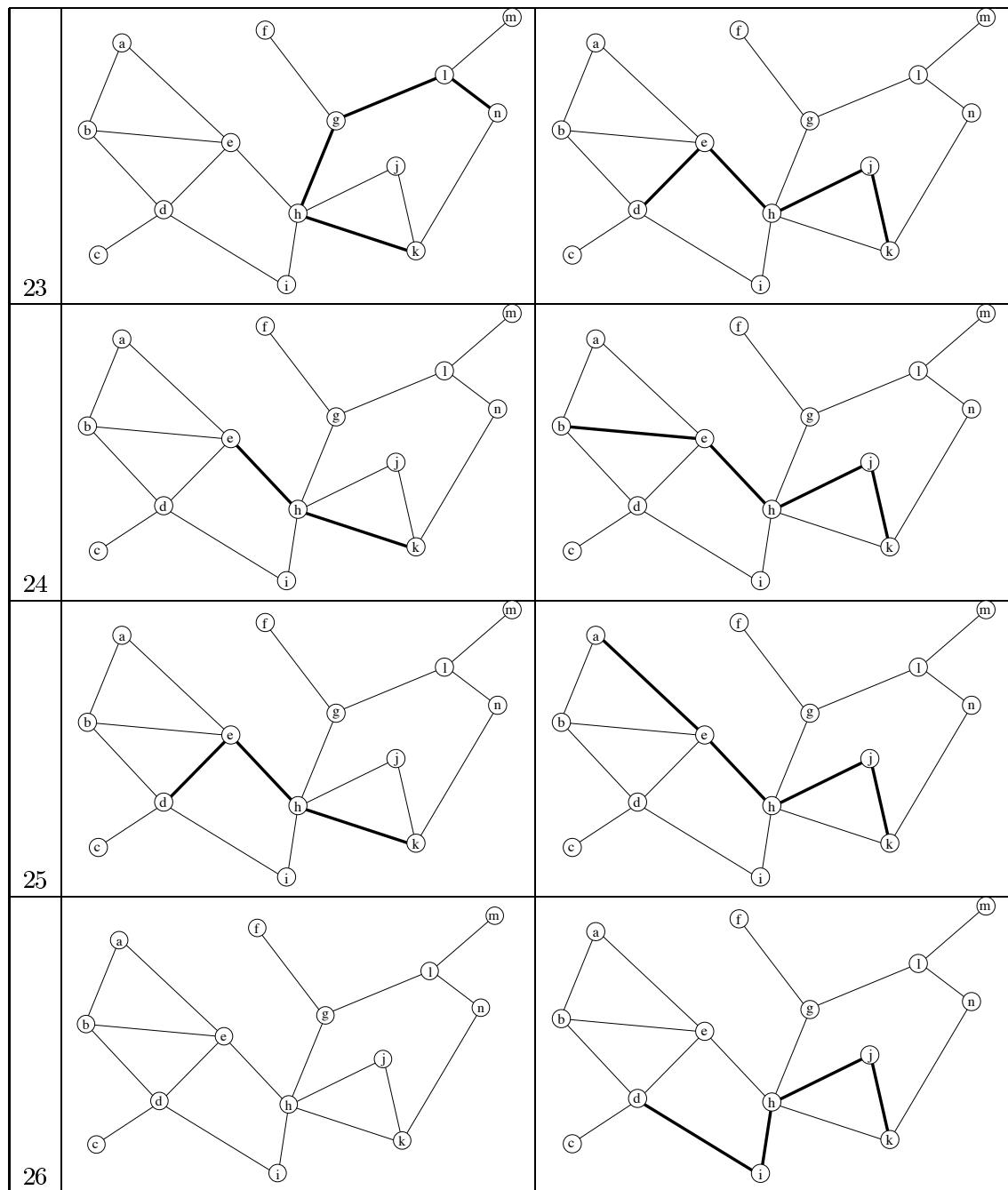












### 8.3.9 Best-First

Heuristik

### 8.3.10 Vergleich der drei Verfahren

Die ersten beiden Verfahren werden als *blinde* Suchverfahren bezeichnet, da sie

...

Laufzeit/Speicherplatz

### 8.3.11 Ausblick

iterative Deepening Nearest Neighbour A\*

## 8.4 to do

Eimerproblem mit diesen Algorithmen lösen ...

# Kapitel 9

## Sprachverarbeitung

In diesem Kapitel wird einführend gezeigt, wie schriftliche natürliche Sprache auf syntaktischer Ebene in Prolog analysiert werden kann.

### 9.1 Natürliche und formale Sprachen

Sprachen wie z.B. Englisch oder Deutsch werden als *natürliche Sprachen* bezeichnet. Gegenüber den natürlichen Sprachen gibt es künstliche, *formale Sprachen*, die von Menschen entworfen und auf strikte Art und Weise definiert werden. Die Prädikatenlogik und Programmiersprachen gehören zu den formalen Sprachen.

Die Theorie der formalen Sprachen kann auf einen Ausschnitt einer natürlichen Sprache angewendet werden, in dem der Ausschnitt formal modelliert wird. Es wird also eine formale Sprache erzeugt, die ein Fragment der natürlichen Sprache realisiert.

Eine formale Sprache ist definiert als die Menge der Sätze der Sprache. Sätze sind aus *Terminalsymbolen* bestehende Ketten. Die endliche Menge der Terminalen Symbole ist das Vokabular der Sprache.

“Aal”, “Aalsuppe”, “Abbild” und “Abklatsch” sind einige Terminalen Symbole der deutschen Sprache. Die Menge der Sätze, die sich aus deutschen Wörtern bilden lassen, ist unendlich groß. Es gibt Ketten, die aus Terminalen bestehen, jedoch keine Sätze sind, wie z.B. “Spucken Aalsuppe Osterhase.”

Unendliche Mengen können nicht durch Aufzählung, aber durch Produktionsregeln beschrieben werden. Eine *Grammatik* beschreibt durch solche Regeln, auf welche Weise gültige Sätze einer formalen Sprache erzeugt werden. Eine Grammatik kann verwendet werden, um korrekte Sätze zu generieren und um für gegebene Sätze zu prüfen, ob sie aus der Grammatik abgeleitet werden können.

Die Regeln einer Grammatik beschreiben die Form von Sätzen. Aufeinanderfolgende Terminalen Symbole lassen sich zu größeren Einheiten, den so genannten *Phrasen*, zusammenfassen. Aufeinanderfolgende Phrasen konstituieren größere Phrasen. Die größte Phrase ist der Satz. Alle Terminalen und Phrasen werden in solche Kategorien eingeteilt.

### 9.1.1 Ein Beispiel einer Grammatik

Die Terminale des folgenden Beispielsatzes lassen sich in die angegebenen Kategorien einteilen:

Der Weihnachtsmann bringt Geschenke.  
 DET N                    V                    N

Weihnachtsmann und Geschenke sind Nomen, kurz N. Der ist ein Artikel, der sich auf das Nomen Weihnachtsmann bezieht. In der Computerlinguistik werden Artikel oft als *Determiner*, kurz DET, bezeichnet. bringt ist ein Verb, V. Dies wird durch die folgenden Regeln formalisiert:

DET → [der]  
 N → [Weihnachtsmann]  
 N → [Geschenke]  
 V → [bringt]

DET kann also die Form der annehmen, N die Form Weihnachtsmann oder Geschenke und V die Form bringt. Diese Regeln bilden eine Kategorie auf erlaubte Terminalsymbole ab.

der Weihnachtsmann ist das Subjekt des Satzes, die beiden Terminale lassen sich zu einer so genannten *Nominalphrase* (NP) verbinden:

NP → DET N

Eine Nominalphrase kann also die Form DET gefolgt von N annehmen. Der Beispielsatz enthält eine weitere Nominalphrase, die ohne Artikel auskommt: Geschenke. In diesem Fall tritt eine NP als N in Erscheinung:

NP → N

Der Weihnachtsmann bringt Geschenke.  
 DET N                    V                    N  
 \ /    |  
 NP    NP

bringt ist das Prädikat des Satzes, Geschenke das zugehörige Objekt. Beide lassen sich zu einer *Verbalphrase* zusammenfassen.

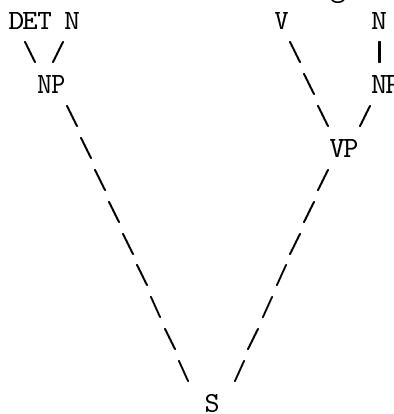
VP → V NP

Der Weihnachtsmann bringt Geschenke.  
 DET N                    V                    N  
 \ /    |  
 NP    NP  
 \ /  
 VP

Der Satz “Der Weihnachtsmann bringt Geschenke.” besteht also aus einer NP gefolgt von einer VP.

S → NP VP

Der Weihnachtsmann bringt Geschenke.



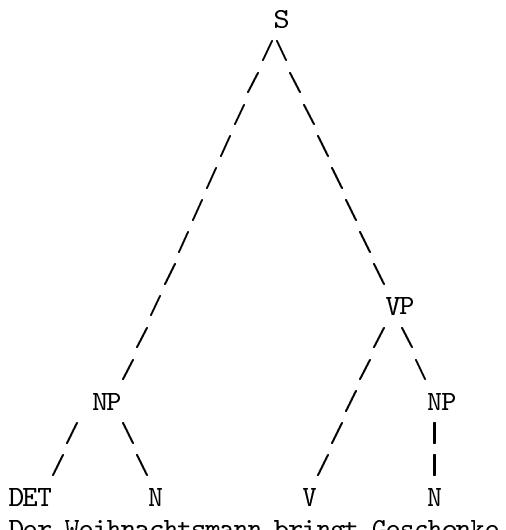
Wir haben folgende Grammatik konstruiert:

```

S -> NP NP
VP -> V NP
NP -> N
NP -> DET N
DET -> [der]
N -> [Weihnachtsmann]
N -> [Geschenke]
V -> [bringt]
  
```

Symbole wie S, NP, VP bezeichnen Kategorien von Phrasen und heißen *Nicht-terminalen*.

Wir haben folgende *Phrasenstruktur* für den Beispielsatz ermittelt:



Wir haben nicht nur *erkannt*, dass der Beispielsatz gemäß der Grammatik korrekt ist, sondern die Ableitung gleich mitgeliefert. Letzteres wird *parsen* genannt. Bäume dieser Art werden als Syntax- oder Phrasenstrukturbäume bezeichnet.

### 9.1.2 Kontextfreie Grammatiken

Die im vorherigen Beispiel aufgestellten *Phrasenstrukturregeln* erlauben die Ersetzung der linken Regelseite durch die rechte. Auf der linken Seite steht dabei ein einzelnes Nichtterminal. Auf der rechten Seite kann eine beliebige Folge von Terminalen und Nichtterminalen stehen. Grammatiken dieser Bauart gehören zur Klasse der *kontextfreien Grammatiken*. Die *Chomsky-Hierarchie* klassifiziert Grammatiken anhand des verwendeten Regeltyps und macht Aussagen über die Ausdrucksmächtigkeit und den Berechnungsaufwand für Ableitungen. Sie ist in nahezu jedem Buch über theoretische Informatik zu finden.

## 9.2 Kontextfreie Grammatiken in Prolog

Kontextfreie Grammatikregeln können direkt in Prologregeln überführt werden. Terminalen werden durch Atome, Phrasen als Listen von Terminalen repräsentiert. Der Beispielsatz lautet in dieser Schreibweise `[der, weihnachtsmann, bringt, geschenke]`. In doppelte Anführungszeichen eingeschlossene Zeichenketten ermöglichen die Repräsentation von großgeschriebenen Terminalen. In diesem Kapitel werden wir der Einfachheit halber alle Terminalen klein schreiben.

Die Phrasenstrukturregel  $S \rightarrow NP\ VP$  kann folgendermaßen in Prolog formuliert werden:

```
% S -> NP NP

s(Z) :- np(X),
        vp(Y),
        append(X,Y,Z).
```

X, Y und Z sind Listen von Terminalen. Z ist eine Phrase der Kategorie S, wenn X eine Phrase der Kategorie NP, Y eine Phrase der Kategorie VP und Z die Verkettung von X und Y ist.

Die anderen Regeln lassen sich analog umsetzen. Die Abbildung von Kategorien auf Terminalen geschieht durch Fakten.

```
% S -> NP NP

s(Z) :- np(X), vp(Y), append(X,Y,Z).

% VP -> V NP
vp(Z) :- v(X), np(Y), append(X,Y,Z).

% NP -> N
np(Z) :- n(Z).

% NP -> DET N
np(Z) :- det(X), n(Y), append(X,Y,Z).

% DET -> [der]
det([der]).
```

```
% N -> [Weihnachtsmann]
n([weihnachtsmann]).
```

```
% N -> [Geschenke]
n([geschenke]).
```

```
% V -> [bringt]
v([bringt]).
```

Dieses Prologprogramm ist ein *Erkenner* für die Beispielgrammatik. Der Beispielsatz wird folgendermaßen auf seine Ableitbarkeit geprüft:

```
?- ist([der, weihnachtsmann, bringt, geschenke]) ein korrekter Satz?
?- s([der, weihnachtsmann, bringt, geschenke]).
```

Yes

Der Syntaxbaum wird nicht mitgeliefert. Später werden wir das Programm zu einem *Parser* erweitern, der dies leistet.

Prolog beginnt seinen Beweis mit der Regel `s([...])` und arbeitet sich von dort zu den Terminalen herunter. Dieses Vorgehen wird top-down genannt. `s` wird in diesem Zusammenhang als *Startsymbol* bezeichnet.

Bei der Anfrage `s([der, weihnachtsmann, bringt, geschenke]).` erzeugt das System so lange Verkettungen von NPs und VPs bis das Resultat der übergebenen Liste entspricht. Dieses Vorgehen ist sehr ineffizient. Die Grammatik erzeugt die folgenden NPs und VPs:

<code>?- np(X).</code>	<code>?- vp(X).</code>
<code>X = [weihnachtsmann] ;</code>	<code>X = [bringt, weihnachtsmann] ;</code>
<code>X = [geschenke] ;</code>	<code>X = [bringt, geschenke] ;</code>
<code>X = [der, weihnachtsmann] ;</code>	<code>X = [bringt, der, weihnachtsmann] ;</code>
<code>X = [der, geschenke] ;</code>	<code>X = [bringt, der, geschenke] ;</code>
<code>No</code>	

Der Beweiser wählt als erste NP `[weihnachtsmann]` und kombiniert diese mit jeder der VPs. Dabei kann nie die Kette `[der, weihnachtsmann, bringt, geschenke]` entstehen. Beim Backtracking wird die NP `[geschenke]` gewählt und wiederum mit jeder VP kombiniert, obwohl keinerlei Erfolgsaussicht besteht. Erst danach wird `[der, weihnachtsmann]` herangezogen, der schließlich mit der zweiten VP den übergebenen Satz bildet. Alle VPs wurden mindestens zwei mal völlig umsonst generiert. Den "Fehler", dass einige der Phrasen (z.B. `[bringt, der, geschenke]`) keine Phrasen des Deutschen sind, werden wir später berichtigen.

Das Programm wäre weit effizienter, wenn es keine Phrasen probieren würde, die auf keinen Fall zu einem Ergebnis führen. Wenn `[der, weihnachtsmann, bringt, geschenke]` in eine NP und eine VP zerlegt werden soll, dann geht das nur, indem die NP ein Präfix dieser Liste ist. Das verbleibende Suffix der Liste muss eine VP konstituieren.

% Präfix (NP)	% Suffix (VP)
<code>[der]</code>	<code>[weihnachtsmann, bringt, geschenke]</code>
<code>[der, weihnachtsmann]</code>	<code>[bringt, geschenke]</code>
<code>[der, weihnachtsmann, bringt]</code>	<code>[geschenke]</code>
<code>[der, weihnachtsmann, bringt, geschenke]</code>	<code>[]</code>

Das NP-Prädikat kann so modifiziert werden, dass die Präfixe einer übergebenen Liste erkennt, die eine NP konstituieren und den überbleibenden Rest der Liste liefert, so dass dieser von weiteren Prädikaten verarbeitet werden kann.

Die neue Version von np bekommt zwei Argumente. Im ersten wird die Liste der Terminale übergeben. Das Prädikat knabbert nun ein passendes Präfix dieser Liste ab und liefert das daraus resultierende Suffix im zweiten Argument zurück.

Eine Anfrage könnte dann z.B. so aussehen:

```
?- np([der, weihnachtsmann, bringt, geschenke], Suf).
Suf = [bringt, geschenke]
Yes
```

np konsumiert das Präfix [der, weihnachtsmann] und liefert das Suffix [bringt, geschenke]. Somit wurden keine zum Scheitern verurteilten NPs generiert.

np delegiert die Aufgabe an det und n weiter. det veranschlagt ein Präfix der Wortliste für sich und liefert das entsprechende Suffix. n bekommt dieses Suffix als Wortliste und konsumiert nun seinerseits ein Präfix dieser Wortliste, wobei wiederum ein Suffix entsteht. Damit wäre eine NP als Präfix der an np übergebenen Liste gefunden. Das dazugehörige Suffix ist schon bekannt, es ist das Suffix, das n liefert hat.

Die alternative np-Regel delegiert die Arbeit einfach an n weiter.

Die neue Definition von np lautet:

```
np(L, Suf) :- det(L, L2), n(L2, Suf).
```

```
np(L, Suf) :- n(L, Suf).
```

Die Definitionen von det und n müssen angepasst werden.

```
det([der|X], X).
n([weihnachtsmann|X], X).
n([geschenke|X], X).
```

det bekommt eine Liste übergeben. Matcht der Kopf der Liste mit der, so wird das zweite Argument mit dem Rest der Liste instanziert. Die Definitionen der anderen terminale geschehen analog. Einige Beispielaufrufe:

```
?- det([der, weihnachtsmann, bringt, geschenke], Suf).
Suf = [weihnachtsmann, bringt, geschenke]
Yes
```

```
?- det([der, weihnachtsmann], Suf).
Suf = [weihnachtsmann]
Yes
```

```
?- det([der], Suf).
Suf = []
Yes
```

```
?- det([Rentierbraten], Suf).
No
```

Wir passen die gesamte Grammatik an und erhalten:

```
% S -> NP NP
s(L, Suf) :- np(L, L2), vp(L2, Suf).

% NP -> DET N
np(L, Suf) :- det(L, L1), n(L1, Suf).

% NP -> N
np(L, Suf) :- n(L, Suf).

% VP -> V NP
vp(L, Suf) :- v(L, L2), np(L2, Suf).

% Terminale
det([der|X], X).
n([weihnachtsmann|X], X).
n([geschenke|X], X).
v([bringt|X], X).
```

Die Aufrufe von `append/3` sind aus den Regeln verschwunden.

Um nun zu prüfen, ob `[der, weihnachtsmann, bringt, geschenke]` ein Satz ist, muss `s/2` mit dieser Liste und einer leeren Liste aufgerufen werden. Das zweite Argument entspricht dem Suffix, das über ist, wenn das Präfix der Liste der Kategorie S entspricht. Da von dem Satz nichts über bleiben soll, muss das Suffix leer sein.

```
?- s([der, weihnachtsmann, bringt, geschenke], []).  
Yes
```

### 9.2.1 DCG

Die meisten Prologsysteme bieten einen Formalismus an, der kontextfreie Grammatikregeln automatisch in Prologprogramme übersetzt und den Namen *Definite Clause Grammar* trägt.

Die Beispielgrammatik lautet in DCG-Schreibweise:

```
s --> np, vp.  
np --> det, n.  
np --> n.  
vp --> v, np.  
  
det --> [der].  
n --> [weihnachtsmann].  
n --> [geschenke].  
v --> [bringt].
```

Diese Definition ist sehr einfach lesbar und lässt sich direkt laden. Prolog übersetzt sie in Regeln, die denen entsprechen, die wir im Verlaufe des Kapitels per Hand erstellt haben.

```
%  
?- [dcg-beispiel].  
% dcg-beispiel.pl compiled 0.00 sec, 1,800 bytes
```

Yes

```
% Listing der aus der DCG erzeugten Prädikate:  
?- listing([s,np,vp,det,n,v]).
```

```
s(A, B) :-  
    np(A, C),  
    vp(C, B).
```

```
np(A, B) :-  
    det(A, C),  
    n(C, B).  
np(A, B) :-  
    n(A, B).
```

```
vp(A, B) :-  
    v(A, C),  
    np(C, B).
```

```
det([der|A], A).
```

```
n([weihnachtsmann|A], A).  
n([geschenke|A], A).
```

```
v([bringt|A], A).
```

Yes

```
% Beispielsatz  
?- s([der, weihnachtsmann, bringt, geschenke], []).
```

Yes

DCG macht die Erstellung kontextfreier Grammatiken in Prolog bequem.

### 9.2.2 Generierung

Eine Grammatik kann nicht nur zum Erkennen oder Parsen von Sätzen, sondern auch zu ihrer Erzeugung eingesetzt werden.

Eine Anfrage mit einem uninstanziertem ersten Argument führt dazu, dass sukzessive alle durch die Grammatik erzeugbaren Sätze generiert werden:

```
?- s(X, []).  
X = [der, weihnachtsmann, bringt, der, weihnachtsmann] ;  
X = [der, weihnachtsmann, bringt, der, geschenke] ;  
X = [der, weihnachtsmann, bringt, weihnachtsmann] ;  
X = [der, weihnachtsmann, bringt, geschenke] ;  
X = [der, geschenke, bringt, der, weihnachtsmann] ;  
X = [der, geschenke, bringt, der, geschenke] ;  
X = [der, geschenke, bringt, weihnachtsmann] ;  
X = [der, geschenke, bringt, geschenke] ;  
X = [weihnachtsmann, bringt, der, weihnachtsmann] ;  
X = [weihnachtsmann, bringt, der, geschenke] ;
```

```
X = [weihnachtsmann, bringt, weihnachtsmann] ;
X = [weihnachtsmann, bringt, geschenke] ;
X = [geschenke, bringt, der, weihnachtsmann] ;
X = [geschenke, bringt, der, geschenke] ;
...
X = [der, weihnachtsmann, bringt, geschenke] ;
...
```

Dabei wird sichtbar, ob die Grammatik die geforderten Sätze abdeckt ([der, weihnachtsmann, bringt, geschenke]) und ob sie sinnlose oder gar ungrammatische<sup>1</sup> Sätze produziert.

### 9.2.3 Zusätzliche Argumente

Die Beispielgrammatik generiert viele ungrammatische Sätze, in denen die erforderliche Kongruenz zwischen Kasus, Numerus und Genus verletzt ist. Die Grammatik muss angepasst werden. Dies könnte über neue Kategorien und Regeln gelöst werden. Beispielsweise könnten feminine, maskuline und sächliche Nominativ- und Akkusativ-NPs eingeführt werden, um die Bildung ungrammatischer Sätze auszuschliessen. Das würde jedoch die Regelmenge aufblättern und unübersichtlich machen.

DCG erlaubt es, Kategorien mit zusätzlichen Argumenten und damit Merkmalen zu versehen. Wie wir sehen werden, lässt sich die KNG-Kongruenz damit recht elegant erzwingen.

Zunächst statten wir mal den Weihnachtsmann mit Kasus, Numerus und Genus aus. Dazu verwenden wir willkürlich gewählte, aber konsistent benutzte Konstanten:

```
% weihnachtsmann = Nomen (Nominativ, singular, maskulin)
n(nom,sing,mask) --> [weihnachtsmann].
```

Auf die gleiche Weise werden die übrigen Terminale angepasst:

```
% der = Artikel (Nominativ, singular, maskulin)
det(nom,sing,mask) --> [der].
```

```
% geschenke = Nomen (Akkusativ, plural, feminin)
n(akk,plur,fem) --> [geschenke].
```

```
% bringt = Verb (dritte Person, singular)
v(3,sing) --> [bringt].
```

Von einer NP fordern wir nun durch den Einsatz von Variablen, dass alle Konstituenten im gleichen Kasus, Numerus und Genus auftreten:

```
np(K,N,G) --> det(K,N,G), n(K,N,G).
np(K,N,G) --> n(K,N,G).
```

```
% Beispieldaufruf:
?- np(K,N,G,Phrase,[]).
K = nom
```

---

<sup>1</sup>im Sinne unseres Sprachverständnisses

```

N = sing
G = mask
Phrase = [der, weihnachtsmann] ;

K = akk
N = plur
G = fem
Phrase = [geschenke] ;

K = nom
N = sing
G = mask
Phrase = [weihnachtsmann] ;

```

No

VPs besitzen nun die Attribute Person und Numerus und bestehen aus einem passenden V und einer NP im Akkusativ:

```
vp(P,N) --> v(P,N), np(akk,_,_).
```

Ein Satz wird konstituiert von einer NP im Nominativ und einer im Numerus passenden VP. Die gesamte Grammatik lautet:

```

s --> np(nom,N,_), vp(3,N).
np(K,N,G) --> det(K,N,G), n(K,N,G).
np(K,N,G) --> n(K,N,G).
vp(P,N) --> v(P,N), np(akk,_,_).

det(nom,sing,mask) --> [der].
n(akk,plur,fem) --> [geschenke].
v(3,sing) --> [bringt].
n(nom,sing,mask) --> [weihnachtsmann].

```

```

% Alle möglichen Sätze generieren:
?- s(X,[]).
X = [der, weihnachtsmann, bringt, geschenke] ;
X = [weihnachtsmann, bringt, geschenke] ;
no

```

#### 9.2.4 Einfügen von Prolog-Code

Es ist möglich, beliebigen Prolog-Code auf der rechten Seite von DCG-Regeln einzufügen. Dieser muss in geschweifte Klammern eingeschlossen werden und wird bei der Übersetzung von DCG nach Prolog unverändert übernommen. Auf diese Weise können zusätzliche Testbedingungen an Regeln geknüpft und mehrere Regeln zusammengefasst werden.

```

n(nom,plur,fem) --> [geschenke].
n(gen,plur,fem) --> [geschenke].
n(akk,plur,fem) --> [geschenke].

```

```
% Die obigen drei Regeln lassen sich zusammenfassen:
n(Kas,plur,fem) --> [geschenke], {member(Kas,[nom,gen,akk])}.
```

Werden mehrere Prädikate verwendet, so müssen diese durch Kommas getrennt werden. Der Code in den geschweiften Klammern wird nicht mit einem Punkt abgeschlossen.

### 9.2.5 Anmerkungen

Beim Schreiben von Grammatiken wird versucht, geeignete Regeln für das zu beschreibende *Fragment* einer Sprache aufzustellen. Dabei gibt es nie nur eine einzige Lösung. Kategorien und deren Bezeichnungen werden willkürlich gewählt. Auch wenn es einige Kategorien und Regeln gibt, die in vielen Quellen als Beispiele auftauchen, sind diese nicht verbindlich.

Es gibt viele Möglichkeiten, die Beispielgrammatik zu erweitern, um einen größeren Teil des Deutschen abzudecken, beispielsweise auf VPs ohne oder mit zwei Objekten:

```
s --> np(nom,N,_), vp(3,N).

np(K,N,G) --> det(K,N,G), n(K,N,G).
np(K,N,G) --> n(K,N,G).

vp(P,N) --> v0(P,N).
vp(P,N) --> v(P,N), np(akk,_,_).
vp(P,N) --> v2(P,N), np(dat,_,_), np(akk,_,_).

det(nom,plur,fem) --> [die].
det(akk,plur,fem) --> [die].
det(nom,sing,mask) --> [der].
det(nom,plur,neut) --> [die].
det(dat,plur,neut) --> [den].

n(Kas,plur,fem) --> [geschenke], {member(Kas,[nom,gen,akk])}.
n(Kas,plur,neut) --> [kinder], {member(Kas,[nom,gen,akk])}.
n(dat,plur,neut) --> [kindern].
n(nom,sing,mask) --> [weihnachtsmann].

v0(3,plur) --> [singen].
v(3,sing) --> [bringt].
v(3,sing) --> [bringt].
v2(3,sing) --> [gibt].
```

Semantische Restriktionen, wie z.B., dass Geschenke nicht singen können, können über weitere Argumente und Tests umgesetzt werden.

### 9.2.6 Syntaxstrukturen

Unsere bisherige Beispielgrammatik konnte grammatische Sätze erkennen und generieren. Ein *Parser* liefert zusätzlich noch die syntaktische Struktur, also

den Ableitungsbaum. Durch zusätzliche Argumente in den Regeln kann diese beim Prolog-Beweis mit aufgesammelt werden. Jede Kategorie liefert in ihrem ersten Argument eine Beschreibung von sich selbst, die sich wiederum aus den Beschreibungen der Unterkategorien zusammensetzt.

```
s(satz(NP,VP)) --> np(NP, nom,N,_), vp(VP, 3,N).

np(np(DET,NOM), K,N,G) --> det(DET, K,N,G), n(NOM, K,N,G).
np(np(NOM), K,N,G) --> n(NOM, K,N,G).

vp(vp(V,np), P,N) --> v(V, P,N), np(NP, akk,_,_).

det(det([nom,sing,mask]:der), nom,sing,mask) --> [der].

n(n([akk,plur,fem]:geschenke), akk,plur,fem) --> [geschenke].
n(n([nom,sing,mask]:weihnachtsmann), nom,sing,mask) --> [weihnachtsmann].

v(v([3,sing]:bringt), 3,sing) --> [bringt].
```

```
% Struktur von [der, weihnachtsmann, bringt, geschenke]
?- s(Struktur,[der, weihnachtsmann, bringt, geschenke],[]).
Struktur = satz(np(det([nom, sing, mask]:der),
                  n([nom, sing, mask]:weihnachtsmann)),
                vp(v([3, sing]:bringt),
                  np(n([akk, plur, fem]:geschenke))))
Yes
```

```
% baumähnlicher:
satz(
  np(
    det([nom, sing, mask]:der),
    n([nom, sing, mask]:weihnachtsmann)
  ),
  vp(
    v([3, sing]:bringt),
    np(
      n([akk, plur, fem]:geschenke)
    )
  )
)
```

\*\*\* Grafische Darstellung \*\*\*

### 9.2.7 Ambiguität

Der Satz

Peter fesselt den Mann mit der Krawatte.

hat zwei *Lesarten*. Entweder Peter benutzt eine Krawatte um den Mann zu fesseln, oder Peter fesselt den Mann, der eine Krawatte trägt. Diese Mehrdeutigkeit wird *Ambiguität* genannt. Werden beide Lesarten von der Grammatik abgedeckt, so erhält man auch beide Parsebäume.

\*\*\* Grammatik

\*\*\* beide Bäume grafisch

Viele Sätze sind hochgradig ambig, besitzen aber für Menschen eine bevorzugte Lesart.

## 9.3 Todo

-Beispiel: Grammatik für englische Zahlen

-Stringkonstanten in doppelten Anführungszeichen, um Großschreibung zu ermöglichen

-Links- und, Rechtsrekursion (s-; s, und, s)

-freie Wortstellung im Deutschen

-Einlesen von Zeichenketten von stdin -; liste ...-; io!

-Adventure?



# Kapitel 10

## Ein- und Ausgabe

Prolog stellt Prädikate zur Ein- und Ausgabe von Termen und Zeichen bereit. Solche Prädikate liegen außerhalb der Logik, lassen sich jedoch dank der prozeduralen Semantik des Systems sinnvoll einsetzen. (Vgl. 4.3.2 auf Seite 65.) Neben dem hier vorgestellten Edinburgh-Standart besitzen viele Prologsysteme zusätzlich leistungsfähigere Prädikate.

In der Ein- und Ausgabe nach Edinburgh-Manier gibt es im System genau einen Ein- und genau einen Ausgabestrom. Diese Ströme können nach Belieben mit Dateien, bzw. Tastatur und Bildschirm verbunden werden. Per Voreinstellung erfolgt die Ausgabe auf dem Bildschirm, während die Eingabe von der Tastatur besorgt wird.

In diesem Kapitel werden nur die einfachsten Handgriffe vorgeführt. Für alles weitere sei auf das SWI-Manual verwiesen.

### 10.1 Lesen und Schreiben von Termen

#### 10.1.1 Lesen von Termen: `read/1`

Das Prädikat `read` liest vom Eingabestrom (standartmäßig Tastatur) einen Prolog-Term. Die Eingabe muss mit einem `.` und einem Return oder Leerzeichen abgeschlossen werden. Das System wartet (blockiert) solange, bis die Eingabe erfolgt ist. Konnte kein gültiger Term gelesen werden, wird eine Fehlermeldung ausgegeben. Der gelesene Term wird mit dem Argument unifiziert. `read` gelingt nur einmal, wird es erneut angestossen, liefert es den nächsten Term. `read` berücksichtigt ebenso wie `write` aktuell vorhandene Operatordefinitionen.

```
?- read(abc).  
|: abc.  
Yes  
  
?- read(Term).  
|: 123.  
Term = 123  
Yes
```

```
?- read(Term).
|: atom
|: .
Term = atom
Yes

?- read(Term).
|: interessant(wasanderes).
Term = interessant(wasanderes)
Yes

?- read(Term).
|: ich glaub, ich wer' zu'n Schwein!
|: .
ERROR: Stream user_input:70: Syntax error: Operator expected

?-
```

### 10.1.2 Schreiben von Termen: `write/1`

`write` gibt sein Argument auf dem Ausgabestrom (voreingestellt ist der Bildschirm) aus. Dabei werden — anders als bei `display/1` — Operatordefinitionen berücksichtigt.

```
?- _X= a*b+c/d, write(_X), nl, display(_X).
a*b+c/d
+(*(a, b), /(c, d))
Yes

?- write(hallo).
hallo
Yes

?- write(X).
_G186
X = _G186
Yes

?- write(ausserlogisch(io)).
ausserlogisch(io)
Yes
```

## 10.2 Lesen und Schreiben von Zeichen

### 10.2.1 Lesen von Zeichen: `get/1`, `get0/1`

`get/1` und `get0/1` lesen ein Zeichen vom Eingabestrom und unifizieren es mit dem Argument. Ist das Argument uninstantiiert, gelingen sie immer. Beide Prädikate blockieren, bis ein Zeichen verfügbar ist und bieten keine Alternativen.

`get/1` liefert das nächste anzeigbare Zeichen, überliest also Steuerzeichen wie etwa ‘Return’. `get0/1` liefert das nächste Zeichen.

```
?- get(X).
|: a
X = 97
Yes

?- get(X).
|: % Return
|: % Return
|: % Return
|: .
X = 46
Yes

?- get0(X).
|: % Return
X = 10
Yes
```

### 10.2.2 Schreiben von Zeichen: `put/1`

`put/1` schreibt ein Zeichen (ASCII) auf den Ausgabestrom. `put` gelingt immer und hat keine Alternativen.

```
?- put(65).
A
Yes

?- put(X).
ERROR: Arguments are not sufficiently instantiated

?- put(a).
a
Yes

?- put(abc).
ERROR: Type error: ‘character’ expected, found ‘abc’

?- put('a').
a
Yes
```

### 10.2.3 Tabulatoren und Zeilenumbruch

Das Prädikat `n1/0` bewegt den Cursor an den Anfang der nächsten Zeile. `tab/1` erzeugt die angegebene Anzahl Leerzeichen.

```
?- tab(7), put(42), nl, tab(23), put(58), put(45), put(41).
*  
:-)
```

Yes

### 10.3 Ein- und Ausgabe auf Dateien

Der Eingabestrom kann mit dem Prädikat `see/1` auf eine Datei gesetzt werden. `see(user).` setzt den Strom zurück auf die Tastatur. `seeing/1` liefert, auf was der Eingabestrom gesetzt ist. `seen/0` schließlich schließt den Strom. Analog dazu öffnet `tell/1` eine Datei zum Schreiben, `telling/1` gibt Auskunft und `told/0` schließt die Datei. Das Atom `end_of_file` wird geliefert, wenn das Dateiende erreicht worden ist.

```
% in Datei schreiben
?- tell('test').
Yes
```

```
?- write(ist_ein(dies, test)), write('.'), nl.
Yes
```

```
?- write(daswars), write('.').
Yes
```

```
?- told.
Yes
```

```
% aus Datei lesen
?- see('test').
Yes
```

```
?- read(X).
X = ist_ein(dies, test) ;
No
```

```
?- read(X).
X = daswars
Yes
```

```
?- read(X).
X = end_of_file
Yes
```

```
?- read(X).
X = end_of_file
Yes
```

```
?- see(user).  
Yes
```

```
?- read(X).  
|:
```

Selbstverständlich können auch die Prädikate zur Ein-/Ausgabe von Zeichen genutzt werden. `get0` und `get` liefern `-1` wenn keine weiteren Zeichen aus einer Datei gelesen werden können.



## Kapitel 11

# Dynamische Änderung der Wissensbasis

Die Wissensbasis kann zur Laufzeit eines Prologprogrammes geändert werden, d.h. es ist möglich, neue Klauseln hinzuzufügen, bzw. zu entfernen. Da die Änderungen nicht durch das Backtracking rückgängig gemacht werden, können Informationen über ihre normale Lebensdauer hinaus haltbar gemacht werden. Es lässt sich auf diese Weise mit sogenannten *Seiteneffekten* arbeiten.

### 11.1 `asserta/1`, `assertz/1` und `retract/1`

Mit den Prädikaten `assert`, `asserta`, `assertz` lässt sich eine neue Klausel in die Wissensbasis schreiben. `asserta` fügt die neue Klausel dabei als erste Klausel des betreffenden Prädikates ein, `assertz` fügt die neue Klausel als letzte Klausel des Prädikates ein. `assert` ist äquivalent zu `assertz`.

Das Argument — die einzufügende Klausel — muss soweit instantiiert sein, dass das System erkennen kann, um welches Prädikat es sich handelt.

Die Effekte dieser Prädikate werden nicht vom Backtracking rückgängig gemacht. Möchte man eine Klausel aus der Wissensbasis entfernen, kann man dies mit `retract/1` tun. `retract/1` löscht die erste mit dem Argument matchende Klausel aus der Wissensbasis. Das Prädikat gelingt nur einmal, wird es erneut aufgerufen, so löscht es die nächste matchende Klausel. Gelöschte Klauseln werden durch das Backtracking nicht wieder hergestellt.

Es handelt sich um ausserlogische Prädikate.

#### 11.1.1 `consult/1`

Ursprünglich war `assert` dazu gedacht, Prologprogramme in den Speicher zu laden. Es wird eine Datei geöffnet, die den Prolog-Code enthält. Anschließend werden solange Terme gelesen und mit `assert` in die Wissensbasis geschrieben, bis das Dateiende erreicht ist. Handelt es sich bei einem Term um eine Direktive, so muss diese ausgeführt statt eingefügt werden.

```
myconsult(File) :-
```

```

seeing(Oldfile),          % alte Verbindung merken
see(File),               % Datei öffnen
repeat,                  % unendlich viele Choicepoints
read(Term),              % Term lesen
process(Term),           % Term verarbeiten
seen,                    % Datei schließen
see(Oldfile),            % Eingabestream auf alte Verbindung zurücksetzen
! .                      % myconsult darf nur einmal gelingen

process(end_of_file) :-  % Dateiende erreicht?
! .                      % Ja: Ab jetzt gibt es keine Alternativen
                           % mehr für process.

process(:Directive) :-   % Term hat die Form :-Goal soll also ausgeführt werden
! ,                      % Wenn wir in dieser Klausel landen, so wollen wir
                           % die anderen Klauseln nicht mehr beachten
call(Directive),         % Direktive ausführen
! ,                      % Alternativen der Direktive abschneiden
fail.                   % scheitern, und damit Backtracking erzwingen

process(Klausel) :-      % Bei Term handelt es sich um eine Klausel
assert(Klausel),         % Klausel in die Wissensbasis einfügen
fail.                   % scheitern, und damit Backtracking erzwingen

```

Backtrackingschleifen wurden auf im Abschnitt 5.8.1 auf Seite 79 vorgestellt.

Das Prädikat **repeat** stellt unendlich viele Choicepoints zur Verfügung. (**repeat**, **wiederhole** siehe Seite 79.)

Wenn wir das Öffnen und Schließen der Datei einmal weglassen, bleibt von **myconsult** folgendes übrig:

```

myconsult(File) :-
repeat,
read(Term),
process(Term),
! .

```

**repeat** hat unendlich viele Alternativen. **read(Term)** gelingt nur einmal und hat keine Alternativen. **process(Term)** ‘verarbeitet’ den **Term**. Wenn **process** scheitert, führt das Backtracking zurück zu **repeat**. Dort gibt es immer eine Alternative. Nachdem eine **repeat**-Alternative gewählt wurde, ist wiederum zuerst **read(Term)** (neuen **Term** einlesen) und anschließend **process(Term)** zu beweisen. Jedes Scheitern von **process(Term)** führt zu einem neuen Durchlauf dieser Backtrackingschleife.

Wenn **process** schließlich gelingt, wird der Cut überschritten und **myconsult** gelingt. Der Beweis geht erfolgreich zu Ende, ohne dass noch ein Alternative für **myconsult** möglich wäre. Damit gelingt **myconsult** genau einmal.

Die Steuerung der Schleife hängt also von **process** ab. **process(Term)** führt eine Fallunterscheidung durch: **Term** kann eine Klausel, eine Direktive

oder das Dateiendezeichen sein. Die Cuts sorgen dafür, dass nur eine der Klauseln gewählt werden kann.

```
process(end_of_file) :-  
    !.  
  
process((:-Directive)) :-  
    !,  
    call(Directive),  
    !,  
    fail.  
  
process(Klausel) :-  
    assert(Klausel),  
    fail.
```

Im Falle der Dateiendezeichen- Klausel gelingt `process`. Damit findet die Backtrackingschleife ein Ende.

Ist `Term` eine Direktive (2. Klausel), wird sie mit dem Builtin `call` aufgerufen und eventuelle Alternativen des Goals abgeschnitten. Das abschließende `fail` kann nicht vor den Cut zurückgelangen, bewirkt also Backtracking bis hinauf zu `repeat` in `myconsult`.

Die dritte Klausel von `process(Term)` wird gewählt, falls `Term` eine Klausel ist. `Term` wird in die Wissensbasis eingefügt und das folgende `fail` veranlasst Backtracking zu `repeat` in `myconsult`, da `assert` keine Alternativen hat. Der Effekt von `assert` wird durch das Backtracking nicht rückgängig gemacht. Daher kann die Backtrackingschleife genutzt werden, um die ganze Datei `Term` für `Term` zu lesen und abzuarbeiten.

### 11.1.2 `findall/3`

TODO

## 11.2 Weitere Prädikate zum Zugriff auf die Wissensbasis

listing

```
clause(X,Y)  
nochmal auf functor und arg in kap2 verweisen
```

## 11.3 todo

Seiteneffekte

nicht definiert bei ... z.B.: `a(1) :- write('hallo'), asserta((a(1):-write('bye'))), fail.`

verwandtschaft zu Datenbanken  
dynamic, static, Compilierung,...

benötigt für consult, reconsult, findall, gensym  
-fibonacci mit cache  
clause()?

# GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.  
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA  
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## Preamble

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 11.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, L<sup>A</sup>T<sub>E</sub>X input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

## 11.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

### 11.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

### 11.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together

with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled “History”, and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled “Acknowledgements” or “Dedications”, preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
- Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the

Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 11.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

## 11.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 11.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

## 11.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

## 11.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 11.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License or any later version applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

# Index

- Platzhalter, 6
  - anonym, 8
  - Ausgabe unterdrücken, 8
  - gemeinsame, 7
  - in Anfragen, 5
  - in Listen, 46
  - in Regeln, 10, 18
  - in Strukturen, 28–30, 33
  - mit Strukturen belegen, 30, 32
  - sind Terme, 27
  - Unifikation, 37
- Unifikation
  - das Unifikationsverfahren, 37
  - Funktionen der, 37
  - von Listen, 46
- Variable, *siehe* Platzhalter