PROLOG

Eine Einführung

Sven Naumann

Sommersemester 2007

Inhaltsverzeichnis

I Logikprogrammierung 7

1 Grundbegriffe 11
- 1.1 Prolog und Logikprogrammierung . . . 11
- 1.2 Terme . . . 13
- 1.3 Klauseln . . . 13
  - 1.3.1 Fakten . . . 14
  - 1.3.2 Fragen . . . 15
  - 1.3.3 Regeln . . . 18
- 1.4 Ein einfacher abtrakter Interpreter für Logikprogramme . . . 21
- 1.5 Die Bedeutung eines Logikprogramms . . . 23

2 Unifikation 25
- 2.1 Der Unifikationalgorithmus . . . 25
- 2.2 Ein Interpreter für Logikprogramme . . . 28
- 2.3 Suchbäume . . . 31
- 2.4 Prologs Beweisstrategie . . . 32

II Programmieren in Prolog 35

3 Erste Schritte 37
- 3.1 Benutzung des Prolog-Interpreter . . . 37
  - 3.1.1 Aufrufen und Verlassen des Interpreters . . . 37
  - 3.1.2 Erstellung von Programmen . . . 38
  - 3.1.3 Anzeige der Wissensbasis . . . 38
- 3.2 Programmentwicklung . . . 40
  - 3.2.1 Schritte der Programmentwicklung . . . 40
  - 3.2.2 Gestaltung von Programmen . . . 40
  - 3.2.3 Programmdokumentation . . . 41
- 3.3 Das Affe/Banane–Problem . . . 41
  - 3.3.1 Das Problem . . . 41
  - 3.3.2 Vorüberlegungen . . . 42
  - 3.3.3 Das Programm . . . 43

4 Listen **45**
4.1 Notationskonventionen 45
4.1.1 Standardnotation 45
4.1.2 Operator–Notation 46
4.1.3 Erweiterte Standardnotation 47
4.2 Operationen auf Listen 47
4.2.1 Ein Typ–Prädikat für Listen 48
4.2.2 Suchen eines Objekts in einer Liste 48
4.2.3 Entfernen von Objekten aus einer Liste 49
4.2.4 Die *Append*–Relation 50
4.2.5 Die Teillisten–Relation 50
4.3 Multifunktionalität von Prädikaten 52

5 Arithmetik **55**
5.1 Arithmetische Operatoren und Systemprädikate 55
5.2 Zahlenspiele 58

6 Strukturen **61**
6.1 Binäre Bäume 61
6.2 Strukturuntersuchungen 62
6.2.1 Typprädikate 62
6.2.2 Analyse von Termen 65

7 Endliche Automaten **73**
7.1 Grundlegende Konzepte 73
7.2 Repräsentationsmöglichkeiten 75
7.3 Repräsentation endlicher Automaten in Prolog 76
7.4 Recognizer, Generator und Parser 78
7.4.1 Ein Recognizer für Übergangsnetzwerke 78
7.4.2 Ein Generator für Übergangsnetzwerke 80
7.4.3 Ein Parser für Übergangsnetzwerke 80
7.5 Transducer 81

8 Metalogische Prädikate **83**
8.1 Metalogische Typprädikate 84
8.2 Der Vergleich offener Terme 88
8.3 Metavariablen 90

9 Kontrolliertes Backtracking **93**
9.1 Der Cut–Operator 93
9.1.1 Grüne Cuts 94
9.1.2 Rote Cuts 97
9.2 Negation in Prolog 99

10 Ein- und Ausgabe **103**
10.1 Dateien 103
10.1.1 Dateiorganisation 105
10.1.2 Programm–Dateien 106
10.2 Verarbeitung von Termen 109
10.2.1 Formatierung von Termen 111
10.2.2 Verarbeitung von Dateien von Termen 113
10.3 Verarbeitung von Zeichen 114
10.4 Zerlegen und Aufbauen von Atomen 115

11 Weitere Systemprädikate **117**
11.1 Datenbasismanipulation 117
11.2 Iterative Programmierung in Prolog 119
11.3 Ein Prädikat für alle Lösungen 121
11.4 Interaktive Programme 122
11.4.1 Ein zeilenorientierter Editor 122
11.4.2 Eine interaktive Shell mit Protokoll 124

12 Operatoren und Differenzlisten **127**
12.1 Operatoren 127
12.1.1 Vorrang 127
12.1.2 Assoziativität 129
12.2 Differenzlisten 131
12.3 Differenz-Strukturen 136
12.4 Wörterbücher 137

Teil I

Logikprogrammierung

A logic program is a set of axioms, or rules, defining relationships between objects. A computation of a logic program is a deduction of consequences of the program.
A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning.

[Sterling/Shapiro 1994, S.9]

Prolog (**PRO**gramming in **LOG**ic) ist eine logikbasierte Programmiersprache, die sich in vielen Punkten deutlich von prozeduralen Programmiersprachen wie Pascal, Modula und C unterscheidet. Wie alle modernen Programmiersprachen kann auch Prolog prinzipiell zur Implementierung beliebiger berechenbarer Verfahren verwendet werden. Es gibt allerdings – wie für jede andere Programmiersprache – Aufgabenstellungen, für die sich Prolog besonders/weniger eignet. Vor allem bei der Lösung nicht-numerischer Probleme, die eine explizite Repräsentation der verfügbaren Informationen nahelegen oder fordern, kann sich eine deklarative Programmiersprache wie Prolog gegenüber anderen Sprachen auszeichnen.

Ein sinnvoller Umgang mit logikbasierten Programmiersprachen ist nur möglich, wenn man sich zunächst mit der Grundidee und den wichtigsten Konzepten der Logikprogrammierung vertraut gemacht hat. Aus diesem Grund werden in den ersten beiden Kapiteln Grundbegriffe der Logikprogrammierung eingeführt und zwei abstrakte Interpreter für Logikprogramme entwickelt, die ein weitgehend sprach- und maschinenunabhängiges Verständnis der Ausführung von Logikprogrammen vermitteln sollen.

Kapitel 1

Grundbegriffe

Dieses Kapitel skizziert kurz, wie sich Logikprogramme von prozeduralen Programmen unterscheiden und diskutiert den Zusammenhang zwischen Logikprogrammierung und dem Programmieren in Prolog. Anschließend werden die für die Logikprogrammierung zentralen Grundbegriffe und vier Deduktionsregeln eingeführt. Im letzten Teil des Kapitels wird ein einfacher abstrakter Interpreter für Logikprogramme entwickelt.

1.1 Prolog und Logikprogrammierung

Was versteht man unter Logikprogrammierung? Ähnlich wie in der funktionalen Programmierung wird in der Logikprogrammierung das von Neumann Maschinenmodell, das den meisten heutigen Rechnern und Programmiersprachen zugrundeliegt, durch ein anderes, abstraktes Modell ersetzt, dessen theoretische Grundlagen im Fall der Logikprogrammierung eine Teilmenge der Prädikatenlogik der 1. Stufe (*Hornklausel-Logik*) bildet. Für den Programmbegriff hat dieses Modell zwei Konsequenzen:

1. Bei der Entwicklung eines Logikprogramms beschränkt man sich darauf, das für die Lösung einer Aufgabenstellung relevante Wissen zu beschreiben, indem man alle Objekte und die zwischen ihnen bestehenden Relationen in Form von logischen Axiomen notiert.

2. Bei der Ausführung eines Logikprogramms wird versucht, die durch eine Frage formulierte Zielaussage aus dem Programm abzuleiten.

Anders formuliert: ein Programm besteht aus einer Menge von Axiomen, und die Ausführung des Programms kann als der Versuch betrachtet werden, einen konstruktiven Beweis der Zielaussage aus dem Programm zu finden. Der entscheidende Unterschied zu prozeduralen Sprachen liegt also darin, daß nur das aufgabenspezifische Wissen kodiert werden muß. Nicht beschrieben werden muß dagegen, wie dieses Wissen zur Lösung des Problems einzusetzen ist; denn der Beweismechanismus ist Bestandteil der Programmiersprache.

Ein Logikprogramm kann als eine spezifische Form der *Wissensrepräsentation* betrachtet werden: Das Logikprogramm kodiert das im Bezug auf eine Frage–/Problemstellung verfügbare Anfangswissen bzw. *explizite Wissen* und die aus ihm ableitbare Folgerungen stellen das aus dem Anfangswissen erschließbare Wissen bzw. *implizite Wissen* dar.

Der enge Zusammenhang, der zwischen Logikprogrammierung und PROLOG besteht, sollte allerdings nicht dazu führen, beides miteinander zu identifizieren:

- Logikprogrammierung basiert auf zwei abstrakten und maschinenunabhängigen Begriffen: dem der Wahrheit und dem der logischen Deduktion. Die Frage, ob eine Aussage aus einer Menge von Axiomen folgt oder nicht, kann unabhängig von verschiedenen zur Generierung eines Beweises möglichen Verfahren betrachtet werden.

- Eine logikbasierte Programmiersprache wie PROLOG dagegen realisiert ein spezifisches Beweisverfahren. Dieses Beweisverfahren kann bestimmte Anforderungen an die Präsentation der Axiome stellen (z.B. Reihenfolge der Axiome), die für die Logikprogrammierung selbst irrelevant sind. PROLOG ist insofern eine mehr oder weniger gute Realisierung der Konzepte, die der Logikprogrammierung zugrunde liegen.

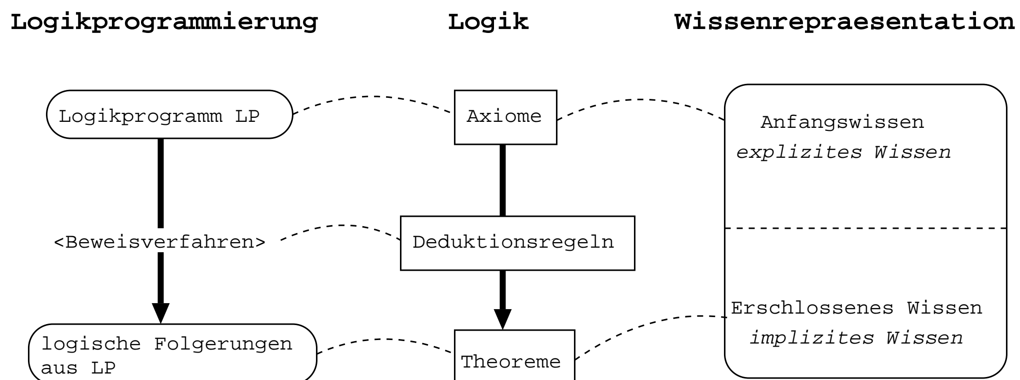


Abbildung (1-1)

PROLOG-Geschichte. PROLOG ist verglichen mit Sprachen wie FORTRAN, COBOL und LISP eine relativ neue Programmiersprache. Ihre Entwicklung ist eng mit der Arbeit von B.Kowalski und der Gruppe um A.Colmerauer zu Anfang der 70er Jahre verbunden:

- Kowalski entwickelte eine prozedurale Interpretation der Hornklausel-Logik. Er zeigte, daß ein Axiom der Form A *wenn* B_1 *und* B_2 ... *und* B_n als eine Prozedur einer rekursiven Programmiersprache betrachtet werden kann; denn neben der für die Logikprogrammierung entscheidenen *deklarativen Lesart* für dieses Axiom (A ist wahr wenn alle B's wahr sind), ist auch folgende *prozedurale Lesart* möglich: *Um A zu beweisen, beweise zunächst* B_1*, dann* B_2*, ...*

- Die Gruppe um Colmerauer entwickelte in Marseille einen speziellen in FORTRAN geschriebenen Theorembeweiser für die Hornklausel-Logik, der sich diese prozedurale Interpretation zunutze machte. Das verwendete Beweisverfahren wird so zu einem Interpreter für eine Programmiersprache, und der Unifikationsalgorithmus, der den Kern dieses Beweisverfahrens bildet, übernimmt die wichtigsten Operationen auf den Daten (Wertzuweisung, Parameterübergabe, etc.).

1.2 Terme

Logikprogramme bestehen aus *Termen* und *Klauseln*. Es gibt zwei Typen von Termen: einfache und komplexe Terme.

Definition (1–1) *Term*

1. Variablen und Konstanten sind einfache Terme.
2. Ein komplexer Term besteht aus einem Funktor und einem oder mehreren Argumenten: *funktor(argument*$_1$*,..., argument*$_n$*)*.

Es werden numerische und symbolische Konstanten unterschieden. Symbolische Konstanten werden als *Atome* und komplexe Terme als *Strukturen* bezeichnet. Als Funktor einer Struktur kann ein beliebiges Atom, als Argumente können beliebige Terme verwendet werden; d.h. Strukturen bilden eine rekursive Datenstruktur. Strukturen werden durch den Namen ihres Funktors und ihre Stelligkeit identifiziert.

Beispiel (1–1)

Variablen		X, _Y11, Hallo_Wer_Da, _
Konstanten	*numerisch*	-11, 99, 3.14
	symbolisch	prolog, x, hallo_Wer_da
Strukturen		alter(X, Y), datum(30, oktober, 1990), einkommen(batman, X), baum(baum(nil, 3, nil), 5, R)

Obwohl es in den ersten beiden Kapiteln vor allem um Grundbegriffe der Logikprogrammierung geht, verwenden wir auch hier schon die für Prolog geltenden syntaktischen Regeln. Beispiel (1–1) läßt erkennen, daß Atome immer mit einem Kleinbuchstaben beginnen, Variablen dagegen mit einem Großbuchstaben oder dem Unterstrich '_'.

1.3 Klauseln

Es werden drei Typen von Klauseln unterschieden: *Fakten*, *Regeln* und *Fragen*.

1.3.1 Fakten

Fakten bilden den einfachsten Typ von Klauseln. Ein Fakt beschreibt eine Beziehungen zwischen Objekten bzw. konstatiert das Bestehen eines bestimmten Sachverhalts. Syntaktisch betrachtet besteht ein Fakt aus einem Term (einem Atom oder einer Struktur) gefolgt von einem Punkt. Das Fakt *vater(peter, maria)*. z.B. legt fest, daß Peter der Vater von Maria ist; d.h. zwischen den mit den Namen „Peter“ und „Maria“ bezeichneten Objekten besteht eine bestimmte Relation: die *Ist_ Vater_ von-Relation*.

Die durch eine Struktur festgelegte Relation wird auch als *Prädikat* bezeichnet. Durch Prädikate lassen sich auch Operationen abbilden, die normalerweise funktional beschrieben werden. So wird die Addition z.B. häufig als zweistellige Funktion charakterisiert. Ein Logikprogramm kann sie durch Verwendung eines dreistelligen Prädikats beschreiben:[1]

Beispiel (1–2)

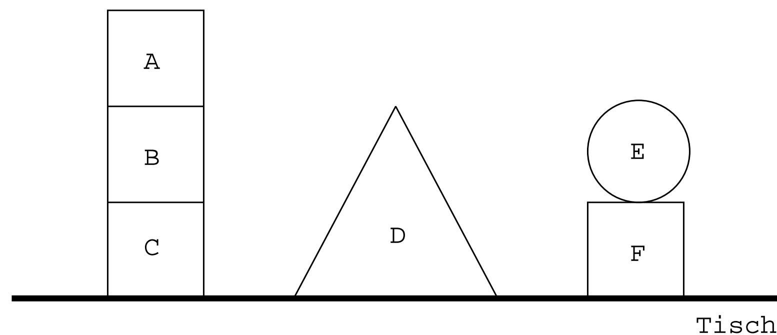
Additionstafel

plus(0, 0, 0).	plus(0, 1, 1).	plus(0, 2, 2).	plus(0, 3, 3).
plus(1, 0, 1).	plus(1, 1, 2).	plus(1, 2, 3).	plus(1, 3, 4).

Ein Logikprogramm besteht im einfachsten Fall, wie dieses Beispiel zeigt, aus einer endlichen Zahl von Fakten. Die Reihenfolge, in der die Fakten genannt werden, ist in einem Logikprogramm irrelevant. Die meisten nicht-trivialen Logikprogramme bestehen allerdings nicht allein aus Fakten, sondern aus Fakten und Regeln.

Beispiel (1–3)

Spielwelt



[1] Jede n-stellige Funktion kann auf ein n+1-stelliges Prädikat abgebildet werden.

Diese Welt läßt sich durch eine Reihe Fakten beschreiben, die die fundamentalen Eigenschaften der Objekte und die zwischen ihnen bestehenden Relationen festhalten:

Form	*Farbe*	*Position*
block(a).	blau(a).	liegt_auf(a, b).
block(b).	rot(b).	liegt_auf(b, c).
block(c).	rot(c).	liegt_auf(c, tisch).
pyramide(d).	gelb(d).	liegt_auf(d, tisch).
kugel(e).	blau(e).	liegt_auf(e, f).
block(f).	gruen(f).	liegt_auf(f, tisch).

1.3.2 Fragen

Fakten und Regeln (s.u.) dienen dazu, das über einen gegebenen Gegenstandsbereich verfügbare Wissen zu kodieren. Durch Fragen lassen sich aus einem Logikprogramm Informationen gewinnen. Der Kontext erlaubt es, eine Klausel eindeutig als Fakt oder Frage zu identifizieren. Aus Gründen der besseren Lesbarkeit werden aber meistens Fragen anders als Fakten mit einem Fragezeichen abgeschlossen.

Beispiel (1–4)
Die Frage *liegt_auf(a, b)?* wird mit *ja* bzw. *true* beantwortet, gdw. aus dem zugrundeliegenden Logikprogramm folgt, daß das Objekt a auf dem Objekt b liegt.

Einfache Fragen

Wir betrachten zunächst den einfachsten Fall: L sei ein Logikprogramm, das ausschließlich aus Fakten besteht. Diese Fakten konstatieren das Bestehen bestimmter Sachverhalte. Um eine Frage der Form P? zu beantworten, wird geprüft, ob der durch P ausgedrückte Sachverhalt, auch *Ziel* genannt, zu den von L konstatierten Sachverhalten gehört. Das ist genau dann der Fall, wenn das durch P formulierte Ziel aus L *logisch folgt*; d.h., wenn P aus L durch die Anwendung von *Deduktionsregeln* ableitbar ist.

Zwei Typen von Fragen werden unterschieden: *Einfache Fragen*, die aus einem Ziel bestehen und *komplexe Fragen*, die aus einer Konjunktion von Zielen gebildet werden. Wenn P eine einfache Frage ist, die keine Variable enthält, dann benötigt man nur eine einfache Deduktionsregel: die *Identitätsregel*.

Deduktionsregel (1) *Identität*
Eine einfache Frage folgt logisch aus einem identischen Fakt ($P \models P$).

Um zu überprüfen, ob eine einfache Frage P aus einem Programm G folgt, muß also nach einem Fakt in G gesucht werden, das mit P identisch ist.

Variablen

In einem Logikprogramm haben Variablen eine andere Funktion als in konventionellen Programmiersprachen: Sie bezeichnen keinen Speicherabschnitt, sondern ein nicht näher spezifiziertes einzelnes Objekt. Eine Frage wie *liegt_ auf(X, tisch)?* ist zu lesen als "Gibt es ein Objekt X, für das gilt: X liegt auf dem Tisch?". Terme, die keine Variablen enthalten, werden *geschlossene Terme* genannt; alle anderen Terme werden als *offene Terme* bezeichnet.

Definition (1–2) *Substitution*

Eine Substitution α ist eine endliche Menge von Paaren $<X_i, t_i>$, die aus einer Variable X_i und einem Term t_i bestehen. Für alle Paare $<X_i, t_i>$, $<X_j, t_j>$ in α gilt:

(i). $X_i \neq X_j$ für $i \neq j$ und

(ii). X_i kommt nicht in t_i vor.

Wenn wir noch einmal die letzte Frage betrachten, dann ist relativ zu der Datenbasis aus Beispiel (1–3) $\alpha = \{<X, c>\}$ eine mögliche Substitution. Das Ergebnis der Anwendung einer Substitution $\alpha = \{<X_1, t_1>, \ldots, <X_n, t_n>\}$ auf einen Term A (notiert als: $A\alpha$) ist der Term, den man erhält, indem man jedes Vorkommen von X_i in A durch t_i ersetzt.

Definition (1–3) *Instanz*

A ist eine Instanz von B gdw. es eine Substitution α gibt, so daß $A = B\alpha$.

Wenn z.B. A = liegt_auf(X, tisch) und $\alpha = \{<X, c>\}$, dann gilt: $A\alpha$ = liegt_auf(c, tisch). Außerdem gilt: *liegt_ auf(c, tisch)* ist eine Instanz von *liegt_ auf(X, tisch)*, und *liegt_ auf(X, tisch)* ist wiederum eine Instanz von *liegt_ auf(X, Y)*.

Existentielle Fragen

Variablen in Fragen sind *existentiell* gebunden; d.h. es wird danach gefragt, ob es Objekte gibt, die in der beschriebenen Relation zueinander stehen. Eine Frage $p(T_1, \ldots, T_n)?$, mit $n \geq 1$, die die Variablen $X_1, \ldots, X_k$ enthält, ist zu lesen als *Gibt es Objekte $X_1, \ldots, X_k$, so daß gilt: $p(T_1, \ldots, T_n)?$*. Zur Beantwortung von existentiellen Fragen benötigen wir eine weitere Deduktionsregel: die *Generalisierungsregel*.

Deduktionsregel (2) *Generalisierung*

Eine existentielle Frage P folgt logisch aus jeder ihrer Instanzen P α, für beliebige Substitutionen α.

Eine Antwort auf eine existentielle Frage zu finden erfordert die Suche nach einem Fakt im Programm, das eine Instanz der Frage ist. Gibt es eine derartige Instanz, wird die Frage mit *true/yes*, sonst mit *true/no* beantwortet. Bei positiv zu beantwortenden Fragen wird außerdem die Substitution ausgegeben, die die Frage in die gefundene Instanz überführt. Bei existentiellen Fragen mehr als eine positive Antwort geben:

Beispiel (1–5)

liegt_auf(X, tisch)?	plus(X, Y, 4)?
{<X, c>}	{<X, 0>, <Y, 4>}
{<X, d>}	{<X, 1>, <Y, 3>}
{<X, f>}	......

Universelle Fakten

Variablen in Fakten sind implizit *universell* gebunden; d.h. es wird festgestellt, daß alle Objekte die durch dieses Fakt beschriebene Relation erfüllen; statt zu schreiben:

haßt(eva, logikprogrammierung)
haßt(eduard, logikprogrammierung)
haßt(susi, logikprogrammierung)
⋮ ⋮ ⋮

kann man genausogut *haßt(X, logikprogrammierung)* schreiben. Ein Fakt p(T_1, ..., T_n), das die Variablen X_1, ..., X_k enthält, ist also zu lesen als: *Für alle* X_1, ..., X_k *ist* $p(T_1, \ldots, T_n)$ *wahr*. Aus einem universellen Fakt folgen alle seine Instanzen.

Deduktionsregel (3) *Instantiierung*
Aus einem universellen Fakt P folgen logisch alle Instanzen $P\alpha$, für beliebige Substitutionen α.

Eine Frage, die keine Variable enthält, mit Hilfe eines universellen Fakts zu beantworten ist einfach. Sie folgt aus jedem universellen Fakt, dessen Instanz sie ist:

plus(0, 1, 1)?	*plus(0, 2, 2)*	...
plus(0, X, X).	*plus(0, X, X).*	...
yes	*yes*	...

Enthält die Frage dagegen auch Variablen, ist nach einer gemeinsamen Instanz von Frage und Fakt zu suchen:

Definition (1–4) *Gemeinsame Instanz*

C ist eine gemeinsame Instanz von A und B, wenn C eine Instanz von A und C eine Instanz von B ist; d.h. es gibt Substitutionen α_1, α_2, so daß C = Aα_1 syntaktisch mit Bα_2 übereinstimmt.

Wenn z.B. A = *plus(0, 3, Y)*, B = *plus(0, X, X)*, α_1 = *{<X, 3>}* und α_2 = *{<Y, 3>}*, dann ist C= *plus(0, 3, 3)* eine gemeinsame Instanz von A und B.

Zur Beantwortung existentieller Fragen durch universelle Fakten sind zwei Ableitungsschritte erforderlich: Zum einen muß durch die Instantiierungsregel aus dem Fakt eine geeignete Instanz und mit der Generalisierungsregel die Frage aus der Instanz abgeleitet werden.

Komplexe Fragen

Komplexe Fragen bestehen im Gegensatz zu einfachen Fragen aus einer Konjunktion von Zielen; d.h. sie haben die Form $Q_1, \ldots, Q_n$*?*, mit $n \geq 1$. Enthält eine komplexe Frage keine Variablen, dann wird geprüft, ob alle Q_i aus dem Programm P folgen. Ein interessanter Fall tritt dann ein, wenn eine komplexe Frage eine oder mehrere *gemeinsame Variablen* enthält; Variablen, die in mehreren Zielen der Frage vorkommen. Der Geltungsbereich einer Variablen umfaßt in diesem Fall die gesamte Frage. Durch gemeinsame Variablen werden so komplexe Bedingungen formuliert, die die Objekte zu erfüllen haben, die in der spezifizierten Relation zueinander stehen.

Beispiel (1–6)

Um die Frage *liegt_ auf(X, tisch), rot(X)?* zu beantworten, muß nach einem Objekt gesucht werden, das auf dem Tisch liegt <u>und</u> das rot ist. Relativ zu der Datenbasis aus Beispiel (1–3) ist *{<X, c>}* die einzige mögliche Instanz.

Es gilt also: Eine komplexe Frage folgt logisch aus einem Programm P, gdw. alle Ziele der Frage aus P folgen, wobei gemeinsame Variablen mit demselben Wert instantiiert werden. Um eine Frage $A_1, \ldots, A_n$? relativ zu einem Programm P zu beantworten, ist nach einer Substitution α zu suchen, so daß $A_1\alpha, \ldots, A_n\alpha$ Instanzen von Fakten in P sind.

1.3.3 Regeln

Regeln haben die Form $A \leftarrow B_1, \ldots, B_n$ ($n \geq 0$); A wird als *Kopf* und $B_1, \ldots, B_n$ als *Körper* der Regel bezeichnet. Wie man sieht, bilden Fakten einen Spezialfall von Regeln: Es sind Regeln mit einem leerem Körper.

Beispiel (1–7)
Die folgenden drei Regeln definieren, die *Sohn–*, *Tochter–* und *Großvater–* Relation:

sohn(X, Y) $\leftarrow$	tochter(X, Y) $\leftarrow$	großvater(X, Y) $\leftarrow$
vater(Y, X),	vater(Y, X),	vater(X, Z),
männlich(X).	weiblich(X).	vater(Z, Y).

Regeln können als Mittel betrachtet werden, um komplexe Fragen auf einfache Fragen zu reduzieren: Enthält das Programm die Regel $A \leftarrow B_1, \ldots, B_n$, dann kann die komplexe Frage $B_1, \ldots, B_n?$ ersetzt werden durch die einfache Frage $A?$. Für Regeln gibt es zwei Lesarten:

- **Prozedurale Lesart**
  Um A zu beweisen, beweise zunächst B_1*, dann* B_2*, ..., dann* B_n*.*
  Eine Regel wird ähnlich wie eine Prozedur einer prozeduralen Programmiersprache interpretiert; d.h. sie legt die Reihenfolge fest, in der bestimmte Operationen auszuführen sind.
- **Deklarative Lesart**
  A ist wahr, wenn B_1 *und* B_2 *... und* B_n *wahr ist.*
  Als logische Axiome interpretiert, legen Regeln die Bedingungen fest, unter denen Objekte die durch den Kopf der Regel spezifizierte Relation erfüllen; d.h. eine neue/komplexe Relation wird auf bekannte/einfachere Relation zurückgeführt.

Um Logikprogramme, die neben Fakten auch Regeln enthalten, korrekt verarbeiten zu können, benötigen wir eine weitere Deduktionsregel, die die Indentitäts- und Instantiierungsregel als Spezialfälle enthält.

Deduktionsregel (4) *Modus Ponens*
Aus einer Regel $R = A \leftarrow B_1, B_2, \ldots, B_n$ und den Fakten $B'_1, B'_2, \ldots, B'_n$ kann A' abgeleitet werden, wenn $A' \leftarrow B'_1, B'_2, \ldots, B'_n$ eine Instanz von R ist.

Jetzt können wir die Begriffe *Logikprogramm* und *logische Folge* präzise definieren:

Definition (1–5) *Logikprogramm*
Ein Logikprogramm ist eine endliche Menge von Regeln.

Definition (1–6) *Logische Folge*
Ein existentiell quantifiziertes Ziel G ist eine logische Folge eines Programms P, wenn es eine Klausel in P mit einer geschlossenen Instanz $A \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$ gibt, so daß $B_1, \ldots, B_n$ logische Folgen von P sind und A eine Instanz von G ist.

Beispiel (1–8)

Familienbande

vater(earl, robby).
vater(earl, charline).
mutter(franny, charline).
mutter(franny, baby_sinclair).

männlich(earl).
männlich(robby).
weiblich(franny).
weiblich(charline).

Ausgehend von der einfachen Frage *sohn(robby, earl)?* erhalten wir auf Grundlage dieser Datenbasis und der Sohn–Regel aus dem letzten Beispiel durch Anwendung der Substitution {<X, robby>, <Y, earl>}:

sohn(robby, earl) $\leftarrow$ *vater(earl, robby), männlich(robby).*

Beide Ziele, die den Körper dieser Instanz der Regeln bilden, folgen aus dem Programm. Damit ist das ursprüngliche Ziel bewiesen.

Wie man leicht sieht, ist die Regel, die das *Sohn*–Prädikat festlegt, korrekt aber unvollständig: Es kann z.B. nicht festgestellt werden, daß Baby Sinclair der Sohn von Franny ist, denn Franny ist nicht der Vater von Baby Sinclair (erstaunlich, aber wahr). Man könnte aus diesem Grund eine weitere Regel hinzufügen:

sohn(X, Y) $\leftarrow$ mutter(Y, X), männlich(X).

Aber auch die Definitionen anderer Verwandtschaftsrelationen müßten auf ähnliche Weise ergänzt werden. Sehr viel ökonomischer und die Transparenz des Programms erhöhend ist es , ein Prädikat *elternteil* zu definieren und dieses Prädikat bei der Definition der anderen Prädikate zu verwenden:

elternteil(X, Y) $\leftarrow$ vater(X, Y).
elternteil(X, Y) $\leftarrow$ mutter(X, Y).
sohn(X, Y) $\leftarrow$ elternteil(Y, X), männlich(X).
großelternteil(X, Y) $\leftarrow$ elternteil(X, Z), elternteil(Z, Y).

Alle Regeln, die ein Prädikat definieren, werden als eine *Prozedur* bezeichnet; denn unter prozeduralem Gesichtspunkt entsprechen diese Regeln in etwa den Prozeduren konventioneller Programmiersprachen.

1.4 Ein einfacher abtrakter Interpreter für Logikprogramme

Der abstrakte Interpreter für Logikprogramme, den wir im folgenden betrachten werden, nimmt ein Logikprogramm P und eine Frage F, die aus geschlossenen Termen besteht, als Eingabe. Der Interpreter versucht F relativ zu P zu beweisen. Allerdings terminiert ein Beweis nicht in jedem Fall: Wenn F nicht beweisbar ist, kann der Interpreter in eine Endlosschleife geraten.

PROZEDUR LPI-(1)

∗ *Ein abstrakter Interpreter für Logikprogramme* ∗

EINGABE: Ein geschlossene Frage $F = A_1, \ldots, A_n$ $(n > 0)$ und ein Programm P.
AUSGABE: ja/nein.

METHODE

RESOLVENT $\Leftarrow \{F\}$.
ERGEBNIS $\Leftarrow$ *true*.

WHILE (RESOLVENT $\neq \emptyset$) $\wedge$ (ERGEBNIS $=$ *true*) DO
  Wähle ein Ziel A_i.
  Wenn <Es gibt eine geschlossene Instanz einer Klausel $A \leftarrow B_1, \ldots, B_k$, mit $A = A_i$.>
  Dann <RESOLVENT $= \{A_1, \ldots, A_{i-1}, B_1, \ldots B_k, A_{i+1}, \ldots A_n\}$.>
  Sonst <ERGEBNIS $\Leftarrow$ *false*>

Wenn <RESOLVENT $= \emptyset$>
Dann <Ausgabe(ja)>
Sonst <Ausgabe(nein)>

Beispiel (1–9)

Ein Trace für den Interpreter

Programm: Datenbasis aus Beispiel (1–8) und Regeln aus Beispiel (1–7)
Eingabe: sohn(robby, earl)?

Resolvent = {sohn(robby, earl)} *(Initialisierung)*
  Gewähltes Ziel : sohn(robby, earl).
  Gewählte Klausel : sohn(robby, earl) ← vater(earl, robby), männlich(robby).
  Resolvent = {vater(earl, robby), männlich(robby).}

  Gewähltes Ziel : vater(earl, robby).
  Gewählte Klausel : vater(earl, robby).
  Resolvent = {männlich(robby)}

  Gewähltes Ziel : männlich(robby).

Gewählte Klausel : männlich(robby).
Resolvent = ∅

Ausgabe: ja

Der Trace enthält implizit den Beweis des Ziels aus dem Programm. Eine andere Möglichkeit Beweise zu repräsentieren, bieten *Beweisbäume*, die auf dem Konzept der (geschlossenen) Reduktion basieren:

Definition (1–7) *Geschlossene Reduktion*

Eine geschlossene Reduktion eines Ziels G durch ein Programm P ist die Ersetzung von G durch den Körper der geschlossene Instanz einer Regel aus P, deren Kopf mit G identisch ist.

Die Ausführung eines Logikprogramms basiert auf Reduktionsschritten. Ein Reduktionsschritt entspricht der Anwendung des Modus Ponens: Das durch seine Anwendung ersetzte Ziel wird *reduziert*; die neuen Ziele werden durch den Reduktionsschritt *abgeleitet*. In unserem Beispiel gab es drei Reduktionsschritte:

1. *sohn(robby, earl)* wird reduziert. Die neuen Ziele *vater(earl, robby)* und *männlich(robby)* werden abgeleitet.

2. *vater(earl, robby)* wird reduziert. Es werden keine neue Ziele abgeleitet.

3. *männlich(robby)* wird reduziert. Es werden keine neue Ziele abgeleitet.

Ein Beweisbaum besteht aus Knoten und Kanten, die bei der Ausführung des Programms reduzierte Ziele repräsentieren. Von jedem Knoten, der ein Ziel repräsentiert, führt eine Kante zu einem Knoten, der ein durch die Reduktion dieses Zieles abgeleitetes Ziel repräsentiert. Der Beweisbaum für komplexe Fragen besteht aus den Beweisbäumen für die in ihnen enthaltenen Ziele. Unserem Beispiel korrespondiert folgender Beweisbaum:

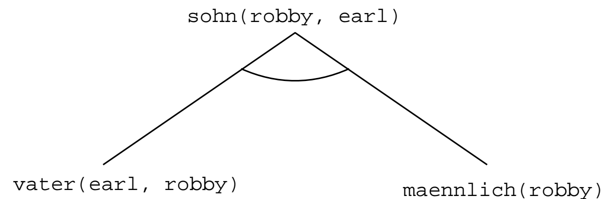


Abbildung (1-2)

Der oben angebene Interpreter für Logikprogramme ist in zweifacher Hinsicht indeterministisch: zu wählen ist zum einen das Ziel, das reduziert werden soll, und zum anderen die Klausel, mit der ein best. Ziel reduziert wird. Beide Entscheidungen unterscheiden sich hinsichtlich ihrer Konsequenzen erheblich:

- Die Wahl des zu reduzierenden Ziels ist irrelevant, da unabhängig von der Reihenfolge, in der Ziele bearbeitet werden, ein Beweis gefunden wird.

- Allerdings ist die Wahl der bei einer Reduktion zu verwendenden Klausel relevant, da es von der Wahl der Klausel abhängen kann, ob ein Beweis für ein Ziel gefunden wird oder nicht und ob der Beweis terminiert.

1.5 Die Bedeutung eines Logikprogramms

Definition (1–8) *Bedeutung eines Logikprogramms*

Die Bedeutung M(P) eines Logikprogramms P ist die Menge der aus P ableitbaren geschlossenen Ziele.

Die *intendierte Bedeutung* eines Programms P ist die Menge aller Ziele, die aus P ableitbar sein sollen. Ein Programm P ist *korrekt* relativ zu seiner intendierten Bedeutung M_i gdw. $M(P) \subseteq M_i$; es ist *vollständig* gdw. $M_i \subseteq M(P)$. Ein Programm ist *vollständig und korrekt* gdw. $M_i. = M(P)$.

Kapitel 2

Unifikation

Der in 1.4 entwickelte abstrakte Interpreter für Logikprogramme arbeitet nur dann korrekt, wenn ausschließlich Ziele betrachtet werden, die keine Variablen enthalten. Wir werden den Interpreter im folgenden so erweitern, daß diese Restriktion nicht mehr zu beachten ist. Wichtigste Voraussetzung für die Erweiterung ist ein *Unifikationsalgorithmus*, der die *gemeinsame Instanz* zweier Terme berechnet. Abschließend werden wir die Beweisstrategie des Interpreters mit der von Prolog verwendeten Beweisstrategie vergleichen.

2.1 Der Unifikationalgorithmus

Wir beginnen mit einer Reihe von Definitionen:

1. Ein Term t ist eine *gemeinsame Instanz* von zwei Termen t_1 und t_2, gdw. es Substitutionen α_1, α_2 gibt, so daß $t = t_1\alpha_1 = t_2\alpha_2$ (vgl. Definition (1-4)).

2. Ein Term t_1 ist *allgemeiner* als ein Term t_2 gdw. t_2 eine Instanz von t_1 ist, aber nicht umgekehrt.

3. Ein Term t_1 ist eine *Variante* eines Term t_2, gdw. t_1 eine Instanz von t_2 und t_2 eine Instanz von t_1 ist.

   Beispiel (2–1)
   Der Term *member(X, baum(Links, X, Rechts))* ist eine Variante des Terms *member(Y, baum(Links, Y, Z))* (und umgekehrt), da beide Terme bis auf die Variablennamen miteinander übereinstimmen.

Definition (2–1) *Unifikator*

Der *Unifikator* zweier Terme t_1, t_2 ist eine Substitution α, für die gilt: $t_1\alpha = t_2\alpha$.

Wenn es für zwei Terme einen Unifikator gibt, sagt man auch: beide Terme *unifizieren*. Wie man sich leicht klarmachen kann, besteht ein enger Zusammenhang zwischen dem Begriff der gemeinsamen Instanz und dem des Unifikators: Der Unifikator zweier Terme ist eine Substitution, durch die sie in eine gemeinsame Instanz überführt werden können.

Definition (2–2) *Allgemeinster Unifikator*

Der *allgemeinste Unifikator* (kurz: MGU für *most general unifier*) zweier Terme ist der Unifikator beider Terme, für den gilt, daß die durch ihn festgelegte gemeinsame Instanz die *allgemeinste gemeinsame Instanz* beider Terme ist.

Für zwei Terme kann es mehrere Unifikatoren, aber von Variablenumbenennungen abgesehen nur einen MGU geben.

Beispiel (2–2)

Die beiden Terme t_1 = *datum(Tag, dezember, Jahr)* und t_2 = *datum(14, Monat, Jahr)* unifizieren. Ein Unifikator für diese Terme ist die Substitution {<Tag, 14>, <Monat, dezember>, <Jahr, 1990>}.

Der MGU von t_1 und t_2 ist die Substitution {<Tag, 14>, <Monat, dezember>}. Durch die Anwendung dieses Unifikators erhält man als gemeinsame Instanz beider Terme den Term *datum(14, dezember, Jahr)*.

PROZEDUR MGU(t_1, t_2)

∗ *Berechnung des allgemeinsten Unifikators der Terme t_1 und t_2* ∗

EINGABE: Zwei Terme t_1 und t_2.
AUSGABE: Der MGU von t_1 und t_2; sonst *false*.

METHODE:

UNIFIKATOR ⇐ ∅.

STACK ⇐ [t_1 = t_2].

FAIL ⇐ *false*.

WHILE (STACK ≠ [ ]) ∧ (FAIL ≠ *true*) DO

Entferne die oberste Gleichung X = Y aus STACK.

Falls

(1) X ist eine Variable, die nicht in Y vorkommt:
Substituiere in STACK und in UNIFIKATOR X durch Y.
UNIFIKATOR ⇐ UNIFIKATOR ∪ {<X, Y>}.

(2) Y ist eine Variable, die nicht in X vorkommt:
Substituiere in STACK und UNIFIKATOR Y durch X.
UNIFIKATOR ⇐ UNIFIKATOR ∪ {<Y, X>}.

(3) X und Y sind identische Konstanten oder Variablen.

(4) X = f(X_1, ..., X_n) und Y = f(Y_1, ..., Y_n), mit $n \geq 1$:
FOR I = 1 TO n DO
PUSH(X_i = Y_i, STACK).

(5) Sonst: FAIL ⇐ *true*.

Wenn <FAIL = *true*>
Dann <Ausgabe(*false*)>
Sonst <Ausgabe(UNIFIKATOR)>

Wenn es sich bei einem der beiden zu unifizierenden Terme um eine Variable handelt, wird überprüft, ob diese Variable in dem anderen Term vorkommt (*occur-check*). Dieser Test ist erforderlich, um die Terminierung des Algorithmus sicherzustellen[1]. Die meisten PROLOG-Implementierung verzichten aus Effizienzgründen auf diesen Test.

Beispiel (2–3)

Wenn t_1 = *objekt(a, Typ, rot, Position)* und
t_2 = *objekt(Name, block, Farbe, liegt_auf(Name, X))*, dann gilt:

1. STACK = <objekt(a, Typ, rot, Position) = objekt(Name, block, Farbe, liegt_auf(Name, X)>

[1] Welchen Wert liefert der Algorithmus ohne diesen Test für t_1 = X und t_2 = s(X)?

(Initialisierung)

2. STACK = <a = Name, Typ = block, rot = Farbe,
   Position = liegt_auf(Name, X)>
   (Fall 4)
3. STACK = <Typ = block, rot = Farbe, Position = liegt_auf(a, X)>
   UNIFIKATOR = {<Name, a >}
   (Fall 2)
4. STACK = <rot = Farbe, Position = liegt_auf(a, X)>
   UNIFIKATOR = {<Name, a >, <Typ, block>}
   (Fall 1)
5. STACK = <Position = liegt_auf(a, X)>
   UNIFIKATOR = {<Name, a >, <Typ, block>, <Farbe, rot>}
   (Fall 2)
6. STACK = <>
   UNIFIKATOR = {<Name, a >, <Typ, block>, <Farbe, rot>,
   <Position, liegt_auf(a, X)>}
   (Fall 1)

Ausgabe: {<Name, a >, <Typ, block>, <Farbe, rot>, <Position, liegt_auf(a, X)>}

Durch Anwendung des MGUs erhalten wir als gemeinsame Instanz beider Terme den Term *objekt(a, block, rot, liegt_auf(a, X))*.

2.2 Ein Interpreter für Logikprogramme

Die Ausführung eines Logikprogramms kann, wie wir wissen, als der Versuch verstanden werden, ein möglicherweise komplexes Ziel durch eine Folge von Reduktionsschritten zu beweisen. In jedem Reduktionsschritt wird aus dem komplexen Ziel ein einfaches Ziel ausgewählt, das durch Anwendung einer Regel reduziert wird. Daher kann der Beweis eines Ziels Z_0 relativ zu einem Programm P durch eine möglicherweise nicht-endliche Folge von Tripeln $<Q_i, G_i, K_i>$ repräsentiert werden, wobei

- Q_i das aktuelle (einfache bzw. komplexe) Ziel ist;
- G_i ein Teilziel von Q_i ist und
- K_i eine Klausel $A \leftarrow B_1, \ldots, B_n$ aus P ist, deren Variablen so umbenannt wurden, daß K_i und Q_i keine gemeinsamen Variablen enthalten.

Q_{i+1} ($i > 0$) ist das Ergebnis der Ersetzung von G_i durch den Körper von K_i in Q_i und der Anwendung des MGU von G_i und A. Der Beweis terminiert entweder wenn Q_i ein einfaches Ziel ist ($G_i = Q_i$) und es ein Fakt in P gibt, das mit Q_i unifiziertbar ist oder wenn es in P keine Regel gibt, deren Kopf mit dem aktuellen Ziel G_i unifizierbar ist. Im ersten Fall ist Q_{i+1} = *true*; im anderen gilt: Q_{i+1} = *false*.

Der *Trace* eines Beweises ist eine Folge von Paaren $<G_i, I_i^{'}>$, mit $I_i^{'}$ als der Teilmenge des MGU I_i, der beim i-ten Reduktionsschritt berechnet wurde, die nur Variablen aus G_i enthält.

PROZEDUR LPI-(2)

Ein abstrakter Interpreter für Logikprogramme

EINGABE: Ein Logikprogramm P und ein Ziel G.
AUSGABE: Eine aus P abgeleitete Instanz G' von G; sonst *false*.

METHODE:

RESOLVENT $\Leftarrow$ G.

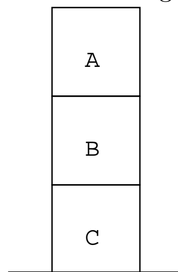
ERGEBNIS $\Leftarrow$ *true*.

WHILE (RESOLVENT $\neq \emptyset$) $\wedge$ (ERGEBNIS $\neq$ *false*) DO
  Wähle ein Ziel A $\in$ RESOLVENT.
  Wenn <Es gibt eine Klausel A' $\leftarrow B_1, ..., B_n$ ($n \geq 0$) in P, mit
    MGU(A, A') $\neq$ *false*.>
  Dann <Entferne A aus RESOLVENT.
    RESOLVENT $\Leftarrow$ RESOLVENT $\cup \{B_1, \ldots, B_n\}$.
    Wende MGU(A, A') auf RESOLVENT und G an.>
  Sonst <ERGEBNIS $\Leftarrow$ *false*.>

Wenn <RESOLVENT = $\emptyset$>
Dann <Ausgabe(G)>
Sonst <Ausgabe(*false*)>

Es gibt Beweise, in denen in jedem Schritt nur genau eine Klausel zur Reduktion des gewählten Ziels verwendet werden kann. Ein Beweis dieses Typs wird *deterministisch* genannt; gibt es in einem Beweisschritt mehrere anwendbare Klauseln, wird der Beweis *nicht–deterministisch* genannt.

Beispiel (2–4)
Gegeben sei das folgende Logikprogramm:




```
liegt_auf(a, b).
liegt_auf(b, c).
liegt_auf(c, tisch).


liegt_unter(X, Y) <-        (R 1)
         liegt_auf(Y, X).
liegt_unter(X, Y) <-        (R 2)
   liegt_auf(Z, X),
   liegt_unter(Z, Y).
```

Ein Beweis für das Ziel *liegt_unter(c, a)* ist nicht–deterministisch, denn es können beide Klauseln zur Reduktion des Ziels verwendet werden:

1. Resolvent = {liegt_unter(c, a)}
   Klausel: liegt_unter(X, Y) ← liegt_auf(Y, X). [R 1]
   MGU = {<X, c>, <Y, a>>}
2. Resolvent = {liegt_auf(a, c)}
   Klausel: Es gibt keine Klausel des Programms, die mit einem Ziel des Resolventen unifiziert. Der Beweis scheitert.

1'. Resolvent = {liegt_unter(c, a)}
   Klausel: liegt_unter(X, Y) ← liegt_auf(Z, X), liegt_unter(Z, Y). [R 2]
   MGU = {<X, c>, <Y, a>}

2'. Resolvent = {liegt_auf(Z, c), liegt_unter(Z, a)}
   Klausel: liegt_auf(b, c). [Fakt]
   MGU = {<Z, b>}

3'. Resolvent = {liegt_unter(b, a)}
   Klausel: liegt_unter(X1, Y1) ← liegt_auf(Y1, X1). [R 1]
   MGU = {<X1, b>, <Y1, a>}

4'. Resolvent = {liegt_auf(a, b)}
   Klausel: liegt_auf(a, b). [Fakt]

5'. Resolvent = ∅

Der Beweis aus dem letzten Beispiel kann durch folgenden Trace repräsentiert werden:

Ziel	
liegt_unter(c, a)	∅
liegt_auf(Z, c)	{<Z, b>}
liegt_unter(b, a)	∅
liegt_auf(a, b)	∅
true	
AUSGABE: *yes*	

Die Aufgabe einen Beweis für ein Ziel zu konstruieren, kann als ein Suchproblem betrachtet werden: In jedem Beweisschritt gilt es eine Regel zu finden, mit der sich das ausgewählte (Teil-)Ziel reduzieren läßt. Oft gibt es mehrere anwendbare Regeln. Abhängig davon, welche der Regeln verwendet wird, kann der Beweis gelingen oder zunächst scheitern. Es gibt verschiedene Strategien, die der Interpreter bei der Suche nach anwendbaren Regeln verwenden kann:

- Bei der *depth-first* Suche wird der gerade gewählte Pfad bis zum Endknoten verfolgt, bevor andere Pfade untersucht werden.
- Bei der *breadth-first* Suche dagegen werden alle Pfade 'parallel' verfolgt.

Die *breadth-first* Suche bildet anders als die *depth-first* Suche ein vollständiges Beweisverfahren für Logikprogramme; denn nicht-endliche Pfade können bei der Verwendung der *depth-first* Suche dazu führen, daß keine Lösung gefunden wird, obwohl es eine Lösung gibt. PROLOG verwendet aus praktischen Gründen die *depth-first* Suche.

Der Beweis eines Ziels G relativ zu einem Programm P terminiert, gdw. G_n = *true*/*false*, für ein $n \geq 0$. Der Beweis ist in diesem Fall endlich und hat die Länge n. In rekursiven Programmen ist es möglich, daß es nicht–terminierende Beweise gibt.

Beispiel (2–5)
Einen nicht–terminierenden Beweis erhalten wir, wenn wir in Beispiel (2–4) immer die zweite Klausel (R 2) zur Reduktion verwenden und als aktuelles Ziel immer die *liegt_ unter*–Klausel wählen:

<{liegt_unter(c, a)}, liegt_unter(c, a), (R 2)>
<{liegt_auf(Z, c), liegt_unter(Z, a)}, liegt_unter(Z, a), (R 2)>
<{liegt_auf(Z, c), liegt_auf(Z1, Z), liegt_unter(Z1, a)},
liegt_unter(Z1, a), (R 2)>
<{liegt_auf(Z, c), liegt_auf(Z1, Z), liegt_auf(Z2, Z1), liegt_unter(Z2, a)},
liegt_unter(Z2, a), (R 2)>
... ...

(∗ nicht-terminierende Berechnung ∗)

2.3 Suchbäume

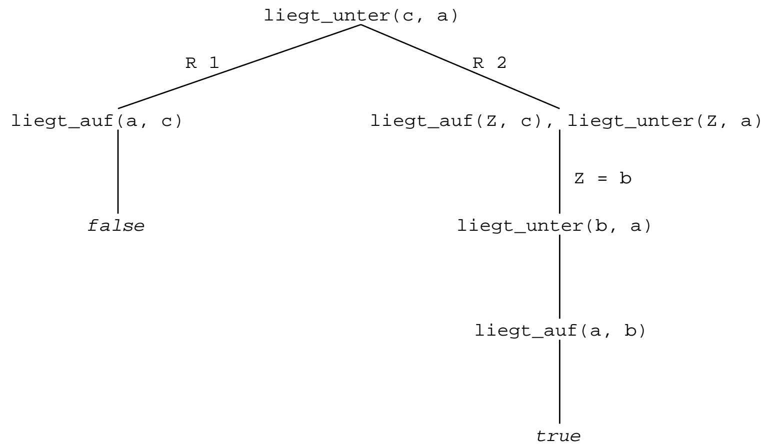
Die Beweisbäume, durch die Berechnungen des einfachen Interpreters für Logikprogramme repräsentiert werden können, basieren auf der Voraussetzung, daß immer die korrekte Klausel zur Reduktion des aktuellen Ziels verwendet wird. Betrachtet man dagegen alle möglichen Reduktionen eines Ziels, kann die Berechnung als *Suchbaum* repräsentiert werden.

Definition (2–3) *Suchbaum*

G sei ein beliebiges Ziel und P ein Logikprogramm. Der Suchbaum S für G relativ zu P wird nach folgendem Verfahren generiert:

- Die Wurzel von S ist G.
- Die Knoten von K sind Konjunktionen von Zielen. Eines dieser Ziel wird für den nächsten Reduktionsschritt ausgewählt.
- Für jede Programmklausel, die mit dem gewählten Ziel unifiziert, wird eine Kante generiert, die von diesem Knoten zu einem neuen Knoten führt: dem durch diese Reduktion erzielten Resolventen.
- Blätter, die aus dem leeren Ziel bestehen, werden mit *true* etikettiert und als *Erfolgsknoten* bezeichnet; alle anderen Blätter repräsentieren das Ende erfolgloser Lösungsversuche.

Beispiel (2–6)
Für das Ziel *liegt_ auf(a, c)* erhalten wir folgenden Suchbaum:



2.4 Prologs Beweisstrategie

Ein reines Prologprogramm ist ein Logikprogramm, in dem die Programmklauseln und die Ziele im Körper einer Klausel in einer bestimmten Reihenfolge verarbeitet werden. Anders formuliert: Die Reihenfolge, in der die Klauseln eines Prädikats angeordnet sind, und die Anordnung der Ziele innerhalb einer Klausel beeinflussen die Semantik eines Prologprogramms.

Logikprogramme, die unter Verwendung des Prolog-Beweisverfahrens ausgeführt werden, bezeichnet man als *reines Prolog*. Dieses Beweisverfahren, das wir im folgenden beschreiben werden, ist eine der möglichen Realisierungen des abstrakten Interpreters für Logikprogramme auf sequentiellen Maschinen. Um auf der Basis des abstrakten Interpreters für Logikprogramme eine funktionstüchtige Programmiersprache zu entwickeln, sind zwei Entscheidungen erforderlich:

1. Es muß festgelegt werden, welches Ziel des Resolventen in jedem Beweisschritt reduziert werden soll.

2. Ein Verfahren zur Selektion der Programmklausel, mit der die Reduktion ausgeführt werden soll, muß spezifiziert werden.

Prologs Beweisstrategie ist durch folgende Entscheidungen charakterisiert:

- Die Ziele des Resolventen werden von *links nach rechts* verarbeitet; d.h. es wird immer das erste Ziel des Resolventen reduziert.

- Bei der Wahl der Programmsuche verwendet Prolog die *depth-first* Suchstrategie mit *Backtracking*: Es wird immer die erste Programmklausel gewählt, deren Kopf mit dem aktuellen Ziel unifizierbar ist. Wenn in einem Reduktionsschritt verschiedene Programmklauseln verwendet werden konnten und die zunächst gewählte zum Scheitern des Beweisversuchs führt, dann wird der Ausgangszustand vor dieser Reduktion wiederhergestellt und die nächste geeignete Programmklausel verwendet.

Wenn der Suchbaum keine nicht-terminierenden Äste enthält, erlaubt diese Beweisstrategie es, alle Lösungen für ein Ziel zu finden. Der Trace eines Prologprogramms ist ein erweiterter Trace eines Logikprogramms:

Beispiel (2–7)
Ausgehend von dem Logikprogramm in Beispiel (2–4) erhalten wir für das Ziel *liegt_unter(X, a)* folgenden Trace:

```
liegt_unter(X, a)
  liegt_auf(a, X)                {<X, b>}                       [R 1]
      true
      X = b
      ;
  liegt_auf(Y, X)                                               [R 2]
      liegt_unter(Y, a)
         liegt_auf(a, Y)         {<Y, b>, <X, c>}               [R 1]
           true
           X = c
           ;
         liegt_auf(Z, Y)                                        [R 2]
           liegt_unter(Z, a)
              liegt_auf(a, Z)    {<Z, b>, <Y, c>, <X, tisch>}   [R 1]
                true
                X = tisch
```

Durch die Eingabe des Semikolons wird Backtracking erzwungen; d.h. Prolog sucht nach einer weiteren Lösung für das Ausgangsziel.

Solange kein Backtracking erfolgt, ähnelt die Ausführung eines Prologprogramms der Ausführung eines Programms einer gewöhnlichen prozeduralen Programmiersprache wie PASCAL z.B.: Eine Klausel A $\leftarrow$ B_1, ..., B_n kann als eine Prozedurdefinition der folgenden Form aufgefaßt werden:

```
PROZEDUR A
    BEGIN
        Aufruf B1
        Aufruf B2
          ⋮
        Aufruf Bn
    END.
```

Unterschiede werden erst deutlich, wenn das Scheitern eines Ziels Backtracking auslöst: In einer konventionellen Programmiersprache führt eine vergleichbare Situation zu einem Programmabbruch und einer Fehlermeldung, in Prolog kann die Berechnung einfach mit der nächsten geeigneten Programmklausel fortgesetzt werden. Ein weiterer Unterschied, auf den wir bereits hingewiesen haben, liegt darin, daß Prologvariablen nicht auf Speicherabschnitte, sondern zunächst nicht näher spezifizierte Objekte verweisen: Destruktive Operationen auf Datenstrukturen sind (normalerweise) nicht möglich. Operationen auf Datenstrukturen, wie z.B.

- Wertzuweisung;
- Parameterübergabe;
- Generierung von RECORD-artigen Strukturen und
- Lese- und Schreibzugriff auf den Feldern dieser Strukturen.

werden ausschließlich durch die (monotone) Operation der Unifikation realisiert.

Teil II

Programmieren in Prolog

Kapitel 3

Erste Schritte

Nachdem wir in den letzten beiden Sitzungen kurz die konzeptuellen Grundlagen der Logikprogrammierung beschrieben haben, wenden wir uns jetzt dem Programmieren in Prolog zu. Zunächst erläutern wir die Grundregel für Benutzung des (SICSTUS-)Prolog-Interpreters. Anschließend formulieren wir einige Richtlinien zur Programmentwicklung und -dokumentation.

3.1 Benutzung des Prolog-Interpreter

Es ist üblich zwischen zwei Typen von Prädikaten zu unterscheiden: Als *Systemprädikate* werden diejenigen Prädikate bezeichnet, die Bestandteil der Programmiersprache Prolog sind, d.h., die dem Benutzer nach Aktivierung des Prologinterpreters zur Verfügung stehen. Alle anderen Prädikate werden als *benutzerdefinierte* Prädikate bezeichnet. Wir werden im neue Prädikate unter Verwendung folgender Konventionen eingeführen:

- *prädikat*/n bezeichnet ein n-stelliges Prädikat mit dem Funktor *prädikat*.
- *prädikat*(+Arg1, –Arg2, ?Arg3, ...) beschreibt ein n-stelliges Prädikat. Die mit '+' versehenen Argumente sind Inputparameter und müssen beim Aufruf des Prädiakts instantiiert sein; die mit '–' markierten Argumente sind Outputparameter und dürfen beim Aufruf des Prädikats nicht instantiiert sein. Alle mit '?' markierten Argumente können beim Aufruf instantiiert sein, müssen es aber nicht.

Wir werden nun kurz die für die Benutzung des SICSTUS–Prolog Interpreters wichtigsten Kommandos beschreiben:

3.1.1 Aufrufen und Verlassen des Interpreters

Der Prolog-Interpreter kann von der Betriebssystemsebene aus einfach durch die Eingabe von *prolog* aufgerufen werden:

```
~ $ prolog
SICStus 3 #5: Fri Nov 1 15:49:55 MET 1996

| ? –
```

Zum Verlassen des Interpreters dient das *halt*/0 Prädikat:

```
?- halt.
~ $
```

3.1.2 Erstellung von Programmen

Zur Erstellung von Prologprogrammen kann ein beliebiger Editor (z.B. *VI* oder *Emacs*) verwendet werden. Um das Programm anschließend in die Prolog–Wissensbasis zu laden, führt man von Prolog aus das Kommando [*Dateiname*] aus. Als Dateiname muß ein Prolog-Atom verwendet werden. Enthält der Dateiname Sonderzeichen oder eine Extension, ist er in einfache Hochkommata zu setzen.

Beispiel (3–1)

```
| ? – [meinedatei].
        {consulting meinedatei}
        yes
| ? – ['meinedatei.pro'].
        {consulting meinedatei.pro}
        yes
```

3.1.3 Anzeige der Wissensbasis

Den Inhalt der Prolog-Wissensbasis kann man sich mit Hilfe der folgenden beiden Prädikate anzeigen lassen:

- *listing*/0 zeigt den vollständigen Inhalt der Datenbasis an.
- *listing*/1 zeigt alle Klauseln eines Prädikats an; d.h. es hat das Format *listing*(+*funktor*). Im Funktornamen dürfen die *wildcard*-Zeichen '?' und '*' verwendet werden. Das Textmuster ist in diesem Fall durch Hochkommata einzuschließen.

Beispiel (3–2)

Wenn die Prolog-Wissensbasis die Klauseln

```
liegt_unter(X, Y) :- liegt_auf(X, Y).
liegt_unter(X, Y) :- liegt_auf(Z, X), liegt_unter(Z, Y).

ergebnis(9).
```

enthält, dann führt der Aufruf von *listing*/0 zu folgender Bildschirmausgabe:

```
liegt_unter(_161, _162) :-
        liegt_auf(_161, _162).
liegt_unter(_163, _164) :-
        liegt_auf(_165, _163),
        liegt_unter(_165, _164).
ergebnis(9).
```

Der Aufruf von *listing*(*ergebnis*) bzw. *listing*(*'er*'*) führt dazu, daß nur das *ergebnis*-Fakt angezeigt wird.

3.2 Programmentwicklung

Kriterien für die Beurteilung von Programmen:

- Korrektheit
- Dokumentation
- Transparenz
- Robustheit
- Effizienz

3.2.1 Schritte der Programmentwicklung

1. *Problemspezifikation*; möglichst präzise umgangssprachliche Beschreibung des Problems und der Problemlösung
2. Wahl der grundlegenden Datenstrukturen und Operationen
3. Formulierung des Algorithmus
4. Implementierung
5. Testen des Programms

empfohlenes heuristisches Prinzip::
top–down Entwicklung (Zerlegen des Ausgangsproblems in einfacher zu lösende Teilprobleme, ...)

3.2.2 Gestaltung von Programmen

- modularer Aufbau
  erhöht die Transparenz und erleichtert es, Programmteile unabhängig voneinander zu testen und so Fehler schneller zu lokalisieren
  - kurze Klauseln (nicht mehr als 3–5 Ziele pro Klausel)
  - kurze Prozeduren (nicht mehr als 3 Klauseln pro Prozedur)
- Formatierung
  - der Kopf einer Klausel beginnt in der ersten Spalte, der Körper dagegen wird eingerückt
  - nur ein Ziel pro Zeile
  - eine Leerzeile zwischen den Klauseln einer Prozedur
  - mindestens zwei Leerzeilen zwischen Prozeduren

3.2.3 Programmdokumentation

(a) Selbstdokumentation

- für Prädikate und Variablen sollten sinnvolle Bezeichner verwendet werden
- Variablen, die Listen bezeichnen, sollten mit einem 'L' enden (z.B. *IntegerL*)
- bei Kopf/Rest–Notation sollte die Rest–Variable mit einem 's' enden (z.B. *[Integer|Integers]*)

(b) Kommentare

- Programmkopf
  * Titel
  * Autor(en)
  * Datum der letzten änderung
  * intendierte Semantik
  * Voraussetzungen
  * Liste aller im Programm definierten Prädikate
- Prozedurkopf
  * Name und Stelligkeit des definierten Prädikats
  * Beschreibung von Wert und Argumenten
  * intendierte Semantik
  * Besonderheiten
- Klausel
  wenn komplex: vorangestellte Kommentarzeile(n); sonst selektive Kommentierung einzelner Ziele

3.3 Das Affe/Banane–Problem

3.3.1 Das Problem

Das Affe/Banane–Problem ist ein *klassisches* Problem aus dem Bereich der künstlichen Intelligenz, auf das man in vielen Einführungs- und Programmierbüchern stößt. Es läßt sich folgendermaßen beschreiben:

> *Gegen sei ein Raum mit einer Tür und einem Fenster. In der Mitte des Raumes hängt eine Banane an einer kurzen Schnur von der Decke. In dem Raum befinden sich ein Affe, der an der Tür steht und eine Kiste, die vor dem Fenster steht. Der Affe ist nicht in der Lage, die Banane vom Boden aus zu erreichen. Das Problem ist, ob es dem Affen gelingen kann, die Banane in seinen Besitz zu bringen.*

Für die meisten menschlichen Betrachter dieses Szenarios liegt die Lösung auf der Hand: Der Affe muß zum Fenster gehen, die Kiste in die Mitte des Raumes schieben, sie anschließend besteigen, bevor er endlich die Banane ergreifen kann. Wie aber ist es möglich, ein Computerprogramm zu entwickeln, das diesen Plan selbstständig findet?

3.3.2 Vorüberlegungen

Zur Lösung des Problems sind vier verschiedene Operationen erforderlich:

1. gehen : von der Tür zum Fenster
   (allgemeiner: von einer Position P1 zu einer neuen Position P2)

2. schieben : der Kiste vom Fenster zur Zimmermitte
   (allgemeiner: der Kiste von einer Position P1 zu einer neuen Position P2)

3. klettern : der Affe klettert auf die Kiste

4. greifen : der Affe greift die Banane

Für jede diese Operationen gibt es bestimmte *Vorbedingungen*: Der Affe kann nur gehen, wenn er sich auf dem Boden (und nicht auf der Kiste) befindet. Er kann die Kiste nur schieben bzw. sie besteigen, wenn er sich direkt neben der Kiste befindet. Die Banane kann er nur greifen, wenn er auf der Kiste steht und sich die Kiste in der Mitte des Raums befindet.

Alle Faktoren, die die Ausführbarkeit der für die Lösung des Problems relevanten Operationen determinieren, bestimmen gemeinsam die aktuelle Problemlösungssituation, kurz *Zustand* genannt. Dazu gehören:

- Position des Affen im Raum (an der Tür, am Fenster, ...)
- Operationsort des Affen (auf dem Boden, auf der Kiste)
- Position der Kiste (am Fenster, in der Mitte des Raumes, ...)
- Erfolg (der Affe hat die Banane, er hat sie nicht)

Ein Zustand kann, wie man leicht sieht, durch ein vierstelliges Prolog–Prädikat repräsentiert werden, das wir – aus naheliegendem Grund – *zustand*/4 nennen:

zustand(Position_ Affe, Affe_ unten/oben, Position_ Kiste, Hat_ Banane?).

Der Ausgangszustand ist der, in dem der Affe an der Tür und die Kiste am Fenster steht:

zustand(an_ der_ tuer, auf_ dem_ boden, mitte, hat_ keine_ banane).

Der Zielzustand läßt sich durch folgendes Fakt beschreiben:

zustand(mitte, auf_ der_ kiste, mitte, hat_ die_ banane).

Die Ausführung einer Operation (von jetzt an: *Zug* genannt) verändert den aktuellen Zustand: Sie führt dazu, das der aktuelle Zustand Z in einen neuen

Zustand Z' übergeht. Jeder Zug hat die Form *zug(AlterZustand, Aktion, Neuer-Zustand)*. Da es vier verschiedene Operationen gibt, sind auch vier verschiedene Züge zu unterscheiden.

Der Affe kann die Banane ergreifen, wenn es eine Folge von Zügen gibt, die den Ausgangszustand in den Zielzustand überführt: Das Affe/Banane–Problem kann also als ein einfaches Suchproblem rekonstruiert werden, zu dessen Lösung die von Prolog verwendete Depth–first–Suche mit Backtracking verwendet werden kann.

3.3.3 Das Programm

```
% zug(Zustand1, Aktion, Zustand2)
% Durch Ausführung von Aktion wird Zustand1 in Zustand2 überführt.

% 1. Klausel: Greifen
% Vorbedingung: Die Kiste befindet sich in der Mitte des Raumes und der Affe sich auf ihr.
% Er hat noch nicht die Banane.
% Neue Situation: Er hat die Banane.

zug(zustand(mitte, auf_der_kiste, mitte, hat_keine_banane),
    greifen,
    zustand(mitte, auf_der_kiste, mitte, hat_die_banane)).

% 2. Klausel: Klettern
% Vorbedingung: Der Affe befindet sich auf dem Boden und an derselben Position wie die
% Kiste.
% Neue Situation: Er ist auf der Kiste.

zug(zustand(P, auf_dem_boden, P, _),
    klettern,
    zustand(P, auf_der_kiste, P, _)).

% 3. Klausel: Schieben
% Vorbedingung: Der Affe befindet sich auf dem Boden und an derselben Position P wie die
% Kiste.
% Neue Situation: Er und die Kiste befinden sich an einer neuen Position P1.

zug(zustand(P, auf_dem_boden, P, _),
    schieben(P, P1),
    zustand(P1, auf_dem_boden, P1, _)).

% 4. Klausel: Gehen
% Vorbedingung: Der Affe befindet sich auf dem Boden an einer Position P.
% Neue Situation: Er befindet sich an einer neuen Position P1.

zug(zustand(P, auf_dem_boden, _, _),
    gehen(P, P1),
    zustand(P1, auf_dem_boden, _, _)).
```

```
% kann_banane_bekommen(Zustand)
% Dieses Prädikat ist erfüllt, wenn von Zustand ein Zustand erreichbar ist, in dem der Affe
% die Banane hat.

% 1. Klausel
% Wenn der Affe die Banane hat, dann kann er sie offensichtlich auch bekommen.

kann_banane_bekommen(zustand(_, _, _, hat_die_banane)).

% 2. Klausel
% Der Affe kann die Banane bekommen, wenn von dem aktuellen Zustand ein anderer
% Zustand erreichbar ist, in dem er sie bekommen kann.

kann_banane_bekommen(Zustand1) :-
    zug(Zustand1, _, Zustand2),
    kann_banane_bekommen(Zustand2).
```

Für das Ziel *kann_banane_bekommen(zustand(an_der_tuer, auf_dem_boden, mitte, hat_keine_banane))* werden folgende Zustände durchlaufen:

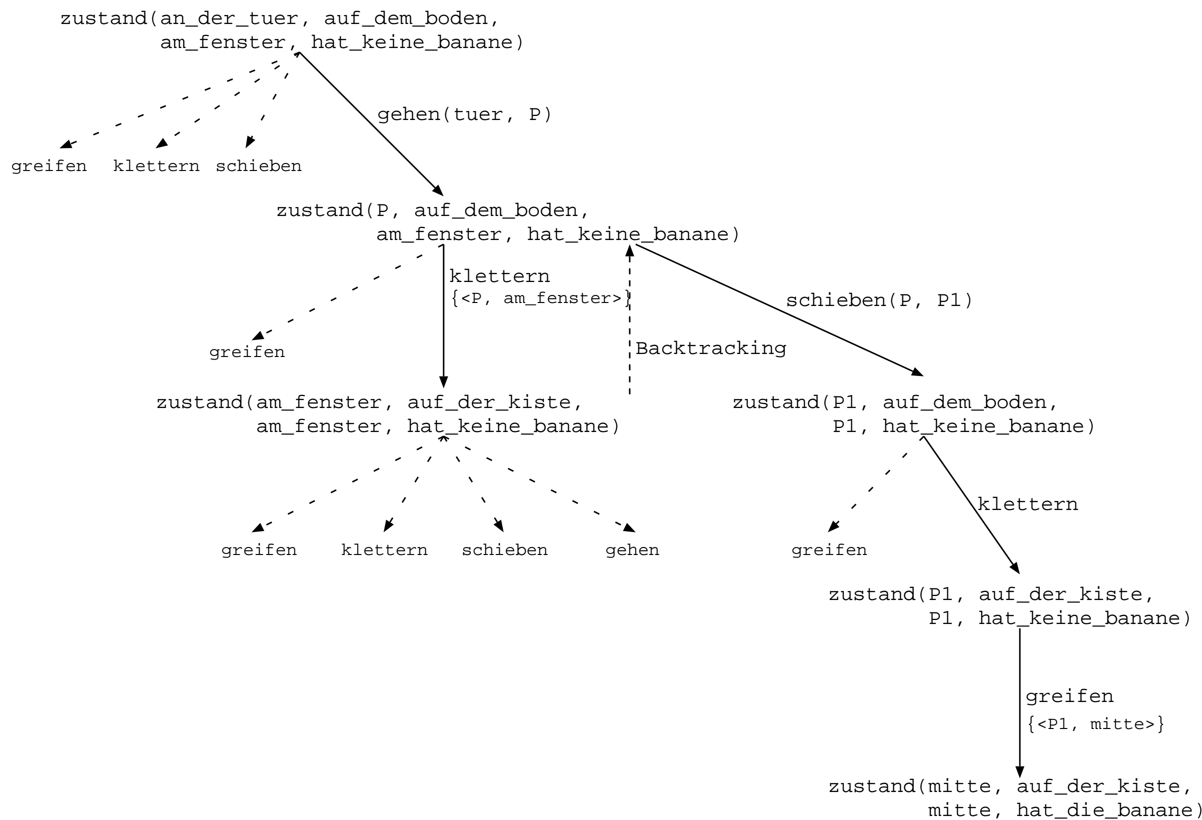


Abbildung (3-1)

Kapitel 4

Listen

Listen bilden eine der wichtigsten in Prolog verfügbaren Datenstrukturen. Eine Liste ist eine prinzipiell beliebig lange Folge von Objekten. Es gibt verschiedene Möglichkeiten, Listen in Prolog zu notieren. Neben den verschiedenen Notationskonventionen werden wir eine Reihe von Prädikaten zur Listenverarbeitung einführen.

4.1 Notationskonventionen

Listen können in Prolog notiert werden, indem ihre Elemente einfach aufgezählt werden (Standardnotation), durch Verwendung des Listenoperators (Operator–Notation) und durch Angabe von Listenkopf und Restliste (erweiterte Standardnotation).

4.1.1 Standardnotation

In der Regel verwendet man die eckigen Klammern '[' und ']' um eine Liste in Prolog zu notieren. So bezeichnet z.B.

- [ ] die leere Liste;
- [a, b, c] die Liste, die aus den drei Atomen a, b und c besteht und
- [a, [ ], b, [c, d]] eine Liste, die zwei Atome und zwei weitere Listen als Elemente enthält.

Die Reihenfolge, in der Objekte in einer Liste vorkommen, ist anders als bei ungeordneten Gruppierungen von Objekten (*Mengen*) wichtig; so gilt z.B. [a, b, c] = [a, b, c], nicht aber [a, b, c] = [c, b, a].

Für jede wohlgeformte Liste L gilt:

- L ist entweder leer oder besteht aus einer (endlichen) Folge von Objekten.

- Wenn L eine nicht–leere Liste ist, dann besteht L – syntaktisch betrachtet – aus zwei Teilen:

  1. dem ersten Element der Liste, auch *Kopf* (head) genannt und
  2. der Restliste oder kurz *Rest* (tail).

Beispiel (4–1)
Gegeben sei eine Liste α = [a, b, c]. Es gilt: *Kopf*(α) = a und *Rest*(α) = [b, c]. Die Liste [b, c] wiederum besteht aus dem Kopf b und dem Rest [c], die selbst aus dem Kopf c und dem Rest [ ] besteht. Die leere Liste ist nicht weiter zerlegbar.

Dieses für Listen geltende syntaktische Prinzip ist der Grund dafür, daß alle (nicht–leeren) Prolog–Listen als binäre Bäume repräsentiert werden können:

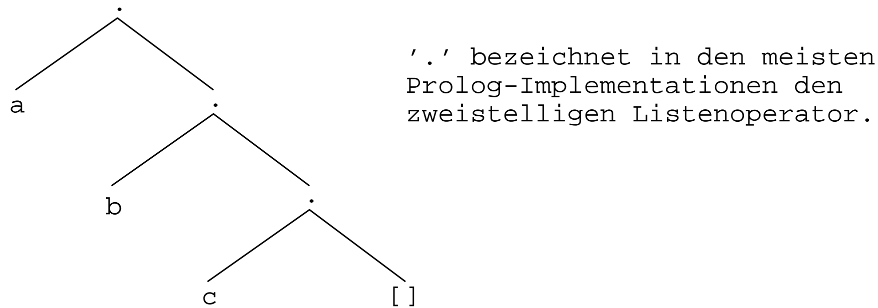


Abbildung (4-1)

Damit kommen wir zur zweiten Notationskonvention für Listen: der Operator–Notation.

4.1.2 Operator–Notation

Wenn α eine Liste ist, mit Kopf(α) = X und Rest(α) = Y, dann gilt: α = .(X, Y)[1].

Beispiel (4–2)

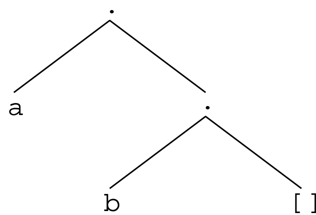
[1]Der Operator '.' entspricht in Lisp der Listenbildungsfunktion CONS.

[c] = .(c, [ ])
[b, c] = .(b, .(c, [ ]))
[a, b, c] = .(a, .(b, .(c, [ ])))

Allerdings sollte man bei der Verwendung der Operator–Notation vorsichtig sein: In Prolog wird jede *echte* Liste aus der leeren Liste aufgebaut; d.h., sie ist äquivalent mit einem Ausdruck in Operator–Notation, in dem das zweite Argument des innersten Listenausdrucks eine leere Liste ist (s.o.). Prinzipiell ist es auch möglich, Pseudolisten zu bilden, die nicht auf die leere Liste zurückführbar sind.

Beispiel (4–3)
.(a, .(b, [ ])) erzeugt eine Liste, die die Objekte a und b enthält;
.(a, b) dagegen eine Pseudoliste, die aus den Objekten a und b besteht.

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




```
    .
  /   \
 a     b
```

Die Generierung und Verwendung von Pseudolisten sollte unbedingt vermieden werden, denn viele für Listen definierte Standardfunktionen führen bei Pseudolisten zu unerwarteten Resultaten: Wenn z.B. α = .(a, b), dann scheitert das Ziel *member(b, α)?*.

4.1.3 Erweiterte Standardnotation

Um in der Standardnotation Kopf und Rest einer Liste zu markieren, verwendet man in Prolog den senkrechten Strich '|'. Wenn also α eine Liste mit Kopf X und Rest Y ist, dann gilt:

$$\alpha = .(X, Y) = [X|Y]$$

Durch Verwendung ist es möglich, eine beliebige Anzahl von Objekten vom Rest der Liste zu trennen:

$$[a, b, c] = [a|[b, c]] = [a, b|[c]] = [a, b, c|[\ ]]$$

4.2 Operationen auf Listen

Wir werden jetzt eine Reihe (z.T. in Prolog schon vordefinierter) Operationen auf Listen einführen. Zu den Standardoperationen auf Listen gehört u.a. das

Löschen bzw. Einfügen von Objekten, das Verknüpfen von zwei Listen und das Suchen nach Vorkommen von Objekten in einer Liste. Jeder Definition eines Prädikates in Prolog sollte ein *Prozedurkopf* vorangestellt werden, der folgende Angaben enthält:

- Prädikatsname/Stelligkeit

- Parameterspezifikation

- Semantik (intendierte Bedeutung)

- Definitionsidee

Da wir die in diesem Kapitel entwickelte Prädikate sehr ausführlich beschreiben, verzichten wir hier in manchen Fällen auf die Angabe der Parameterspezifikation und der Definitionsidee.

4.2.1 Ein Typ–Prädikat für Listen

Semantik. Typprädikate sind einstellige Prädikate die testen, ob es sich bei seinem Argument um ein Objekt eines bestimmten Typs handelt. Das Prädikat *liste*/1 ist erfüllt, wenn es sich bei seinem Argument um eine Liste handelt.

Definitionsidee. Ein Objekt X ist eine Liste gdw. X die leere Liste ist oder X sich in *Kopf* und *Rest* zerlegen läßt, so daß *Rest*(X) eine Liste ist.

Realisierung.

```
% liste/1
liste([ ]).                   % Die leere Liste ist eine Liste.
liste([X|Xs]) :-              % Eine nicht-leere Liste besteht aus einem beliebigen Kopf
  liste(Xs).                  % und einem Rest, der eine Liste ist.
```

Beispiel (4–4)

Echte Liste:	*Pseudoliste*:
liste([a, b, c])	liste([a|b])
liste([b, c])	liste(b)
liste([c])	*false*
liste([ ])	
true	

4.2.2 Suchen eines Objekts in einer Liste

Semantik. Das Prädikat *member(Objekt, Liste)* prüft, ob *Objekt* in *Liste* vorkommt.

Definitionsidee. Ein Objekt ist in einer Liste enthalten, wenn es entweder mit dem Kopf der Liste identisch ist oder im Rest der Liste enthalten ist; d.h., die *member*–Relation läßt sich, wie auch schon die *liste*–Relation, durch eine einfache rekursive Prozedur definieren:

Realisierung.

```
% member/2
% member(X, Liste) ← X ist in Liste enthalten.

member(X, [X|_]).            % Der Kopf der Liste ist das gesuchte Objekt.
member(X, [_|Ys]) :-         % Der Körper der Liste wird durchsucht.
    member(X, Ys).
```

Diese Definition verdeutlicht die Struktur rekursiver Prozeduren. In Prolog muß jede rekursive Prozedur aus mindestens zwei Klauseln bestehen:

1. Eine Klausel bestimmt unter welchen Umständen das Beweisverfahren terminiert (*Abbruchkriterium*). Bei dieser Klausel handelt es sich meistens um an Fakt, das aufgrund des von Prolog verwendeten Beweisverfahrens an den Anfang der Prozedur gestellt werden muß.

2. Eine Klausel zerlegt das Anfangsproblem in einfachere Teilprobleme. Mindestens eines dieser Teilprobleme ist eine Instanz des ursprünglichen Problems (rekursiver Aufruf der Prozedur).

4.2.3 Entfernen von Objekten aus einer Liste

Semantik. Das Prädikat *delete*/3 entfernt ein Vorkommens eines Objektes O aus einer Liste L.

Beispiel (4–5)

```
?- delete(b, [a, b, c, b], X).
   X = [a, c, b]
   yes
?- delete(d, [a, b, c, b], X).
   X = [a, b, c, b]
   yes
```

Definitionsidee. Die Ausgangsliste wird Element für Element abgearbeitet: Wenn der Kopf der Liste mit dem zu löschenden Objekt übereinstimmt, dann ist das Resultat der Rest der Liste. Anderenfalls muß das Objekt in der Restliste gelöscht werden.

Realisierung.

```
% delete/3
% delete(X, Liste1, Liste2) ← Liste2 das Resultat der Löschung eines Vorkommens von X
% in Liste1 ist.
```

```
delete(_, [ ], [ ]).
delete(X, [X|Xs], Xs).
delete(X, [Y|Ys], [Y|Zs]) :-
    delete(X, Ys, Zs).
```

4.2.4 Die *Append*–Relation

Semantik. Durch Verkettung zweier Listen L_1 und L_2 soll eine neue Liste L_3 erzeugt werden, die alle Objekte aus L_1, gefolgt von den Objekten aus L_2, enthält. Wenn also $L_1 = [x_1, \ldots, x_n]$ und $L_2 = [y_1, \ldots, y_m]$, dann gilt: $L_3 = [x_1, \ldots, x_n, y_1, \ldots, y_m]$.

Beispiel (4–6)

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

Definitionsidee. Die Verkettung zweier Listen L1 und L2 kann dadurch realisiert werden, daß alle Elemente von L1 in der richtigen Reihenfolge in eine Kopie von L2 übertragen werden. Dabei müssen zwei Fälle unterschieden werden:

1. Wenn $L_1 = [\,]$, dann ist $L_3 = L_2$.
2. Wenn $L_1 = [Kopf|Rest]$, dann muß *Kopf* in die Liste übertragen werden, die sich durch Verkettung von *Rest* mit L_2 ergibt.

Realisierung.

% append/3
% *append(L1, L2, L3)* ← durch Verkettung von *L1* und *L2* erhält man die Liste *L3*.

```
append([ ], Xs, Xs).                % Abbruchbedingung
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs, Ys, Zs).             % rekursiver Aufruf
```

4.2.5 Die Teillisten–Relation

Semantik. Eine Liste L1 ist eine Teilliste einer Liste L2 wenn die Elemente von L1 einen Abschnitt der Liste L2 bilden, d.h., wenn sie in L2 in derselben Reihenfolge wie in L1 vorkommen und keine anderen Objekte diese Abfolge unterbrechen.

Beispiel (4–7)

```
?- teilliste([a, b], [a, b, c]).
   yes
?- teilliste([b, a], [a, b, c]).
   no
?- teilliste([a, c], [a, b, c]).
   no
```

Definitionsidee. Jede Liste läßt sich ein Anfangssegment (*Präfix*) und ein Endsegment (*Suffix*) zerlegen. L1 ist eine Teilliste von L gdw. es einen Präfix P von L gibt und L1 ein Suffix von P ist, bzw. es einen Suffix S von L gibt und L1 ein Präfix von S ist. Bevor wir von diesen Vorüberlegungen ausgehend das Prädikat *teilliste*/2 definieren, müssen wir zunächst Prädikate für die Präfix– und die Suffix–Relation formulieren:

```
% praefix/2
% praefix(L1, L2) ← L1 ist leer oder L1 ist ein Anfangssegment von L2.
%

praefix([ ], L).

praefix([X|Xs], [X|Ys]) :-
    praefix(Xs, Ys).


% suffix/2
% suffix(L1, L2) ← L1 ist mit L2 identisch oder bildet ein Endsegment von L2.
%

suffix(Xs, Xs).
suffix(Xs, [Y|Ys]) :-
   suffix(Xs, Ys).
```

Realisierung. Mit Hilfe der Präfix– und der Suffix–Relation ist es möglich, eine Reihe unterschiedlicher Definitionen für die Teillisten–Relation zu geben:

```
% teilliste/2

% 1. Variante: Suffix eines Präfixes
teilliste(L1, L2) :-
   praefix(P, L2),
   suffix(L1, P).

% 2. Variante: Präfix eines Suffixes
teilliste(L1, L2) :-
   suffix(P, L2),
   praefix(L1, P).

% 3. Variante: Rekursive Definition
teilliste(L1, L2) :-
```

```
    praefix(L1, L2).
teilliste(L1, [Y|Ys]) :-
    teilliste(L1, Ys).

% 4. Variante: Präfix eines Suffixes (mit append)
teilliste(Xs, PsXsSs) :-
    append(Ps, XsSs, PsXsSs),
    append(Xs, Ss, XsSs).

% 5. Variante: Suffix eines Präfixes (mit append)
teilliste(Xs, PsXsSs) :-
    append(PsXs, Ss, PsXsSs),
    append(Ps, Xs, PsXs).
```

4.3 Multifunktionalität von Prädikaten

Ein großer Vorteil von Prolog–Prädikaten gegenüber Prozeduren in prozeduralen bzw. Funktionen in funktionalen Programmiersprachen liegt in ihrer *Multifunktionalität*: Abhängig davon, welche ihrer Argumente instantiiert werden, lassen sich mehrstellige Prolog–Prädikate häufig zu ganz verschiedenen Zwecken verwenden.

Beispiel (4–8)

(a) *prädikative* Verwendung:

```
append([a, b], [c], [a, b, c]).            member(a, [a, b, c]).
```

(b) mögliche *funktionale* Verwendungen:

```
% Verkettung zweier Listen:
   append([a, b], [c], X).
   X = [a, b, c]
   yes
```

```
% alle Zerlegungen einer Liste          % alle Elemente einer Liste
% anzeigen lassen:                      % anzeigen lassen:
append(L1, L2, [a, b, c]).                 member(X, [a, b, c]).
L1 = [ ]        L2 = [a, b, c];            X = a;
L1 = [a]        L2 = [b, c];               X = b;
L1 = [a, b]     L2 = [c];                  X = c;
L1 = [a, b, c]  L2 = [ ];                  no
no
```

```
% Differenz aus Ergebnisliste und einer Argumentliste berechnen:
   append(L1, [c], [a, b, c]).
   L1 = [a, b]
   yes
   append([a, b], L2, [a, b, c]).
```

L2 = [c]
yes

% das erste Element einer Liste extrahieren:
append([X], _, [a, b, . . .]).
X = a
yes

% das letzte Element einer Liste extrahieren:
append(_, [X], [a, b, . . . , y, z]).
X = z
yes

(c) unmögliche *funktionale* Verwendungen:
append(L1, [b, c], L3). member(a, L).
append([a], L2, L3).

Kapitel 5

Arithmetik

Prolog ist sicher keine Programmiersprache, die für komplexere arithmetische Berechnungen besonders geeignet ist. Allerdings bieten modernere Prolog-Dialekte mittlerweile ein beträchtliches Inventar an arithmetischen Systemprädikaten, die für die meisten Probleme vollkommen ausreichend sind. Wir werden in diesem Kapitel kurz die wichtigsten arithmetischen Operatoren und Prädikate beschreiben.

5.1 Arithmetische Operatoren und Systemprädikate

In allen Prolog-Implementierungen gibt es mindestens zwei Typen von Zahlen:

1. ganze Zahlen (*INTEGER*): ...-3, -2, -1, 0, 1, 2, 3 ... und
2. reele Zahlen (*REAL*): ... -3.14555 ... 0.0 ... 3.14555 ....

Es stehen außerdem die folgenden arithmetischen Operatoren zur Verfügung:

Operator	*Argumenttyp*	*Bedeutung*
$+$	INTEGER/REAL	Addition
$-$	INTEGER/REAL	Subtraktion
$*$	INTEGER/REAL	Multiplikation
$/$	REAL	Division
$//$	INTEGER	Division
mod	INTEGER	ganzzahlige Rest der Division

Alle diese Operatoren können – wie in Prolog üblich – als Präfix– oder als Infixoperatoren verwendet werden:

$$+(7, 99) \iff 7 + 99$$
$$*(3, -(7, 2)) \iff 3 * (7 - 2)$$

Außerdem gibt es die folgenden binären Vergleichsoperatoren:

<	kleiner
=<	kleiner gleich
>	größer
>=	größer gleich
=:=	gleich
=\=	ungleich

Eine Frage, die sich aufdrängt, ist die, wie man einer Variablen den Wert eines arithmetischen Ausdrucks zuweisen kann. Der Begriff der Wertzuweisung stammt aus imperativischen bzw. prozeduralen Programmiersprachen und ist mit dem Konzept einer rein deklarativen Programmiersprache genaugenommen nicht vereinbar. Normalerweise wird in Prolog eine Variable via Unifikation mit einem Objekt assoziiert. Als Unifikationsoperator dient das '='-Zeichen ('\=' steht für *unifiziert nicht*). Ein Ziel der Form $term_1 = term_2$ gelingt also, wenn beide Terme miteinander unifizierbar sind. Ist einer der beiden Terme eine nicht-instantiierte Variable, so wird sie mit dem anderen Term instantiiert; d.h. an ihn gebunden.

Beispiel (5–1)

```
?- X = (3 * 27) + 11.
   yes
?- X = (3 * 27) + 11, X > 50.
   no
   % Andere Möglichkeit: Eine Fehlermeldung, da die Variable X
   % an eine Struktur und nicht an eine Zahl gebunden ist.
?- 11 = 27 – 16.
   no
?- 2 * 3 = 3 * 2.
   no
?- Y = 3 + 8, X = 3 + 8, X = Y.
   yes
?- X = 3 + 8, X = Y.
   Y = 3 + 8
   yes
```

Es gibt in Prolog einen besonderen Operator, der die Auswertung eines arithmetischen Ausdrucks erzwingt und eine ungebundene Variable mit dem Wert des Ausdrucks instantiiert: *is*/2. Dieser Operator kann als Infixoperator verwendet werden. *Is*–Ziele haben die Form: *Term is Ausdruck*.

Bei diesen Zielen wird nur der rechte Term evaluiert und anschließend der aus dieser Berechnung resultierende Wert mit dem ersten Term unifiziert. In den meisten Fällen handelt es sich bei dem ersten Term um eine nicht-instantiierte Variable, die so mit dem Wert des arithmetischen Ausdrucks, der selbst keine ungebundene Variable enthalten darf, instantiiert wird.

Beispiel (5–2)

?- X is 8 + 3
  X = 11
  yes
?- X is 13 mod 3, Y is 22 -21, X = Y.
  X = 1
  Y = 1
  yes
?- 3 is 27 – 24.
  yes
?- 2 * 3 is 3 * 2.
  no
?- Y is X + 8.
  EXCEPTION: arith_ expr_ expected: _ 133 is _ 144 + 8
  % Der arithmetische Term enthält eine ungebundene Variable!
?- X = 8, Y is X - 3.
  X = 8
  Y = 5
?- X is test.
  EXCEPTION: arith_ expr_ expected: _ 133 is test
  % *test* ist kein arithmetischer Term!

Ziele, die mit Hilfe der oben angegebenen Vergleichsoperatoren formuliert werden, dürfen ebenfalls keine nicht-instantiierten Variablen enthalten. Als Argumente können ausschließlich arithmetischen Ausdrücke bzw. Variablen, die mit arithmetischen Ausdrücken instantiiert sind, verwendet werden.

Beispiel (5–3)

?- 8 + 3 = 3 + 8.
  no
?- 8 + 3 =:= 3 + 8.
  yes
?- (33 – 11) * 2 = 44.
  no
?- (33 – 11) * 2 =:= 44.

yes
?- 33 * Y = X * 8.
X = 33
Y = 8
yes
?- 33 * Y =:= X * 8.
*EXCEPTION: arith_ expr_ expected: 33 * _133 =:= _134 * 8*
% Beide arithmetische Terme enthalten ungebundene Variablen!

5.2 Zahlenspiele

Fakultät. Wir werden jetzt eine Reihe von Beispielen geben, die zeigen sollen, wie sich mit den in Prolog zur Verfügung stehenden arithmetischen Operatoren einfache Prädikate definieren lassen.

Ein beliebtes Beispiel, das in keinem Programmiersprachenkurs fehlen darf, bildet die Fakultätsfunktion, die für natürliche Zahlen definiert ist. Zwei Fälle sind zu unterscheiden:

- Die Fakultät von 0 ist 1.
- Die Fakultät einer Zahl N > 0 erhält man, indem man N mit der Fakultät von N − 1 multipliziert.

Wir erhalten so folgende Prädikatsdefinition:

```
% fakultaet(+N, ?Wert)
% Das Prädikat berechnet die n–te Fakultät.

fakultaet(0, 1).
fakultaet(Z, W) :-
    Z > 0,
    Z1 is Z - 1,
    fakultaet(Z1, W1),
    W is Z * W1.
```

Maximum. Das nächste Beispiel zeigt, daß es bereits bei relativ einfachen Problemen sinnvoll sein kann, zusätzliche Hilfsprädikate zu definieren. Das Prädikat *maximum*/2 soll die größte Zahl in einer Liste von Zahlen bestimmen. Eine möglichkeit dieses Ziel zu erreichen besteht darin, die Liste Element für Element abzuarbeiten (durch *Kopf*/ *Rest*–Zerlegung). Zu diesem Zweck ruft das Hauptprädikat ein dreistelliges Hilfsprädikat auf, das als zusätzliches Argument die bislang größte in der Liste gefundene Zahl erhält:

% maximum(+Liste, ?Zahl).
% Ein Ziel der Form *maximum(liste, zahl)* trifft zu, wenn *zahl* die größte Zahl in *liste* ist.
% *maximum*/2 verwendet ein Hilfsprädikat *maximum*/3, dessen zweites Argument die bis-
% lang größte gefundene Zahl bildet; d.h. zunächst wird es mit dem ersten Element der Liste
% instantiiert.

```
maximum([X|Xs], M) :-
    maximum(Xs, X, M).

maximum([ ], M, M).
maximum([X|Xs], Y, M) :-
    X =< Y,
    maximum(Xs, Y, M).
maximum([X|Xs], Y, M) :-
    X > Y,
    maximum(Xs, X, M).
```

Zahlenfolge. Das folgende Programm kann verwendet werden, um eine Liste von Zahlen innerhalb eines spezifizierten Bereichs zu generieren. Es kann allerdings nicht verwendet werden, um die Unter- bzw. Obergrenze einer gegebenen Liste zu berechnen, da die Variablen im <-Ziel in der zweiten Klausel in diesem Fall nicht instantiiert wären!

```
% bereich(+Untergrenze, +Obergrenze, ?Zahlenliste).
% Dieses Prädikat generiert eine Liste von Zahlen (Typ: INTEGER) im Bereich zwischen
% untergrenze und obergrenze.

bereich(N, N, [N]).
bereich(M, N, [M|Ns]) :-
    M < N,
    M1 is M + 1,
    bereich(M1, N, Ns).
```

Länge. Das abschließende Beispiel verdeutlicht noch einmal den Unterschied zwischen Unifikation (+/2) und Evaluierung (is/2). Das Prädikat *laenge*/2 kann verwendet werden, um die Länge einer Liste zu berechnen:

```
% laenge(+Liste, ?Zahl).
% Das Prädikat berechnet die Länge einer Liste (= Zahl ihrer Elemente).

laenge([ ], 0).
laenge([_|Rest], N) :-
    laenge(Rest, N1),
    N = 1 + N1.

% Diese Definition des Prädikats liefert interessante Ergebnisse:

?- laenge([a, b, [c, d], c], X).
   X = 1+(1+(1+(1+0)))
   yes
?- laenge([a, b, [c, d], c], X), Laenge is X.
```

X = 1+(1+(1+(1+0)))
Laenge = 4
yes

Wenn man den Unifikationsoperators in der zweiten *laenge*–Klausel durch den *is*–Operator ersetzt, verhält sich das Prädikat wie erwünscht.

Kapitel 6

Strukturen

In diesem Kapitel beschreiben wir zunächst kurz, wie binäre Bäume in Prolog durch Strukturen repräsentiert werden können und führen anschließend die von Prolog zur Analyse von Strukturen bereitgestellten Systemprädikaten.

6.1 Binäre Bäume

Binäre Bäume bilden einen rekursiven Datentyp, der sich zur Repräsentation verschiedenster Typen von Objekten eignet (Listen, Sequenzen von Ja/Nein-Entscheidungen). Umgekehrt lassen sich binären Bäume in Prolog durch *Strukturen* abbilden. Für binäre Bäume gilt folgendes Strukturprinzip:

- Der leere Baum ist ein binärer Baum.
- Ein nicht-leerer binärer Baum besteht aus einer *Wurzel*, einem *linken* und einem *rechten Teilbaum*. Jeder Teilbaum eines binären Baums ist selbst wieder ein binärer Baum.

Den leeren Baum repräsentieren wir durch das Atom *nil*; nicht-leere binäre Bäume repräsentieren wir durch das Prädikat *baum*/3 (baum(Wurzel, LinkerTeilbaum, RechterTeilbaum)).

Der nächste Schritt besteht darin, ein Typprädikat zu definieren, das nur auf Objekte diesen Typs zutrifft:

```
% binaerer_baum(+Baum).

binaerer_baum(nil).                          % Der leere Baum ist ein binärer Baum.

binaerer_baum(baum(Wurzel, Links, Rechts)) :-
    binaerer_baum(Links),
    binaerer_baum(Rechts).
```

Wir definieren jetzt zwei Prädikate *ist_knoten*/2 und *iso_baum*/2.

```
% ist_knoten(+Baum, ?Knoten) :-
% es gibt in dem binären Baum BAUM einen Knoten KNOTEN.

ist_knoten(X, baum(X, Links, Rechts)).        % X ist die Wurzel des Baums.

ist_knoten(X, baum(Y, Links, Rechts)) :-
    ist_knoten(X, Links).                     % X ist ein Knoten des linken Teilbaums.

ist_knoten(X, baum(Y, Links, Rechts)) :-
    ist_knoten(X, Rechts).                    % X ist ein Knoten des rechten Teilbaums.
```

Das Prädikat *iso_baum*/2 prüft, ob zwei binäre Bäume strukturgleich sind. Strukturgleichheit liegt vor, wenn sie eine identische Wurzel besitzen und ihre Teilbäume übereinstimmen. Da in einfachen binären Bäumen keine Ordnungsrelation über ihren Knoten definiert ist, muß geprüft werden, ob der linke Teilbaum des ersten Baums mit dem linken oder dem rechten Teilbaums des zweiten Baums übereinstimmt und entsprechend muß der rechte Teilbaum des ersten Baums mit dem anderen Teilbaum des zweiten Baums strukturgleich sein.

```
% iso_baum(?Baum1, ?Baum2).
% Das Prädikat trifft zu, wenn beide Bäume strukturgleich (isomorph) sind.

iso_baum(nil, nil).

iso_baum(baum(X, Links1, Rechts1), baum(X, Links2, Rechts2)) :-
    iso_baum(Links1, Links2),
    iso_baum(Rechts1, Rechts2).

iso_baum(baum(X, Links1, Rechts1), baum(X, Links2, Rechts2)) :-
    iso_baum(Links1, Rechts2),
    iso_baum(Rechts1, Links2).
```

6.2 Strukturuntersuchungen

Wir werden im folgenden zwei Typen von Prädikaten einführen:

- Typprädikate und
- Prädikate zur Analyse von Strukturen.

6.2.1 Typprädikate

Typprädikate repräsentieren einstellige Relationen, die von bestimmten Klassen von Objekten erfüllt werden. Wir haben bislang folgende Typen von Objekten kennengelernt: Variablen, Konstanten (mit den Subtypen: Zahlen und Atome) und Strukturen. Prolog stellt für diese Objekte[1] folgende Typprädikate zur Verfügung:

- *atom*(+Objekt) – *Objekt* ist ein Atom; d.h. eine symbolische Konstante.

[1] Ausgenommen Variablen, die wir zu einem späteren Zeitpunkt behandeln werden.

- *atomic*(+Objekt) – *Objekt* ist atomar; d.h. eine numerische/symbolische Konstante.
- *integer*(+Objekt) – *Objekt* ist eine Zahl vom Typ INTEGER.
- *real*(+Objekt) – *Objekt* ist eine Zahl vom Typ REAL.
- *numeric*(+Objekt) – *Objekt* ist eine Zahl (beliebiger zulässiger Typ).
- *compound*(+Objekt) – *Objekt* ist eine Struktur.

Die Semantik dieser Prädikate kann man sich konzeptuell betrachtet durch eine Tabelle geeigneter Fakten determiniert denken.

Beispiel (6–1)
Das Verhalten dieser Prädikate illustriert der folgende Dialog:

?- integer(0).
yes

?- real(0).
no

?- numeric(0.0).
yes

?- atomic(0.0).
yes

?- atom(0.0).
no

?- atom(hallo).
yes

?- atom(vater(X, kurt)).
no

?- atomic(vater(X, kurt)).
no

?- compound(vater(X, kurt)).
yes

In allen Fällen, in denen ein Ziel diesen Typs mit einer nicht-instantiierten Variablen aufgerufen wird, scheitert der Beweis:

Beispiel (6–2)

?- numeric(X).

no

?- X = 77, numeric(X).
yes

?- compound(X).
no

?- X = hanoi(a, b, c, Y), compound(X).
yes

Ein einfaches Beispiel, das die sinnvolle Verwendung von Typprädikaten verdeutlicht, ist die *flatten*/2-Relation, die zwischen zwei Listen Xs, Ys besteht, wenn Ys eine nicht-verschachtelte Liste ist, die alle Elemente von Xs enthält. Die einfachste Realisierung dieser Relation bietet ein birekursives Programm, in dem die folgenden Fälle unterschieden werden:

1. Die leere Liste ist eine nicht-verschachtelte Liste: *flatten([ ], [ ]).*

2. Ist die erste Liste nicht leer, dann '*glätte*' ihren Kopf und ihren Rest und verknüpfe beides miteinander:
   ```
   flatten([X|Xs], Ys) :-
       flatten(X, Ys1),
       flatten(Xs, Ys2),
       append(Ys1, Ys2, Ys).
   ```

3. Da durch die zweite Klausel eine nicht-leere Liste sukzessive in ihre Elemente zerlegt wird, muß es eine Klausel geben, die den Fall behandelt, daß das erste Element der Liste eine Konstante ist:
   ```
   flatten(X, [X]) :-
       atomic(X),
       X \= [ ].
   ```
   Das zweite Ziel in dieser Klausel ist erforderlich, da gilt:
   ?- atomic([ ]).
   yes

Das Programm im überblick:

```
flatten([ ], [ ]).

flatten([X|Xs], Ys) :-
    flatten(X, Ys1),
    flatten(Xs, Ys2),
    append(Ys1, Ys2, Ys).

flatten(X, [X]) :-
    atomic(X),
    X \= [ ].
```

6.2.2 Analyse von Termen

Es gibt in Prolog die Möglichkeit, Strukturen in ihre Bestandteile zu zerlegen: Die Systemprädikate *functor*/3 und *arg*/3 ermöglichen den Zugriff auf den Funktor und die Argumente eines Terms:

- functor(Term, Funktor, N) :-
  Term ist ein Term mit dem Funktor *Funktor* und *N* Argumenten.

- arg(N, Term, Arg) :-
  Arg ist *N*-tes Argument des Terms *Term*.

Beide Prädikate können sowohl zum Zerlegen einer gegebenen Struktur, wie zur Generierung einer neuen Struktur verwendet werden.

Beispiel (6–3)

?- functor(vater(haran, lot), vater, 2).
yes

?- functor(vater(haran, lot), haran, 2).
no

?- functor(vater(haran, lot), X, Y).
X = vater
Y =2

?- functor(X, vater, 2).
X = vater(_152, _153).

?- functor(X, Y, 2).
ERROR: atom_expected

?- functor(X, vater, Y).
ERROR: integer_expected

?- arg(1, vater(haran, lot), haran).
yes

?- arg(2, vater(haran, lot), X).
X = lot

?- arg(2, vater(haran, X), lot).
X = lot

ähnlich wie *functor*/3 kann auch *arg*/3 als Selektor und als Konstruktor verwendet werden; d.h. um auf bestimmte Teile eines Term zuzugreifen bzw. sie zu spezifizieren.

Beispiel (6–4)
In folgenden Fällen scheitern die *arg*-Ziele bzw. führen sie zu Fehlermeldungen:

?- arg(2, vater(haran, lot), abraham).
no

?- arg(2, vater, X).
ERROR: structure_expected

?- arg(3, vater(haran, lot), X).
ERROR: out_of_range: ...

?- arg(2, X, Y).
ERROR: structure_expected

?- arg(2, X, haran).
ERROR: structure_expected

?- arg(X, vater(haran, lot), haran).

ERROR: integer_ expected

Wie leistungsfähig beide Prädikate sind, läßt sich gut bei der Definition der *subterm*-Relation zeigen, die von zwei Termen *Term1* und *Term2* erfüllt wird, wenn *Term1* in *Term2* vorkommt:

```
% subterm(Term1, Term2).
% Das Prädikat ist erfüllt, wenn Term1 in Term2 vorkommt.
subterm(Term, Term).

subterm(Subterm, Term) :-
    compound(Term),
    functor(Term, F, N),
    subterm(N, Subterm, Term).

subterm(N, Subterm, Term) :-
    arg(N, Term, Arg),
    subterm(Subterm, Arg).

subterm(N, Subterm, Term) :-
    N > 1,
    N1 is N - 1,
    subterm(N1, Subterm, Term).
```

Erläuterung der Programmklauseln:

1. Klausel 1 charakterisiert die *Subterm*-Relation als eine reflexive Relation.
2. Wenn das erste Argument eines *subterm*/2-Ziels eine Struktur ist, dann wird sie zerlegt, und die Argumente dieser Struktur werden von rechts nach links mit Hilfe des *subterm*/3-Prädikats durchsucht.
3. überprüfe, ob *Subterm* ein Subterm des N-ten Arguments ist.
4. Gibt es noch weitere Argumente, dann untersuche sie der Reihe nach.

Subterm/2 kann verwendet werden, um zu prüfen, ob ein gegebener Term Subterm eines anderen Terms ist oder um alle Subterme eines Terms zu generieren:

Beispiel (6–5)

```
?- subterm(a, test(versuch(a, 3), erfolgreich)).
   yes
?- subterm(a, test(b, c)).
   no
?- subterm(X, test(b, c)).
```

X = test(b, c) ;
X = c ;
X = b ;
no (more) solutions

Einen etwas komplexeren Fall bildet die *substitute*/2-Relation, die zwei Objekte *Alt*, *Neu* und zwei Terme *AlterTerm*, *NeuerTerm* erfüllen, wenn *NeuerTerm* aus *AlterTerm* durch Ersetzung jeden Vorkommens von *Alt* in *AlterTerm* durch *Neu* resultiert:

```
% substitute(+Alt, +Neu, +AlterTerm, ?NeuerTerm) :-
   % NeuerTerm ist das Ergebnis der Ersetzung aller Vorkommen von Alt in
   % AlterTerm durch Neu.

substitute(Alt, Neu, Alt, Neu).                 % Der vollständige Term wird ersetzt.

substitute(Alt, Neu, Term, Term) :-
   atomic(Term),                                % Der Term ist atomar und
   Term \= Alt.                                 % unifiziert nicht mit Alt.

substitute(Alt, Neu, Term, Term1) :-
   compound(Term),                              % Der Term ist eine Struktur.
   functor(Term, F, N),                         % Der Ausgangsterm wird zerlegt.
   functor(Term1, F, N),                        % Der Zielterm wird generiert.
   substitute(N, Alt, Neu, Term, Term1).        % Substitution in den Argumenten.

substitute(N, Alt, Neu, Term, Term1) :-
   N > 0,
   arg(N, Term, Arg),                           % Argumentabarbeitung:
   substitute(Alt, Neu, Arg, Arg1),             % von rechts nach links.
   arg(N, Term1, Arg1),                         % Generierung des neuen Arguments.
   N1 is N-1,                                   % Dekrementierung des Argument-Pointers.
   substitute(N1, Alt, Neu, Term, Term1).

substitute(0, Alt, Neu, Term, Term1).           % Alle Argumente abgearbeitet.
```

Erläuterung des Programms:

1. Wenn der Ausgangsterm mit dem Objekt *Alt* identisch ist, dann besteht der Zielterm aus dem Objekt *Neu*.

2. Ist der Ausgangsterm keine Struktur und die erste Bedingung nicht erfüllt, dann sind Ausgangs- und Zielterm identisch; d.h. es konnte keine Substitution ausgeführt werden.

3. Der Ausgangsterm ist eine Struktur. Diese Struktur wird zerlegt, und es wird eine Zielstruktur mit demselben Funktor und zunächst nicht-instantiierten Variablen als Argumenten generiert. Diese Variablen werden

mit den Termen instantiiert, die sich durch die Ersetzung aller Vorkommen von *Alt* durch *Neu* in den ihnen im Ausgangsterm korrespondierenden Argumenten ergeben.

4. Das Prädikat *substitute*/5 arbeitet alle Argumente des Ausgangsterms von rechts nach links ab.

Neben dem *functor* und *arg* gibt es in Prolog noch ein drittes Prädikat, das zur Analyse bzw. Generierung von Strukturen verwendet werden kann: =../2. Aus historischen Gründen spricht man vom *univ*-Operator. Er wird als Infixoperator verwendet und von einer Struktur und einer Liste erfüllt, wenn die Liste aus dem Funktor, gefolgt von seinen Argumenten, besteht:

Beispiel (6–6)

```
?- vater(haran, lot) =.. [vater, haran, lot].
   yes
?- vater(haran, lot) =.. X.
   X = [vater, haran, lot]
?- X =.. [vater, haran, lot].
   X = vater(haran, lot)
```

Das *univ*/2-Prädikat kann also verwendet werden, um eine Struktur in eine Liste zu überführen bzw. auf Grundlage einer Liste eine Struktur zu bilden. Mit Hilfe dieses Prädikats lassen sich Programme formulieren, die einfacher sind als solche, die *functor* und *arg* verwenden:

Beispiel (6–7)
Mit Hilfe des *univ*-Prädikats läßt sich das *subterm*/2-Prädikat wie folgt definieren:

```
subterm2(Term, Term).
subterm2(Sub, Term) :-
   compound(Term),
   Term =.. [F|Args], subterm_list(Sub, Args).
subterm_list(Sub, [Arg|Args]) :-
   subterm2(Sub, Arg).
subterm_list(Sub, [Arg|Args]) :-
   subterm_list(Sub, Args).
```

Allerdings sind die mit *functor*/3 und *arg*/3 formulierten Programme effizienter, da keine Zwischenterme (Listen bzw. Strukturen) generiert werden.

Der *univ*-Operator läßt sich seinerseits durch *functor*/3 und *arg*/3 reformulieren. Allerdings sind für die beiden unterschiedlichen Verwendungsweisen dieses Operators auch zwei Programme erforderlich:

```
% univ(Term, List) :-
% List ist eine Liste, die den Funktor von Term enthält, gefolgt von den Argumenten von
% Term.
% Version 1
% Umwandlung einer Struktur in eine Liste.

univ1(Term, [F|Args]) :-
   compound(Term),
   functor(Term, F, N),
```

```
    args1(0, N, Term, Args).


args1(I, N, Term, [Arg|Args]) :-
    I < N,
    I1 is I + 1,
    arg(I1, Term, Arg),
    args1(I1, N, Term, Args).
    args1(N, N, Term, [ ]).

% Version 2
univ2(Term, [F|Args]) :-
    anzahl(Args, N),
    functor(Term, F, N),
    args2(Args, Term, 1).
args2([Arg|Args],Term, N) :-
    arg(N, Term, Arg),
    N1 is N + 1,
    args2(Args, Term, N1).
args2([ ], Term, N).
```

Kapitel 7

Endliche Automaten

Endliche Automaten[1] bilden den einfachsten Typ abstrakter Maschinen, die zur Bearbeitung computerlinguistischer Aufgabenstellungen verwendet werden können. Endliche Automaten sind einfach zu implementieren und erlauben effiziente Problemlösungen. Ihre mathematischen Eigenschaften sind bekannt. Allerdings ist ihre Leistungsfähigkeit begrenzt, so daß sie sich nur für bestimmte Problemstellungen eignen: Besonders in der Phonologie und Morphologie - vereinzelt auch in Syntax und Semantik - wurden und werden endliche Automaten verwendet.

7.1 Grundlegende Konzepte

Endliche Automaten sind abstrakte Maschinen, die Zeichen- bzw. Symbolfolgen verarbeiten und diese akzeptieren oder zurückweisen. Es gibt zwei Typen von endlichen Automaten: nicht-deterministische und deterministische Automaten (NEA/DEAs), die eine spezielle Klasse von NEAs bilden.

Definition (7–1) *Nicht-deterministischer endlicher Automat*

[1] Endliche Automaten werden in der englischsprachigen Literatur als „finite state automata“ (FSA) bezeichnet.

Ein *nicht-deterministischer endlicher Automat* $M = <Q, \Sigma, \sigma, q_0, F>$ besteht aus:

1. einer endlichen Menge von Zuständen $Q = \{q_0, q_1, \ldots, q_n\}$;
2. einer endlichen Menge $\Sigma = \{w_1, \ldots, w_m\}$ von Eingabesymbolen;
3. der Übergangsfunktion $\sigma{:}Q \times (\Sigma \cup \{e\}) \rightarrow POT(Q)$[2];
4. dem Anfangszustand $q_0 \in Q$ und
5. einer Menge F (F ist eine Teilmenge von Q) von Endzuständen.

Deterministische endliche Automaten unterscheiden sich von NEAs dadurch, daß von jedem Zustand unter Verarbeitung eines beliebigen Eingabesymbols höchstens ein Zustand erreichbar ist; d.h. σ ordnet jedem $<q_i, w_j> \in Q \times \Sigma$ ein Element aus Q zu.

Definition (7–2) *Konfiguration*

Wenn $M = <Q, \Sigma, \sigma, q_0, F>$ ein endlicher Automat ist, dann ist jedes $<q, w> \in Q \times \Sigma^*$ eine *Konfiguration* von M.
Eine Konfiguration $<q_0, w>$ wird als „Anfangskonfiguration“ bezeichnet; eine Konfiguration $<q, e>$, mit $q \in F$, als „Endkonfiguration“.

M kann von einer Konfiguration $<q, aw>$ in die Konfiguration $<q', w>$ übergehen - notiert als: $<q, aw> \vdash <q', w>$ - gdw. $q' \in \sigma(q, a)$.
„$\vdash^*$“ bezeichnet den reflexiven transitiven Abschluß der Übergangsrelation.

Definition (7–3) *Akzeptierte Sprache*

Wenn $M = <Q, \Sigma, \sigma, q_0, F>$ ein endlicher Automat ist, dann ist die von M *akzeptierte Sprache* definiert als:

$L(M) = \{w \mid w \in \Sigma^* \wedge <q_0, w> \vdash^* <q, e>$ für ein beliebiges $q \in F\}$.

Beispiel (7–1)

$M_1 = <\{1, 2, 3, 4\}, \{h, a, !\}, \sigma, 1, \{4\}>$, mit

σ	h	a	!
1	2		
2		3	
3	2		4
4			

M_1 ist ein DEA: von jedem Zustand ist höchstens ein neuer Zustand erreichbar. Außerdem gibt es Konfigurationen, von denen aus keine andere Konfiguration

[2] Die Übergangsfunktion σ ist eine *partielle* Funktion; d.h. sie muß nicht für alle $<q, w> \in Q \times (\Sigma \cup \{e\})$ definiert sein.

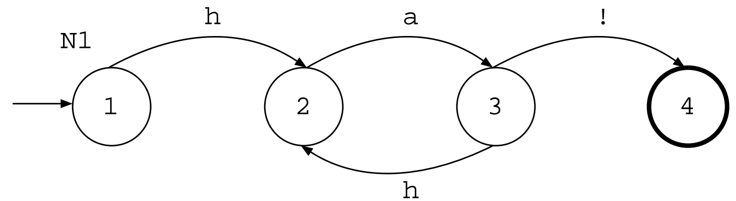
erreicht werden kann; so z.B. <1, a>.
Die von M_1 akzeptierte Sprache besteht aus einer oder mehreren Folgen von „ha“s mit abschließendem '!'; d.h. $L(M_1) = (ha)^+!$.

7.2 Repräsentationsmöglichkeiten

Eine beliebte Repräsentationsform für endliche Automaten bilden die endlichen Übergangsnetzwerke (*finite transition networks*): Die Knoten des Netzes repräsentieren die Zustände des Automaten, und die Kanten zwischen zwei Knoten sind mit dem Symbol des Eingabevokabulars etikettiert, das beim Übergang vom ersten zum zweiten Knoten verarbeitet wird. Start- und Endzustände werden besonders markiert.

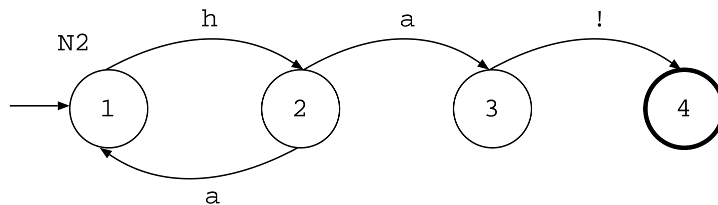
Beispiel (7–2)

1. Der Automat M_1 wird durch folgendes Übergangsnetzwerk re-

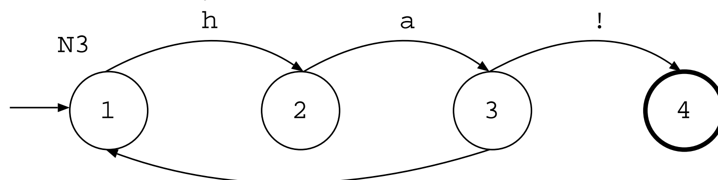


   präsentiert:

2. Das folgende Übergangsnetzwerk repräsentiert einen NEA, der dieselbe Sprache wie M_1 akzeptiert:



3. Nicht bei jedem Übergang von einer Konfiguration in eine andere muß ein Symbol der Eingabe verarbeitet werden. In Netzwerken werden Übergänge dieses Typs durch sogenannte „*jump*-Kanten“ repräsentiert. Im folgenden Netzwerk sind die Knoten 3 und 1 durch eine *jump*-Kante miteinander verbunden.



   Es gilt: $L(N_1) = L(N_2) = L(N_3)$;
   N_1 repräsentiert einen DEA, N_2 und N_3 NEAs.

7.3 Repräsentation endlicher Automaten in Prolog

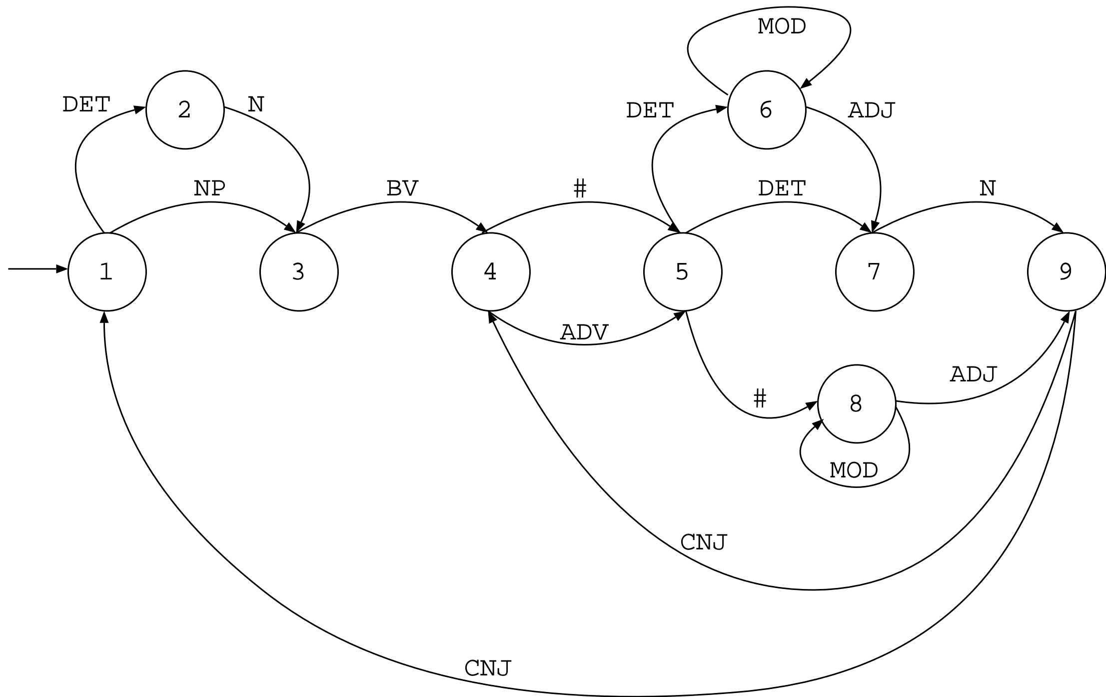
Bei der Implementierung von endlichen Automaten werden wir uns an ihrer Repräsentation als Übergangsnetzwerke orientieren.

Zur Repräsentation von Übergangsnetzwerken in Prolog verwenden wir das Prädikat *kanten*/3. Für jede Kante des Übergangsnetzwerkes wird ein Fakt der Form *kante(Startknoten, Endknoten, Label)* in die Wissensbasis eingetragen. Start- und Endknoten des Netzwerkes werden durch die Prädikate *startknoten*/1 und *endknoten*/1 abgebildet. Als Label bzw. Kantenetiketten werden nicht nur Symbole des Eingabevokabulars zugelassen, sondern auch Kategoriensymbole, die Klassen von Symbolen zusammenfassen. So ist es möglich, alle Kanten (außer *jump*-Kanten), die sich nur durch Etikett unterscheiden, durch eine Kante zu ersetzen. Wenn *Wf* ein Wort der Kategorie *Kat* ist, dann enthält die Wissensbasis einen Eintrag der Form: *wort(Wf, Kat)*.

Beispiel (7–3)
Das folgende Übergangsnetzwerk akzeptiert einfache englische Aussagesätze:

```
startknoten(1).          endknoten(9).

kante(1, 3, np).         kante(1, 2, det).        kante(2, 3, n).
kante(3, 4, bv).         kante(4, 5, adv).        kante(4, 5, '#').
kante(5, 6, det).        kante(5, 7, det).        kante(5, 8, '#').
kante(6, 7, adj).        kante(6, 6, mod).        kante(7, 9, n).
kante(8, 9, adj).        kante(8, 8, mod).        kante(9, 4, cnj).
kante(9, 1, cnj).

wort(np, kim).           wort(np, sandy).         wort(np, lee).
wort(det, a).            wort(det, the).          wort(det, her).
wort(n, consumer).       wort(n, man).            wort(n, woman).
wort(bv, is).            wort(bv, was).           wort(cnj, and).
wort(cnj, or).           wort(adj, happy).        wort(adj, stupid).
wort(mod, very).         wort(adv, often).        wort(adv, always).
wort(adv, sometimes).
```

Die Struktur des Übergangsnetzwerks ENGLISCH verdeutlicht die folgende Abbildung:



Das Netzwerk akzeptiert einfache prädikative Aussagesätze bzw. Konjunktionen solcher Aussagesätze, in denen die Objekt-NP eine beliebige Zahl von Gradadverbien enthalten kann oder aus einer Konjunktion von NPs besteht.

Beispiel (7–4)
Lee is very very stupid.
Kim was a consumer and Lee is always happy.
Her man is sometimes her man.

7.4 Recognizer, Generator und Parser

Wir werden nun eine Reihe von Prolog-Programmen entwickeln, die einen Recognizer, einen Generator und einen Parser für Übergangsnetzwerke realisieren.

7.4.1 Ein Recognizer für Übergangsnetzwerke

Eine Symbolfolge (hier repräsentiert durch eine Liste von Atomen) wird akzeptiert, gdw. es im Netzwerk einen Pfad gibt, der unter Verarbeitung der Symbolfolge von einem Startknoten zu einem Endknoten führt. Die Suche terminiert, sobald ein solcher Pfad gefunden wurde oder es keinen weiteren Pfad im Netzwerk mehr gibt, der überprüft werden könnte.

```
% recognize(?Knoten, ?Wortliste) :-
% es gibt einen Pfad von Knoten zu einem einen Endknoten
% repräsentierenden Knoten, so daß Wortliste mit Erreichen
% dieses Knotens vollständig abgearbeitet ist.

recognize(Knoten, [ ]) :-
    endknoten(Knoten).                    % Analye erfolgreich abgeschlossen.

recognize(Knoten_1, String) :-
    kante(Knoten_1, Knoten_2, Label),
    traverse(Label, String, NewString),
    recognize(Knoten_2, NewString).

traverse(Label, [Label|Worte], Worte) :-
    not(special(Label)).                  % Nächste Wort ist Etikett der Kante.

traverse(Label, [Wort|Worte], Worte) :-
    wort(Label, Wort).                    % Nächste Wort gehört zur Kategorie Label.

traverse('#', String, String).           % jump-Kante

special('#').                             % jump-Kante

special(Kategorie) :-
    wort(Kategorie, _).                   % Kategorien-Kante
```

Beispiel (7–5)
Abhängig davon, welche der Argumente des *recognize*/2-Prädikats instantiiert sind, kann man den Recognizer verwenden, um (a) zu testen, ob von einem Knoten aus eine bestimmte Symbolfolge verarbeitet werden kann; (b) welche Symbolfolge von einem Knoten aus akzeptiert wird; (c) welcher Knoten (als Startpunkt) die Verarbeitung einer Symbolfolge erlaubt und (d) welche Symbolfolgen von welchen Knoten aus akzeptiert werden:

```
?- recognize(1, [kim, is, very, stupid]).
   yes

?- recognize(1, X).
   X = [kim, is, often, a, happy, consumer] ;
   X = [kim, is, often, a, happy, consumer, and, a, happy, consumer]

?- recognize(X, [the, man, is, very, very, stupid]).
   X = 1

?- recognize(X, Y).
   X = 9
   Y = [ ];
   X = 1
   Y = [kim, is, often, a. happy, consumer]
```

7.4.2 Ein Generator für Übergangsnetzwerke

Um zu überprüfen, ob ein endlicher Automat sich so verhält, wie es die Entwickler wünschen, ist es sehr hilfreich, wenn man über ein Programm verfügt, das alle Wortfolgen generiert, die der Automat akzeptiert. Ein solches Programm realisiert einen **Generator** für die durch den Automaten festgelegte Sprache. Der folgende, sehr einfache Generator hat allerdings den Nachteil, daß er bei Netzwerken, die eine nicht-endliche Zahl von Sätzen akzeptieren (s.o.), nicht terminiert.

```
% Der Generator ruft das test/1-Prädikat auf, das einen Startknoten
% auswählt und das recognize-Prädikat verwendet, um einen Satz zu
% generieren, der von diesem Knoten ausgehend akzeptiert wird.
% Das Systemprädikat fail/0 erzwingt Backtracking und stellt so sicher,
% daß alle möglichen Lösungen generiert werden.

generate :-
    test(X),
    write(X), nl,
    fail.

test(Words) :-
    startknoten(Node),
    recognize(Node, Words).
```

7.4.3 Ein Parser für Übergangsnetzwerke

Der Parser unterscheidet sich vom Recognizer dadurch, das ein weiteres Argument eingeführt wird, daß mit dem Pfad (Format: Knoten, Etikett, Knoten, . . .) instantiiert wird, der eine vollständige Verarbeitung der Symbolfolge erlaubt.

```
parse(Knoten, [ ], [Knoten]) :-
    endknoten(Knoten).

parse(Knoten1, String, [Knoten1, Label|Pfad]) :-
    kante(Knoten1, Knoten2, Label),
    traverse(Label, String, NewString),
    parse(Knoten2, NewString, Pfad).
```

Beispiel (7–6)
Wie schon beim Recognizer ist es möglich, auch den Parser in verschiedener Weise zu verwenden:

```
?- parse(1, [kim, is, often, a, happy, consumer], [1, np, 3, bv, 4, adv, 5, det, 6, adj, 7, n, 9]).
```

yes

?- parse(1, [kim, is, often, a, happy, consumer], X).
X = [1, np, 3, bv, 4, adv, 5, det, 6, adj, 7, n, 9].

?- parse(1, X, Y).
X = [kim, is, often, a, happy, consumer]
Y = [1, np, 3, bv, 4, adv, 5, det, 6, adj, 7, n, 9].

7.5 Transducer

Ein Transducer ist ein Automat, der sich von einem einfachen Recognizer darin unterscheidet, daß er für jede akzeptierte Symbolfolge eine '*Übersetzung*' produziert und ausgibt.

Definition (7–4) *Endlicher Transducer*
Ein *endlicher Transducer* M = $<$Q, Σ, Φ, σ, q_0, F$>$ besteht aus:

1. einer endlichen Menge von Zuständen Q = $\{q_0, q_1, \ldots, q_n\}$;
2. einer endlichen Menge Σ = $\{v_1, \ldots, v_m\}$ von Eingabesymbolen;
3. einer endlichen Menge Φ = $\{w_1, \ldots, w_m\}$ von Ausgabesymbolen;
4. der Übergangsfunktion σ: $Q \times (\Sigma \cup \{e\}) \rightarrow Pot(Q \times \Phi^*)$;
5. dem Anfangszustand $q_0 \in Q$ und
6. einer Menge F (F ist eine Teilmenge von Q) von Endzuständen.

Definition (7–5) *Konfiguration*
Wenn M = $<$Q, Σ, Φ, σ, q_0, F$>$ ein endlicher Transducer ist, dann ist jedes $<q, v, w> \in Q \times \Sigma^* \times \Phi^*$ eine *Konfiguration* von M.

Beispiel (7–7)
Der folgende Transducer akzeptiert einfache englische Fragesätze und gibt ihre französische Übersetzung aus:

```
startknoten(1).              endknoten(5).

kante(1, 2, 'WHERE').        kante(2, 3, 'BE').        kante(3, 4, 'FDET').
kante(4, 5, 'FNOUN').        kante(3, 6, 'MDET').      kante(6, 5, 'MNOUN').

wort('WHERE', [where, ou]).                   wort('BE', [is, est]).
wort('FDET', [the, la]).                      wort('MDET', [the, le]).
wort('FNOUN', [exit, sortie]).                wort('FNOUN', [shop, boutique]).
wort('FNOUN', [toilet, toilette]).            wort('MNOUN', [policeman, gendarme]).
```

Das folgende Programm ermöglicht es, Transducer dieses Typs zur Übersetzung von Symbolfolgen zu verwenden. Der entscheidende Unterschied gegenüber der

Implementierung des Recognizers am Anfang des Kapitels liegt darin, daß das Prädikat *transduce* mit drei Argumenten aufgerufen wird: dem Startknoten, der Symbolfolge der Ausgangssprache und der Symbolfolge der Zielsprache.

```
% transduce(Knoten, Symbolfolge_1, Symbolfolge_ 2) :-
% Symbolfolge_2 ist eine Übersetzung von Symbolfolge_1, die man ausgehend von
% Knoten erhält.

transduce(Knoten, [ ], [ ]) :-
   endknoten(Knoten).

transduce(Knoten_1, String1, String2) :-
   kante(Knoten_1, Knoten_2, Label),
   traverse2(Label, String1, NewString1, String2, NewString2),
   transduce(Knoten_2, NewString1, NewString2).

traverse2([Wort1, Wort2], [Wort1|RestString1], RestString1, [Wort2|RestString2], RestStrin
   not(special(Wort1)), not(special(Wort2)).

traverse2(Abbrev, String1, NewString1, String2, NewString2) :-
   wort(Abbrev, NewLabel),
   traverse2(NewLabel, String1, NewString1, String2, NewString2).

traverse2(['#', Wort2], String1, String1, [Wort2|RestString2], RestString2).

traverse2([Wort1, '#'], [Wort1|RestString1], RestString1, String2, String2).

traverse2('#', String1, String1, String2, String2).


% Das test/0-Prädikat realisiert einen Generator für Transducer des oben
% beschriebenen Typs.

test :-
   write('Input: '),
   write([where, is, the, shop]), nl,
   write('Output: '),
   test([where, is, the, shop]),
   write('Input: '),
   write([where, is, the, policeman]), nl,
   write('Output: '),
   test([where, is, the, policeman]).

test(String_A) :-
   startknoten(Knoten),
   transduce(Knoten, String_A, String_B),
   write(String_B), nl.
```

Kapitel 8

Metalogische Prädikate

Als „metalogische Prädikate" werden diejenigen Prädikate bezeichnet, die sich nicht auf Konstrukte der Prädikatenlogik erster Stufe zurückführen lassen. Prädikate diesen Typs erlauben es,

- den Zustand des Beweises für ein gegebenes Ziel zu überprüfen;
- Variablen selbst als Objekte zu behandeln und
- Datenstrukturen in Ziele umzuwandeln.

Außerdem ermöglichen metalogische Prädikate es, zwei Restriktionen aufzuheben, auf die wir in Kapitel 6 gestoßen waren:

1. Viele Systemprädikate setzen voraus, daß bestimmte ihrer Argumente keine nicht-instantiierten Variablen sein dürfen (so z.B. arithmetische und Typprädikate). Daraus resultiert eine eingeschränkte Verwendbarkeit von Prologprogrammen, die auf solchen Prädikaten basieren, gegenüber ihnen korrespondierenden Logikprogrammen.

2. Bei der Analyse von Strukturen kann es geschehen, daß in diesen Strukturen vorkommende Variablen versehentlich instantiiert werden. Aus diesem Grund wurde bei der Definition der Prädikte *subterm*/2 und *substitute*/4 vorausgesetzt, daß ihre Argumente keine nicht-instantiierten Variablen enthalten.

8.1 Metalogische Typprädikate

Neben den Typenprädikaten (*atomic, atom, symbol, integer,* ..., etc), die wir in der letzten Sitzung kennengelernt haben, gibt es noch zwei metalogische Typenprädikate. Sie erlauben es zu überprüfen, ob es sich bei einem Term um eine nicht-instantiierte Variable handelt oder nicht: ein *var*/1-Ziel gelingt, wenn es sich bei dem Argument des Prädikats um eine nicht-instantiierte Variable handelt; ein *nonvar*/1-Ziel, wenn es keine nicht-instantiierte Variable ist.

Beispiel (8–1)

?- var(77).
no

?- var(hallo).
no

?- var([ ]).
no

?- nonvar(77).
yes

?- nonvar(hallo).
yes

?- nonvar([ ]).
yes

?- var(X).
yes

?- X is 2 + 3, var (X).
no

?- X = Y, var(Y).
X = _113
Y = _113
yes

?- nonvar(X).
no

?- X is 2 + 3, nonvar(X).
yes

```
?- X = Y, nonvar(Y).
   no
```

Metalogische Typenprädikate unterscheiden sich kategorial von anderen Typenprädikaten: Sie überprüfen nicht, ob ein Objekt zu einer bestimmten Objektklasse gehört, sondern ob ein sprachlicher Ausdruck von einem bestimmten Typ ist. Sie ermöglichen es, Prologprogramme zu schreiben, die ähnlich flexibel sind wie die ihnen korrespondierenden Logikprogramme:

Beispiel (8–2)
Ein einfaches arithmetische Logikprogramm, das die Additionsoperation abbildet, kann die folgende Form haben[1]:

```
% plus(X, Y, Z) ←
% X, Y und Z sind natürliche Zahlen und Z ist die Summe von X und Y.
plus(0, X, X) ←
   natuerliche_zahl(X).
plus(s(X), Y, s(Z)) ←
   plus(X, Y, Z).
natuerliche_zahl(0).
natuerliche_zahl(s(X)) :-
   natuerliche_zahl(X).
```

Das folgende Prologprogramm entspricht in seiner Flexibilität weitgehend diesem Logikprogramm:

```
plus(X, Y, Z) :-
   nonvar(X),
   nonvar(Y),
   Z is X + Y.
plus(X, Y, Z) :-
   nonvar(X),
   nonvar(Z),
   Y is Z - X.
plus(X, Y, Z) :-
   nonvar(Y),
   nonvar(Z),
   X is Z - Y.
```

Der einzige Unterschied zwischen beiden Programmen besteht darin, daß das Logikprogramm, nicht aber das Prologprogramm dazu verwendet werden kann, eine Zahl in zwei kleinere zu zerlegen (nichtinstantiierte Variablen als 1. und 2. Argument).

[1] Zahlen größer als 0 werden mit Hilfe der Nachfolgerfunktion gebildet, die durch die $s/1$-Struktur abgebildet wird (1 = s(0), 2 = s(s(0)), . . .)

Ziele in Klauseln, die aus metalogischen Prädikaten gebildet werden, bezeichnet man als *metalogische Tests* (s.o.). Diese Tests nehmen Bezug auf den aktuellen Stand des Beweises für das Ausgangsziel. Das folgende Beispiel zeigt, wie metalogische Typenprädikate verwendet werden können, um die Effizienz von Programmen zu verbessern.

Beispiel (8–3)
Das *grosseltern_teil*/2-Prädikat kann verwendet werden, um die Großeltern oder die Enkel einer Person zu bestimmen. Abhängig von der Verwendung des Prädikats läßt sich durch ein Vertauschen der Klauseln im Körper der Klauseln ein Höchtmaß an Effizienz bei der Suche nach einer Lösung garantieren:

```
grosseltern_teil(X, Z) :-
    nonvar(X),                  % Suche nach den Enkeln von X.
    elternteil(X, Y),
    elternteil(Y, Z).
grosseltern_teil(X, Z) :-
    nonvar(Z),                  % Suche nach den Großeltern von Z.
    elternteil(Y, Z),
    elternteil(X, Y).
```

Wie die metalogische Grundprädikate verwendet werden können, um auch komplexere metalogische Prädikate zu definieren, zeigt das folgende Beispiel.

Beispiel (8–4)
Das Prädikat *ground*/1 prüft, ob ein (komplexer) Term nicht-instantiierte Variablen enthält oder nicht:

```
ground(Term) :-
    nonvar(Term),                % 1. Fall: Atomarer Term.
    atomic(Term).
ground(Term) :-
    struct(Term),                % 2.Fall: Eine Struktur,
    functor(Term, F, N),         % deren Argumente keine Variablen enthalten.
    ground(N, Term).
ground(0, Term).
ground(N, Term) :-
    N > 0,
    arg(N, Term, Arg),
    ground(Arg),
    N1 is N-1,
    ground(N1, Term).
```

Eine weitere gute Möglichkeit zur Verwendung der metalogischen Prädikate bietet die Implementierung eines Unifikationsalgorithmus in Prolog. Natürlich ist es möglich, eine triviale Lösung zu wählen, indem man direkt den von Prolog selbst verwendeten Unifikationsalgorithmus nutzt: *unify(X, X).*

Diese Definition verwendet implizit das Systemprädikat =/2: ein *unify*-Ziel gelingt gdw. X = X. Das folgende Programm liefert eine nicht-triviale Lösung. Es reimplementiert den Prologunifikationsalgorithmus; d.h. auf einen Vorkommenstest wird verzichtet[2].

Beispiel (8–5)

```
% unify(Term1, Term2) :-
% Term1 und Term2 werden unter Vernachlässigung des Vorkommenstests
% miteinander unifiziert.
unify(X, Y) :-
    var(X),                      % 1. Fall:
    var(Y),                      % Zwei nicht-instantiierte Variablen.
    X = Y.
unify(X, Y) :-
    var(X),                      % 2. Fall:
    nonvar(Y),                   % 1. Term ist eine nicht instantiierte Variable.
```

[2]Wenn einer der beiden zu unifizierenden Terme eine nicht-instantiierte Variable ist, wird nicht geprüft, ob diese Variable im anderen Term vorkommt.

```
    X = Y.
unify(X, Y) :-
    var(Y),                 % 3. Fall:
    nonvar(X),              % 2. Term ist eine nicht instantiierte Variable.
    Y = X.
unify(X, Y) :-
    nonvar(X),              % 4. Fall:
    nonvar(Y),              % Zwei identische atomare Terme.
    atomic(X),
    atomic(Y),
    X = Y.
unify(X, Y) :-
    nonvar(X),              % 5. Fall:
    nonvar(Y),              % Zwei Strukturen - unifiziere ihre Argumente.
    struct(X),
    struct(Y),
    term_unify(X, Y).
term_unify(X, Y) :-
    functor(X, F, N),
    functor(Y, F, N),
    unify_args(N, X, Y).
unify_args(N, X, Y) :-
    N > 0,
    unify_arg(N, X, Y),
    N1 is N-1,
    unify_args(N1, X, Y).
unify_args(0, X, Y).
unify_arg(N, X, Y) :-
    arg(N, X, ArgX),
    arg(N, Y, ArgY),
    unify(ArgX, ArgY).
```

8.2 Der Vergleich offener Terme

Das Prädikat *unify1* bildet den von Prolog verwendeten Unifikationsalgorithmus nach; d.h. auf einen Vorkommenstest wird verzichtet. Um diesen Test zu realisieren, ist es notwendig zu testen, ob zwei Variablen identisch sind und nicht nur einfach unifizieren. Diese Art von Test ist (wen wundert es) ein metalogischer Test. Prolog stellt zu diesem Zweck zwei Systemprädikate zur Verfügung: ==/2 und \==/2. Ein Ziel der Form *T1 == T2* gelingt, wenn *T1* und *T2* identische Terme (Variablen, Konstanten oder Strukturen, die aus identischen Bestandteilen in gleicher Weise gebildet werden) sind; ein Ziel der Form *T1 \== T2* gelingt, wenn *T1* und *T2* unterschiedliche Terme sind.

Beispiel (8–6)
?- X = 5.
X = 5
yes
?- X == 5.
no
?- X \== 5.
yes
?- X = Y.
X = _122
Y = _122
yes
?- X == Y.
no
?- X \== Y.
yes
?- X = Y, X == Y.
X = _123
Y = _123
yes

Wir definieren jetzt eine Variante des *unify*/2 Prädikats, bei der der Vorkommenstest ausgeführt wird, wenn eine Variablen mit einer Struktur unifiziert wird.

Beispiel (8–7)
Mit Hilfe des Prädikats \== definieren wir ein weiteres metalogisches Prädikat *not_occurs_in*/2, das den Vorkommenstest realisiert.

```
% unify(Term1, Term2) :-
% Term1 und Term2 werden unter Berücksichtigung des Vorkommentests
% miteinander unifiziert.

% 1. Klausel (Unifikation zweier Variablen): s.o.
unify2(X, Y) :-
    var(X),
    nonvar(Y),
    not_occurs_in(X, Y),
    X = Y.
unify2(X, Y) :-
    var(Y),
    nonvar(X),
    not_occurs_in(Y, X),
    Y = X.
```

```
%4. + 5. Klausel (Unifikation von Atomen & Strukturen): s.o.
not_occurs_in(X, Y) :-
    var(Y),
    X \== Y.
not_occurs_in(X, Y) :-
    nonvar(Y),
    atomic(Y).
not_occurs_in(X, Y) :-
    nonvar(Y),
    struct(Y),
    functor(Y, F, N),
    not_occurs_in(N, X, Y).
not_occurs_in(0, X, Y).
not_occurs_in(N, X, Y) :-
    N > 0,
    arg(N, Y, Arg),
    not_occurs_in(X, Arg),
    N1 is N-1,
    not_occurs_in(N1, X, Y).
% Definition von term_unify: s.o.
```

Das *subterm*/2-Prädikat, das wir in der letzten Stunde definiert haben, arbeitet nur dann korrekt, wenn es auf geschlossene Terme angewendet wird. Eine Frage der Form *subterm(X, Y)?* gelingt, obwohl 'X' und 'Y' verschiedene Variablen bezeichnen. Ein einfacher metalogischer Test stellt sicher, daß auch offene Terme korrekt verarbeitet werden:

```
subterm*(X, Term) :-
    subterm(Sub, Term),
    X == Sub.
```

Das *subterm*-Ziel generiert via Backtracking alle Terme, die in dem Ausgangsterm enthalten sind, und das ==-Ziel überprüft die Identität des gesuchten Terms (X) mit einem Teilterm (Sub).

8.3 Metavariablen

In Prolog gibt es keine statische Grenze zwischen Daten und Programmen; d.h. es ist möglich, prinzipiell beliebige Datenstrukturen als Ziele zu verwenden. Um Datenstrukturen in Ziele zu verwandeln, gibt es ein Systemprädikat *call*/1, das einen Prolog-Term in ein zu beweisendes Ziel umwandelt. Anders als in Logikprogrammen, in denen es nicht zulässig ist, Variablen als Ziele zu verwenden, können in den meisten Prologimplementierungen auch Variablen als Ziele verwendet werden. In diesem Fall ist die Verwendung des *call*/1-Prädikats überflüssig. Wenn Variablen als Ziele verwendet werden, muß sichergestellt sein, daß

sie spätestens zu dem Zeitpunkt instantiiert sind, zu dem versucht wird, diese Ziele zu beweisen.

Beispiel (8–8)
Die logische Oder-Verknüpfung läßt sich wie folgt definieren:

```
% X oder Y :- X gilt oder Y gilt.
X oder Y :- X.                    % X ist wahr.
X oder Y :- Y.                    % Y ist wahr.
```

Damit dieses Programm korrekt arbeitet, muß *oder* als zweistelliger Infixoperator definiert werden. Diese Definition hat folgendes Format:

```
:- op(1100, xfy, 'oder').
```

Die Möglichkeiten, die Operatordefinitionen bieten, werden wir in einer der nächsten Stunden ausführlicher behandeln.

Kapitel 9

Kontrolliertes Backtracking

Wie wir gesehen haben, ist es einer der großen Vorzüge einer *deklarativen* Programmiersprache wie Prolog, daß man sich bei der Entwicklung von Programmen weniger um *prozedurale Aspekte* kümmern muß als in traditionellen, prozeduralen Programmiersprachen. In vielen Fällen ist das für Prolog charakteristische Lösungsverfahren (*depth-first* Suche mit Backtracking) ausreichend effizient und es ist nicht erforderlich, den Suchvorgang zu beeinflussen. Es gibt allerdings Situationen, in denen unkontrolliertes Backtracking äußerst ineffizient ist und sogar in extremen Fällen zu nicht-intendierten Ergebnissen führen kann. Wir werden uns in dieser Sitzung mit solchen Prädikaten befassen, die die Lösungssuche in Prolog beeinflussen: dem *Cut*–Operator und den Prädikaten *fail*/0 und *not*/1 bzw. \+.

9.1 Der Cut–Operator

Die zentrale Funktion des Cut–Operator liegt in der Reduzierung der Alternativen, die bei der Ausführung eines Prolog–Prädikats berücksichtigt werden müssen. Anders ausgedrückt: er begrenzt den Beweisbaum, der bei der Lösungssuche aufgebaut wird. Es werden zwei Verwendungsweisen des Cut–Operators unterschieden:

(a) In den Fällen, in denen er nicht die deklarative Bedeutung des Programms verändert – d.h. er den Suchbaum nur so begrenzt, daß ausschließlich solche Lösungswege, die zu keiner Lösung führen können, unterdrückt bzw. *abgeschnitten* werden – spricht man von „*green cuts*“ und

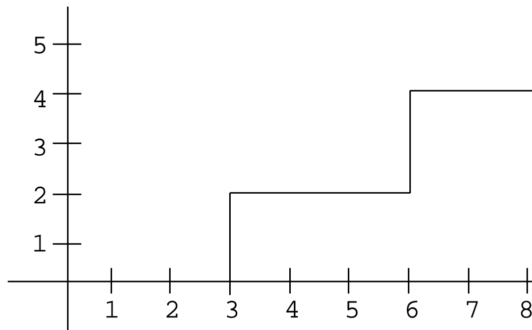
(b) Wenn die Verwendung des *Cuts* nicht nur die prozedurale, sondern auch die deklarative Bedeutung eines Programms ändert, spricht man von „*red cuts*“[1].

[1] Vgl. van Emden, M. (1982) Red and green cuts. *Logic Programming Newsletter*:**2**.

9.1.1 Grüne Cuts

Beispiel (9–1)
Betrachten wir zunächst eine Funktion f(x), deren Eigenschaften folgendermaßen beschrieben werden können:

1. $f(x) = 0$, wenn $x < 3$ ist;
2. $f(x) = 2$, wenn $3 \leq x < 6$ ist und
3. $f(x) = 4$, wenn $x \geq 6$.



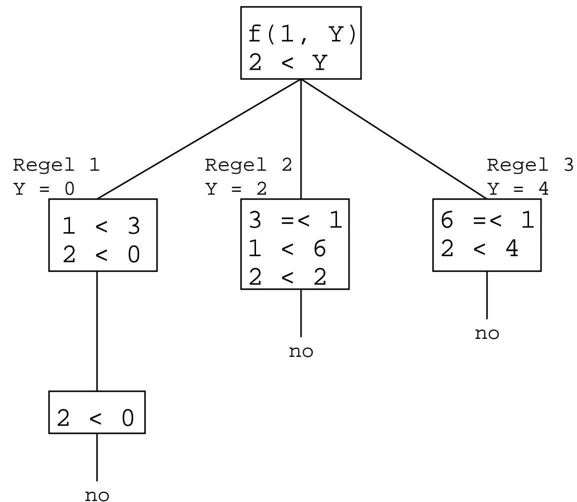
Diese Funktion läßt sich problemlos durch ein geeignetes, zweistelliges Prolog–Prädikat abbilden:

```
f(X, 0) :- X < 3.               % R1
f(X, 2) :- 3 =< X, X < 6.       % R2
f(X, 4) :- 6 =< X.              % R3
```

Diese Prozedur arbeitet korrekt, ist allerdings – wie wir sofort sehen werden – nicht besonders effizient. Das Verhalten von Prolog bei der Suche nach einer Antwort für die Frage

```
?- f(1, Y), 2 < Y.
```

läßt sich durch folgenden Suchbaum repräsentieren:



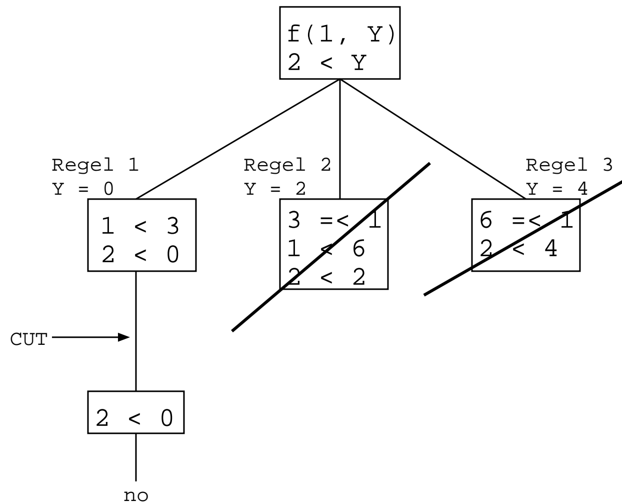
Die Ineffizienz beruht auf der Tatsache, daß in diesem Beispiel immer nur eine (genauer: höchstens eine) der Klauseln der Prozedur zum Erfolg führen kann; d.h. sobald der/die Tests in einer Klausel erfüllt sind, ist es überflüssig, noch die Anwendbarkeit einer anderen Klausel zu testen. Die Verwendung des Cut–Operators kann in diesem Fall den gewünschten Determinismus herstellen: Er verhindert, daß andere Klauseln des Prädikats berücksichtigt werden, nachdem der(die) Test(s) in einer Klausel erfüllt wurden.

```
f(X, 0) :- X < 3, !.              % R1'
f(X, 2) :- 3 =< X, X < 6, !.      % R2'
f(X, 4) :- 6 =< X, !.             % R3'
```

Unter Verwendung dieser Prozedur verhält sich Prolog nun bei der Frage

```
?- f(1, Y), 2 < Y.
```

wie gewünscht:



Die **operationale Wirkung** des Cut–Operators läßt sich folgendermaßen beschreiben:

Wenn

(a) $C = A$:- $B_1, \ldots, B_k$, !, $B_{k+1}, \ldots, B_n$
eine der Klauseln ist, die eine Prozedur **A** definieren;

(b) G ein Ziel ist, das mit dem Kopf der Klausel C unifiziert und

(c) die Ziele $B_1, \ldots, B_k$ bewiesen werden können

Dann gilt:

1. Die bei dem Beweis von $B_1, \ldots, B_k$ getroffenen Entscheidungen (Variableninstantiierungen) werden *eingefroren*; d.h. es gibt kein Backtracking innerhalb von $B_1, \ldots, B_k$.
2. Es wird keine andere Klausel für **A**, deren Kopf mit G unifizieren könnte, betrachtet.
3. Backtracking innerhalb von $B_{k+1}, \ldots, B_n$ ist möglich. Wenn allerdings ein B_r $(r > k)$ nicht beweisbar ist, geht das Backtracking höchstens bis zum Cut zurück.
4. Wenn beim Backtracking der Cut erreicht wird, scheitert er und damit auch die gesamte Klausel.

Grundregel für eine sinnvolle Verwendung des Cuts

Der Cut–Operator sollte immer dann verwendet werden, wenn innerhalb einer Prozedur sich wechselseitig ausschließende Fälle unterschieden werden. Die korrekte Plazierung des Cuts in den Klauseln der Prozedur führt dann zu dem beabsichtigten Determinismus[2].

[2] Implizit weist eine derartige Prozedur eine Struktur auf, wie sie in prozeduralen Sprachen z.B. durch die CASE ... OF–Anweisung erzeugt wird.

Beispiel (9–2)

Bubble–Sort

Fassung 1	Fassung 2
sort(Xs, Ys) :-	sort(Xs, Ys) :-
append(As, [X, Y|Bs], Xs),	append(As, [X, Y|Bs], Xs),
X > Y,	X > Y, **!**,
append(As, [Y, X|Bs], Xs1),	append(As, [Y, X|Bs], Xs1),
sort(Xs1, Ys).	sort(Xs1, Ys).
sort(Xs, Xs).	sort(Xs, Xs).

```
?- sort([3, 2, 1], X).
   X = [1, 2, 3]
   yes
```

Genau wie in Beispiel (9–1) hat auch hier der Cut die Funktion, unnötiges Backtracking zu unterdrücken: Für die Frage

?- sort([3, 2, 1], X)

kann Prolog, solange die erste Fassung der *sort*–Prozedur zugrundegelegt wird, mehrere (gleichlautende) Lösungen finden, die daraus resultieren, welche Elemente in welcher Reihenfolge vertauscht werden. So kann zunächst 3 mit 2 vertauscht werden oder 2 mit 1, usw. Da es aber nur eine korrekte Lösung gibt, ist es sinnlos, Backtracking zuzulassen, nachdem ein Paar von Zahlen gefunden wurde, die miteinander vertauscht werden müssen.

Zusammenzufassend läßt sich feststellen: In beiden der betrachteten Beispiele hat der Cut–Operator ausschließlich die Funktion, überflüssiges Backtracking zu vermeiden. Er beeinflußt nicht die deklarative Bedeutung des Programms, sondern nur seine Effizienz.

> **Grüne Cuts – operationaler Test**
> Um festzustellen, ob sich bei einem Cut innerhalb einer Prozedur um einen *grünen* Cut handelt oder nicht, entferne den Cut aus der Klausel und überprüfe, ob das Programm in allen Fällen dieselben Resultate liefert wie zuvor.

9.1.2 Rote Cuts

Es läßt sich sehr leicht zeigen, daß die Verwendung des Cut–Operators – anders als in den bislang betrachteten Beispielen – die deklarative Bedeutung einer Prozedur sehr wohl verändern kann.

Beispiel (9–3)

Das folgende einfache Prolog–Programm zeigt, wie ein Cut die deklarative Bedeutung eines Prädikates verändern kann:

(a) p :- a, b.
p :- c.
p ≡ (a ∧ b) ∨ c

(b) p :- a, !, b.
p :- c.
p ≡ (a ∧ b) ∨ (–a ∧ c)

An diesem Beispiel kann auch eine weitere Wirkung *roter* Cuts demonstriert werden: Wenn man in (a) die Reihenfolge der beiden Klauseln vertauscht, hat das wie in allen Programmen, die keine Cuts enthalten, keinen Einfluß auf die deklarative Bedeutung des Programms. Vertauscht man in (b) die beiden Klauseln, so erhält man:

p :- c.
p :- a, !, b.
p ≡ c v (a ∧ b)

Cuts, die die deklarative Bedeutung von Prozeduren verändern, können auf dreierlei Weise eingeführt werden:

1. *implizit:*
   durch die Verwendung spezieller Systemprädikate wie *not*/1 und '\=' (s. nächster Abschnitt);

2. *unbeabsichtigt:*
   (a) die deklarative Wirkung der eingeführten Cuts wird übersehen oder
   (b) durch das Weglassen von Bedingungen, die nach der Einführung des Cuts in Klauseln überflüssig werden, können aus *grünen* Cuts *rote* Cuts werden und

3. *gezielt:*
   die deklarative Wirkung der eingeführten Cuts ist beabsichtigt und gewollt.

Weglassen überflüssiger Bedingungen

In der zweiten Fassung von Beispiel (9–1) ist in der zweiten Klausel (R2') die Bedingung *3 =< X* überflüssig; denn wenn die erste Regel nicht anwendbar ist, ist diese Bedingung auf jeden Fall erfüllt. Gleiches gilt auch für die Bedingung in der dritten Klausel (R3'). Wir können also eine dritte Variante des Programms entwickeln:

```
f(X, 0) :- X < 3, !.          % R1”
f(X, 2) :- X < 6, !.          % R2”
f(X, 4).                      % R3”
```

Diese Prozedur generiert dieselben Ergebnisse wie das ursprüngliche Programm. Allerdings wurden durch das Weglassen der Bedingungen aus den *grünen rote* Cuts: Wenn wir beide Cuts entfernen, liefert die Prozedur bei erzwungenem Backtracking mehrere Lösungen, von denen einige – wie soll es bei einer Funktion anders sein – falsch sind. So gilt z.B.:

```
?- f(1, Y).
        Y = 0;
        Y = 2;
        Y = 4;
  no
```

Gezielte Verwendung von roten Cuts

Unter besonderen Umständen kann sich gezielte Verwendung von 'roten' Cuts als sinnvoll erweisen: Als wir uns mit Listen und Prädikten zur Listenverarbeitung auseinandergesetzt haben, entwickelten wir ein Prädikat *member*, das überprüfen sollte, ob ein bestimmtes Objekt in einer Liste enthalten ist. Die von uns entwickelte Version des Prädikats fand nicht nur das erste, sondern alle Vorkommen dieses Objekts. Ein richtig plazierter Cut stoppt das Backtracking, nachdem das erste Vorkommen des Objekts gefunden wurde:

```
member1(X, [X|Rest]) :- !.
member1(X, [Y|Rest]) :-
    member1(X, Rest).
```

Es ist allerdings offensichtlich, daß der Cut die deklarative Bedeutung des Prädikats geändert hat: *member1* findet nur das erste Vorkommen eines Objekts und kann auch nicht mehr wie *member* verwendet werden, um alle Elemente einer Liste aufzuzählen.

Grundregel zur Verwendung von roten Cuts

Mit äußerster Vorsicht und Sparsamkeit einsetzen!

9.2 Negation in Prolog

Bislang besaßen wir nur die sprachlichen Mittel um auszudrücken, daß eine Folge von Bedingungen erfüllt sein muß, um ein gegebenes Ziel zu beweisen. Fälle, in denen zu fordern ist, daß eine oder mehrere Bedingungen nicht erfüllbar sein dürfen, konnten nicht adäquat behandelt werden.

Beispiel (9–4)

Angenommen, Mary ißt gerne Obst, nur Bananen mag sie nicht. Den ersten Teil dieses Sachverhalts kann man in Prolog problemlos durch folgende Regel repräsentieren:

```
essen(mary, X) :- obst(X).
```

Um den zweiten Teil des Sachverhalts zu repräsentieren, fügen wir eine weitere Klausel hinzu, die sicherstellt, daß das Prädikat auf kein Objekt zutrifft, bei dem es sich um eine Banane handelt. Zu diesem Zweck eignet sich das Prolog–Prädikat *fail*, das immer scheitert und so auch zum Scheitern der Klausel führt, in der es vorkommt. Wir erhalten so folgende Prozedur:

```
essen(mary, X) :-                    (R1)
        banane(X), !, fail.
essen(mary, X) :-                    (R2)
        obst(X).
```

Der Cut in R1 verhindert Backtracking, und *fail* führt dazu, daß die Klausel scheitert, wenn *banane(X)* erfüllt ist.

Die an diesem Beispiel demonstrierte Cut/fail–Kombination läßt sich relativ universell anwenden. So kann mit ihr z.B. das built–in Prädikat '\=' rekonstruiert werden, das auf zwei Terme genau dann zutrifft, wenn diese nicht miteinander unifizierbar sind:

```
nicht–identisch(X, X) :- !, fail.      (R1)
nicht–identisch(X, Y).                 (R2)
```

Klauseln des Typs R2 nennt man *catch–all* Klauseln, da sie – wenn Prolog bei der Lösungssuche auf sie stößt – immer gelingen. Unter Verwendung der Cut/fail–Kombination läßt sich auch das einstellige Prädikat *not* definieren, das in den meisten Prolog–Versionen vordefiniert ist:

```
not(P) :-
    P, !, fail.
not(P).
```

Wie man leicht sieht, führt die Verwendung des Prädikats *not* implizit zur Einführung von roten Cuts und ist aus diesem Grund mit Vorsicht zu verwenden.

Unter Verwendung des *not*–Prädikats lassen sich die Prädikate aus den letzten beiden Beispielen reformulieren als:

```
essen(mary, X) :-
   obst(X),
   not banane(X).
nicht–identisch(X, Y) :-
   not(X = Y).
```

Das Prolog–Prädikat *not* entspricht nicht vollständig der logischen Negation. Die Ursache dafür ist Prologs unvollständige Realisierung des Beweisverfahrens für Logikprogramme: Negation in Logikprogrammen ist zu definieren über einen endlichen Suchbaum, dessen Terminalknoten alle mit *no* etikettiert sind. Das Beweisverfahren von Prolog stellt nicht sicher, daß ein solcher Beweisbaum gefunden wird, selbst wenn er existiert.

Die Unterschiede zwischen not und der logischen Negation verdeutlichen die folgenden Beispiele:

Beispiel (9–5)
Die Frage *?- not human(mary).* wird mit *yes* beantwortet, sofern keine anderslautenden Informationen in der Datenbasis enthalten sind.

Diese Antwort ist nicht so zu verstehen, daß Prolog bewiesen hat, daß Mary kein Mensch ist, sondern sie besagt nur, daß Prolog nicht genug Informationen vorliegen, um das Gegenteil zu beweisen. Wie die Definition des not–Prädikats zeigt, versucht Prolog nicht ein not–Ziel direkt zu beweisen, sondern es versucht gerade das nicht–negierte Ziel zu beweisen. Gelingt dieser Beweis nicht, antwortet Prolog mit *yes*. Dieses Vorgehen basiert auf der sogen. *closed–world assumption*; d.h. auf der Annahme, daß alle relevanten Informationen über die Welt in der Datenbasis präsent sind; d.h. daß die Prämissenmenge vollständig ist.

Beispiel (9–6)
Gegeben sei eine Datenbasis mit folgenden Klauseln:

```
r(a).
q(b).
p(X) :- not r(X).
```

Dann gilt:

```
?- q(X), p(X).
   X = b
   yes
?- p(X), q(X).
   no
```

Zu diesen unterschiedlichen Ergebnissen kommt es, da im ersten Fall X instantiiert ist wenn p(X) ausgeführt wird und im zweiten Fall nicht. Die zweite Frage wird reduziert zu:

```
r(X), !, fail, q(X).
```

Der Versuch, dieses (komplexe) Ziel zu beweisen, muß scheitern.

Kapitel 10

Ein- und Ausgabe

Die Mensch–Maschine–Kommunikation, wie wir sie bislang kennengelernt und praktiziert haben, unterlag zahlreichen Restriktionen. Sie beschränkte sich darauf, Prolog–Programme zu laden bzw. mit dem Editor zu bearbeiten und am Terminal zu beobachten, wie Prolog Fragen durch die Instantiierung von Variablen beantwortete. Gegenstand dieses Kapitels bilden die wichtigsten Systemprädikate, die die Ausführung von Ein- und Ausgabeoperationen steuern. Diese Operationen dienen dazu

- Daten aus Dateien zu lesen bzw. in Dateien zu schreiben;
- Prolog–Programme in die Wissensbasis zu laden;
- Daten formatiert auszugeben und
- Atome zu zerlegen bzw. zu bilden.

10.1 Dateien

In Prolog arbeiten alle Ein-/Ausgabeoperationen auf *Streams*. Ein Stream ist ein Objekt, das als Produzent/Konsument von Daten dient. Seine primäre Funktion liegt darin, eine Verbindung zwischen den Ein-/Ausgabeoperationen innerhalb des Prolog-Systems und den externen Ein-/Ausgabegeräten (Terminal, Drucker, ...) und Dateien herzustellen.

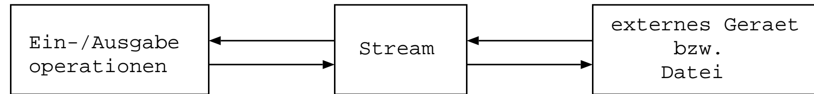


Abbildung (10-1)

Die Dateien, aus denen Prolog Daten liest bzw. in die Prolog Daten schreibt, werden als *Input–Streams* bzw. *Output–Streams* bezeichnet. Grundsätzlich können in Prolog mehrere Streams zum Lesen/Schreiben geöffnet sein. Aber während der Ausführung eines Programms können zu jedem Zeitpunkt immer nur zwei Streams **aktiv** sein: der eine zum Lesen und der andere zum Schreiben. Diese beiden Streams werden als *aktueller Input–Stream* und als *aktueller Output–Stream* bezeichnet. Alle Ein– und Ausgabeoperationen beziehen sich immer auf den aktuellen Input– bzw. Output–Stream.

Wie auch in anderen Programmiersprachen, die das Stream–Konzept verwenden, wird auch in Prolog das Terminal in dieses Konzept integriert: das Eingabegerät des Terminals (normalerweise die Tastatur) ist mit einem *Input–Stream*, das Ausgabegerät (Bildschirm) mit einem *Output–Stream* verknüpft. Diese beiden Pseudodateien werden mit dem Namen *user* bezeichnet und bilden die Standardeingabe– und die Standardausgabedatei; d.h. nach dem Start des Prologsystems werden zunächst Eingaben von der Tastatur erwartet und alle Ausgaben erfolgen auf dem Bildschirm.

Im Zusammenhang mit der Verwaltung von *Input–* und *Output–Streams* gibt es in Prolog sechs wichtige Systemprädikate: *see*/1, *seeing*/1, *seen*/0, *tell*/1, *telling*/1, *told*/0.

see(Datei)	Die Datei DATEI wird zum aktuellen *Input–Stream*. Existiert eine Datei mit diesem Namen nicht, antwortet Prolog mit *no*.
seeing(Datei)	DATEI wird mit dem Namen des aktuellen *Input–Streams* instantiiert.
seen	Der aktuelle *Input–Stream* wird geschlossen, und *user* wird zum aktuellen *Input–Stream*.
tell(Datei)	Die Datei DATEI wird zum aktuellen *Output–Stream*. Existiert bereits eine Datei dieses Namens, wird sie überschrieben.
telling(Datei)	DATEI wird mit dem Namen des aktuellen *Output–Streams* instantiiert.
told	Der aktuelle *Output–Stream* wird geschlossen, und *user* wird zum aktuellen *Output–Stream*.

Beispiel (10–1)

Angenommen es sollen Daten aus einer Datei *datei1* gelesen und verarbeitet werden und anschließend bestimmte Informationen in eine Datei *datei2* geschrieben werden, dann kann diese Aufgabe wie folgt gelöst werden:

(1) see(datei1), (2) see(datei1),

lese(Daten),	*lese*(Daten),
verarbeite(Daten),	*verarbeite*(Daten),
see(user),	**seen**,
tell(datei2),	tell(datei2),
schreibe(Daten),	*schreibe*(Daten),
tell(user).	**told**.

> Der Unterschied zwischen beiden Programmen besteht darin, daß nach Ausführung des ersten Programms die beiden Dateien *datei1* und *datei2* geöffnet bleiben, während sie bei der Ausführung des zweiten Programms durch *seen* bzw. *told* geschlossen werden.

Ein weiteres, in vielen Fällen nützliches Prädikat, ist das *files*/0 Prädikat. Die Frage

```
?- files.
```

führt dazu, daß alle geöffneten Dateien und ihr Status (Typ des ihnen zugeordneten Streams) angezeigt werden.

10.1.1 Dateiorganisation

Bei Prologdateien handelt es sich ihrer Organisationsform nach ausschließlich um *sequentielle Dateien*: Die Daten einer Datei lassen sich nur in der Reihenfolge lesen, in der sie in der Datei gespeichert sind. Beim öffnen einer Datei wird ein Zeiger (Dateizeiger) auf das erste Element der Datei gesetzt. Bei jedem Lesevorgang wird der Dateizeiger um eine Position verschoben. Wenn das Dateiende erreicht ist, liefert eine Leseoperation das Atom *end_ of_ file* zurück. Es ist nicht möglich, bereits gelesene Informationen erneut zu lesen, außer durch Schließen und erneutes öffnen der Datei.

Wenn der aktuelle *Output–Stream* mit einer Datei assoziiert ist, dann führt jede Schreibanweisung dazu, daß die Information an das Ende der Datei angehängt wird. Geschriebene Informationen können nicht selektiv überschrieben werden. Wird eine bestehende Datei zum Schreiben geöffnet, wird sie vollständig überschrieben.

Ihrem Inhalt nach handelt es sich bei allen Prologdateien um *Textdateien*; d.h. die Dateien enthalten Buchstaben, Ziffern und Sonderzeichen. Zu den Sonderzeichen gehören auch die sogenannten *non–printable* Zeichen, die nicht selbst auf dem Bildschirm erscheinen, aber z.B. die Formatierung der Ausgabe beeinflussen (Zeilenumbruch, Leerzeichen, ...).

Wenn eine Datei ausschließlich aus Ausdrücken besteht, bei denen es sich syntaktisch betrachtet um Prolog–Terme handelt, dann kann diese Datei mit den Prädikaten *read*/1 und *write*/1 bearbeitet werden, die einen Term aus einer Datei lesen bzw. in eine Datei schreiben. Anderenfalls müssen die Prädikate *get*/1, *get0*/1 und *put*/1 verwendet werden, die zeichenorientiert arbeiten.

10.1.2 Programm–Dateien

Für einen effizienten Umgang mit Programmdateien gibt es in Prolog die beiden Systemprädikate *consult*/1 und *reconsult*/1:

Ein Ziel der Form *consult(datei)* bewirkt, daß die Datei geöffnet wird und die in ihr enthaltenen Klauseln in die Wissensbasis geladen werden. Wenn eine Klausel K für ein Prädikat P eingelesen wird, dann wird K direkt hinter den bereits für P eingelesenen Klauseln plaziert. Für das *consult*–Prädikat gibt es eine Kurzschreibweise, die wir bereits kennengelernt haben: Statt *consult(datei1)* kann man auch einfach *[datei1]* schreiben; sollen mehrere Dateien geladen werden, schreibt man kurz: *[datei1, datei2, . . .]*.

Ein Ziel der Form *reconsult(datei)* bewirkt, daß die Datei geöffnet wird und die in ihr enthaltenen Klauseln in die Wissensbasis geladen werden. Der Unterschied zu dem *consult*–Prädikat liegt darin, daß in diesem Fall die alten, bereits in der Wissensbasis enthaltenen Definitionen der in der Datei definierten Prozeduren aus der Wissensbasis gelöscht werden. Wie bei *consult*/1 gibt es auch für dieses Prädikat eine Kurzschreibweise: *[–datei]* bzw. *[–datei1, –datei2, . . .]*.

Beispiel (10–2)

1. Angenommen, die Wissensbasis enthält drei Klauseln für ein Prädikat P und die Datei *noch_einmal* enthält ebenfalls diese drei Klauseln:

   Wissensbasis

⋮
K 1
K 2
K 3
⋮

   noch_einmal

K1, K2, K3

   Nachdem die Datei konsultiert wurde (*?- [noch_einmal]*), hat die Wissensbasis die Form:

⋮
K 1
K 2
K 3
K 1
K 2
K 3
⋮

   Verwendet man stattdessen *reconsult*/1, ändert sich der Inhalt der datei nicht.

2. Angenommen, die Wissensbasis enthält wie im ersten Fall drei Klauseln für ein Prädikat P und die Datei *neue_klauseln* drei Klauseln von P, die sich von denen in der Wissensbasis unterscheiden (K 1', K 2', K 3'). Nachdem die Datei konsultiert wurde (*?- [neue_klauseln]*), hat die Wissensbasis die Form:

⋮
K 1
K 2
K 3
K 1'
K 2'
K 3'
⋮

   Verwendet man stattdessen *reconsult*/1, erhält man folgendes Ergebnis:

$\vdots$
K 1'
K 2'
K 3'
$\vdots$

Eine Datei kann neben Klauseln auch Anweisungen enthalten. Eine Anweisung beginnt mit dem Operator :-, auf den dann ein Ziel oder mehrere Ziele folgen. Die in einem Programm enthaltenen Anweisungen werden ausgeführt, sobald das Programm mit *consult*/1 oder *reconsult*/1 in die Wissensbasis geladen wird.

Beispiel (10–3)
Wenn die Datei *datei1* die Anweisungen
:- consult(datei2), consult(datei3). bzw.
:- [datei2, datei3] enthält,
dann werden beim Konsultieren von *datei1* auch die Dateien *datei2* und *datei3* konsultiert.

10.2 Verarbeitung von Termen

Um Terme aus einem Stream zu lesen bzw. in einen Stream zu schreiben, gibt es in Prolog die beiden Systemprädikate *read*/1 und *write*/1:

Ein Ziel der Form *read(X)* führt dazu, daß der nächste Term aus dem aktuellen Input–Stream gelesen wird und die nicht–instantiierte Variable X mit diesem Term instantiiert wird. Dem Term müssen ein Punkt und ein Zeilenvorschub (*carriage–return*) folgen. Wenn das Dateiende erreicht ist, wird X mit dem Atom *end_ of_ file* instantiiert. Ein Ziel der Form *write(Term)* führt dazu, daß Term in den aktuellen Output–Stream geschrieben wird. Bei *read*–Zielen und *write*–Zielen ist kein Backtracking möglich.

Weitere viel verwendete Prädikte sind *tab*/1 und *nl*/0. Ein Ziel der Form *tab(n)* gelingt immer und hat den Seiteneffekt, daß n Leerzeichen in den aktuellen Output–Stream geschrieben werden; *nl* erzwingt einen Zeilenvorschub.

Beispiel (10–4)
Die folgende Klausel definiert ein Prädikat *cube*/2, das das Kubik einer Zahl berechnet:

```
cube(N, C) :-
    C is N * N * N.
```

Mit den oben eingeführten Prädikaten können wir eine Prozedur definieren, die solange auf Eingaben wartet und diese Berechnung durchführt, bis das Atom *stop* eingegeben wird:

```
cube :-
    read(X),
    process(X).
process(stop) :- !.                    % Berechnung terminiert.
```

```
process(N) :-
    C is N * N * N,                    % Berechnung des Wertes.
    write(C), nl,                      % Ausgabe des Wertes.
    cube.                              % Rekursiver Aufruf des Hauptprädikats.
```

Bei Verwendung von Systemprädikaten, die Ein–/Ausgabeoperationen realisieren, ist zu beachten (s.o.), daß sie kein Backtracking erlauben. Es ist z.B. nicht möglich, das Programm aus dem Beispiel (10-4) in folgender Weise zu verbessern:

```
cube :-
    read(stop), !.
cube :-
    read(X),
    C is X * X * X,
    write(C),
    cube.
```

Wenn z.B. *7* eingegeben wird, scheitert das Ziel *read(stop)*, und die zweite *cube*–Klausel wird geprüft: Ein neues *read*–Ziel wird ausgeführt und damit eine neue Eingabe erwartet; d.h. die *7* ist gewissermaßen verlorengegangen.

10.2.1 Formatierung von Termen

Da man in Prolog häufig mit Listen arbeitet und große Listen leicht sehr unübersichtlich wirken, ist es sinnvoll, sich Prädikate zu definieren, die den Inhalt von Listen formatiert ausgeben:

Beispiel (10–5)
Wir definieren nun drei Prädikate, die die Elemente einer Liste Zeile für Zeile in den aktuellen Output–Stream schreiben:

1. Das Prädikat *writelist*/1 gibt alle Top–level–Elemente einer Liste aus:

```
writelist([ ]).
writelist([X|L]) :-
    write(X), nl
    writelist(L).
```

2. Das Prädikat *writelist2*/1 gibt alle Elemente einer Liste formatiert aus:

```
writelist2([ ]).
writelist2([X|L]) :-
    doline(X), nl
    writelist2(L).
doline([ ]).
doline([X|L]) :-
    write(X), tab(1),
    doline(L).
```

?- writelist2([[a, b, c], [d, e, f], [g, h, i]]).
a b c
d e f
g h i

3. Wenn eine Liste ausschließlich aus positiven Zahlen besteht, läßt sie sich sehr einfach 'graphisch' repräsentieren:

```
bar([N|L]) :-
    stars(N), nl
    bar(L).

stars(N) :-
    N > 0,
    write(*),
    N1 is N - 1,
    stars(N1).

stars(N).

?- bar([3, 4, 6, 5].
    ***
    ****
    ******
    *****
```

Natürlich können mit den in diesem Abschnitt eingeführten Prädikaten nicht nur Listen, sondern auch Terme in verschiedenster Art und Weise formatiert werden.

10.2.2 Verarbeitung von Dateien von Termen

Wenn man alle Terme einer Datei *test* verarbeiten will, läßt sich häufig folgendes einfaches Schema verwenden:

... see(test), verarbeite_datei(test), seen, ...

Die Prozedur *verarbeite_datei*/1 liest nacheinander die Terme der Datei *test* und führt eine Reihe von Operationen auf ihnen aus; d.h. dieser Prozedur liegt folgendes Schema zugrunde:

```
verarbeite_datei :-
    read(Term),
    verarbeite(Term).

verarbeite(end_of_file) :- !.

verarbeite(Term) :-
    bearbeite(Term),
    verarbeite_datei.
```

wobei *bearbeite*/1 die Prozedur ist, die die auf alle Terme von *test* anzuwendenden Operationen zusammenfaßt.

10.3 Verarbeitung von Zeichen

Wenn Dateien nicht aus Prolog–Termen bestehen, dann kann ihr Inhalt nur zeichenweise gelesen/ergänzt werden. In Prolog gibt es drei Systemprädikate, um einzelne Zeichen aus einem Stream zu lesen bzw. in einen Stream zu schreiben:

- *put*/1
  Ein Ziel der Form *put(C)*, wobei C mit einer Zahl zwischen 0 und 127 instantiiert sein muß, führt dazu, daß das Zeichen mit diesem ASCII–Kode in den aktuellen Output–Stream ausgegeben wird.

- *get0*/1
  Ein Ziel der Form *get0(C)* führt dazu, daß das nächste Zeichen aus dem aktuellen Input–Stream gelesen wird und C mit dem ASCII–Kode dieses Zeichens instantiiert wird.

- *get*/1
  Dieses Prädikat arbeitet wie *get0*, außer das alle *non–printable* Zeichen (z.B. Leerzeichen) überlesen werden; d.h. es werden solange Zeichen aus dem aktuellen Input–Stream gelesen, bis ein 'normales' Zeichen gelesen wurde.

Beispiel (10–6)
Die Prozedur *squeeze*/0 liest einen Satz aus dem aktuellen Input–Stream und schreibt ihn formatiert in den aktuellen Output–Stream. *Formatiert* bedeutet in diesem Zusammenhang, daß alle Wörter durch genau ein Leerzeichen voneinander getrennt sind. Es wird vorausgesetzt, daß der Satz keine anderen Satzzeichen außer dem Punkt am Satzende enthält.

```
squeeze :-
    get0(C),
    put(C),
    dorest(C).

dorest(46) :- !.                % Satzende erreicht.

dorest(32) :-                   % Leerzeichen gelesen.
    !, get(C),
    put(C),
    dorest(C).

dorest(L) :-                    % Alphanumerisches Zeichen gelesen.
    squeeze.

?- squeeze.
   Der   alte  Mann           fiel    ins Meer.
   Der alte Mann fiel ins Meer.
```

10.4 Zerlegen und Aufbauen von Atomen

Wenn man zeichenweise aus einem Stream liest bzw. in ihn schreibt, dann ist es häufig hilfreich, über ein Prädikat zu verfügen, das aus einer Liste von ASCII–Kodes ein Atom generiert bzw. ein Atom in eine Liste von ASCII–Kodes überführt. Dieses Prädikat heißt *name*/2:

```
?- name(X, [65, 66, 67]).
   X = ABC

?- name(zx80, Y).
   Y = [122, 120, 56, 48]
```

Beispiel (10–7)
Die Prozedur *read_sentence*/1 liest einen Satz ein und bildet eine Liste, die die Worte dieses Satzes enthält:

```
?- read_sentence(X).
   Ein kleiner Hund suchte einen Knochen.[Return]
   X = ['Ein', kleiner, 'Hund', suchte, einen, 'Knochen']
   yes
```

```
% Das Prädikat read_sentence/1 liest das nächste Zeichen aus dem aktuellen Input–Stream
% und übergibt es an die Prozedur read_words, die die weitere Verarbeitung des Satzes
% steuert.

read_sentence(Words) :-
    get0(FirstChar),
    read_words(FirstChar, Words).

% Das read_words/2 Prädikat unterscheidet drei Fälle:
% 1. das nächste Zeichen ist ein Zeichen, das zur Bildung eines Wortes verwendet werden
% kann;
% 2. das nächste Zeichen ist ein Leerzeichen oder
% 3. das nächste Zeichen ist das Satzendezeichen.

read_words(Char, [Word|Words]) :-
    word_char(Char),             % Normales Zeichen: Wortende noch nicht erreicht.
    read_word(Char, Word, NextChar),
    read_words(NextChar, Words).
read_words(Char, Words) :-
    fill_char(Char),             % Leerzeichen: Wort–, aber nicht Satzende erreicht.
    get0(NextChar),
    read_words(NextChar, Words).
read_words(Char, [ ]) :-
```

```
    end_of_words_char(Char).       % Satzende erreicht.

% Das Prädikat read_word/3 liest ein einzelnes Wort ein.
read_word(Char, Word, NextChar) :-
    word_chars(Char, Chars, NextChar),
    name(Word, Chars).

word_chars(Char, [Char|Chars], FinalChar) :-
    word_char(Char), !,            % Wortende noch nicht erreicht:
    get0(NextChar),                % Weiterlesen.
    word_chars(NextChar, Chars, FinalChar).
word_chars(Char, [ ], Char) :-
    not word_char(Char).           % Wortende erreicht.

 word_char(C) :-
    97 =< C,
    C =< 122.                      % Kleinbuchstaben
word_char(C) :-
    65 =< C,
    C =< 90.                       % Großbuchstaben
word_char(95).                     % Underscore

fill_char(32).                     % Leerzeichen

end_of_words_char(46).             % Punkt
```

Kapitel 11

Weitere Systemprädikate

Wir werden in diesem Kapitel eine Reihe von Prädikaten einführen, die es erlauben, den Inhalt der Wissensbasis zu verändern. Außerdem werden wir die Möglichkeiten iterativer Programmierung in Prolog betrachten und abschließend einige Systemprädikate vorstellen, die alle Lösungen für ein Ziel liefern.

11.1 Datenbasismanipulation

Prolog erlaubt es Programme zu schreiben, die sich während ihrer Ausführung selbst modifizieren: Da es in Prolog keine prinzipielle Trennung von Daten und Prozeduren gibt – beide Typen von Informationen werden in der Prolog–Wissensbasis gespeichert – sind zu diesem Zweck nur Prädikate erforderlich, die die Wissensbasis durch gezieltes Löschen bzw. Hinzufügen von Klausen verändern können.

Das Prädikat *clause*/2 ermöglicht den Zugriff auf Prädikate bzw. Klauseln eines Prädikates. Ein Ziel der Form *clause(Kopf, Körper)* führt dazu, daß in der Wissensbasis nach einer Klausel gesucht wird, deren Kopf mit *Kopf* unifiziert. Der Körper dieser Klausel wird dann mit *Körper* unifiziert. Backtracking ist möglich und erzwingt die Fortsetzung der Suche als erstes Argument eines *clause*–Ziels darf keine nicht–instantiierte Variable verwendet werden.

Beispiel (11–1)
Gegeben sei eine Wissensbasis, die die folgenden beiden Klauseln enthält:

```
anzahl([ ], 0).          anzahl([_|Xs], N) :-
                              anzahl(Xs, N1),
                              N is N1 + 1.
```

?- clause(anzahl(X, Y), Z).
X = [ ]
Y = 0

```
  Z = true ;
  X = [_139|_140]
  Y = _141
  Z = anzahl(_140, _142), _141 is _142 + 1
  yes
?- clause(anzahl(_, _), true).
  yes
?- clause(anzahl(_, _), X), compound(X).
  X = anzahl(_141, _141), _141 is _142 + 1
  yes
```

Die Prädikate *asserta*/1, *assertz*/1, *retract*/1 und *abolish*/2 können verwendet werden, um den Inhalt der Wissensbasis zu verändern:

- *asserta(Klausel)*
  Wenn die Wissensbasis Klauseln enthält, die eine Prozedur P definieren und der Kopf von *Klausel* ein Term ist, der denselben Funktor und dieselbe Stelligkeit besitzt wie diese Klauseln, dann wird *Klausel* an den Anfang der Definition von P in die Wissensbasis geschrieben[1].

- *assertz(Klausel)*
  Der Unterschied zu einem entsprechenden *asserta*–Ziel besteht darin, daß *Klausel* am Ende der Definition von P in die Wissensbasis eingefügt wird.

- *retract(Klausel)*
  Die erste Klausel der Wissensbasis, die mit *Klausel* unifiziert, wird aus der Wissensbasis entfernt. *Klausel* darf keine nicht–instantiierte Variable sein. Durch Backtracking können mehrere/alle Klauseln entfernt werden, die diese Bedingung erfüllen.

- *abolish(Funktor, Stelligkeit)*
  Im Gegensatz zu einem *retract*–Ziel werden alle Klauseln aus der Wissensbasis entfernt, deren Kopf aus einem mit *Funktor* gebildeten Term der angegebenen Stelligkeit besteht.

Beispiel (11–2)
Gegeben sei eine Wissensbasis, die die beiden Fakten *intelligent(maria)* und *intelligent(peter)* enthält:

```
?- intelligent(arnold).
  no
?- asserta(intelligent(arnold)), assertz(intelligent(silvia)).
  yes
?- intelligent(X)
```

[1] Gibt es in der Wissenbasis keine derartige Klauseln, wird *Klausel* am Anfang der Wissensbasis eingefügt (bzw. bei *assertz* am Ende).

```
    X = arnold ;
    X = maria ;
    X = peter ;
    X = silvia
?- clause(intelligent(X), true), assertz(intelligent(X)),
   retract(intelligent(Y)),retract(intelligent(Z)).
    X = arnold
    Y = arnold
    Z = maria
    yes
?- intelligent(X)
    X = peter ;
    X = silvia ;
    X = arnold
?- abolish(intelligent,1).
    yes
?- listing(intelligent).
    yes
```

Eine andere Situation, in der diese Prädikate sinnvoll verwendet werden können, liegt dann vor, wenn komplexe Berechnungen auszuführen sind uns sichergestellt werden sollte, daß Teilberechnungen nicht mehrfach ausgeführt werden. In diesem Fall können Zwischenergebnisse als Fakten in der Wissensbasis abgelegt werden und am Ende wieder entfernt werden.

Beispiel (11–3)
Das folgende Programm berechnet das 1×1 und legt das Ergebnis in Form von Fakten des Typs *produkt(Zahl1, Zahl2, Ergebnis)* in der Wissensbasis ab:

```
maketable :-
    L = [1, 2, ..., 9],
    member(X, L),
    member(Y, L),
    Z is X * Y,
    assertz(produkt(X,Y,Z)),
    fail.
maketable.
```

11.2 Iterative Programmierung in Prolog

Wir haben bislang bei der Entwicklung von Prolog–Programmen ausschließlich rekursive Prozeduren verwendet. In einer Reihe von Fällen haben wir allerdings mit Hilfe des *fail*–Prädikats Prädikate definiert, deren Verhalten dem von iterativen Prozeduren in prozeduralen Programmiersprachen ähnelt: Das *fail* am

Ende der 1. Klausel der Definition des *maketable*–Prädikats aus dem letzten Beispiel erzwingt Backtracking und sorgt dafür, daß die vor ihm stehenden Ziele solange ausgeführt werden, bis die Tabelle vollständig berechnet ist.

Konstruktionen dieses Typs werden als *failure–driven loops* bezeichnet. Das Systemprädikat *repeat*/0 ermöglicht in Kombination mit *fail*, Schleifen zu realisieren, die den *Repeat*–Schleifen prozeduraler Programmiersprachen entsprechen. Ein *repeat*–Ziel gelingt immer. Jedesmal wenn es beim Backtracking erreicht wird, sorgt es dafür, daß ein neuer Pfad im Beweisbaum verfolgt wird. Man kann es sich definiert vorstellen als:

```
repeat.
repeat :- repeat.
```

Beispiel (11–4)
Das Prädikat *echo*/0 liest alle Terme des aktuellen Input–Streams und schreibt sie in den aktuellen Output–Stream:

```
echo :-
    repeat,
    read(X),                    % Lies Term aus Input–Stream.
    echo(X), !.

echo(end_of_file) :- !.         % Dateiende erreicht.

echo(X) :-
    write(X),                   % Schreibe Term in den Output–Stream.
    nl, fail.                   % Erzwinge Backtracking.
```

Auch das Systemprädikat *consult*/1 kann durch eine *repeat*–Schleife reimplementiert werden:

```
consult(File) :-
    see(File),                  % Öffne Datei zum Lesen.
    consult_loop,
    seen.                       % Schließe Datei wieder.

consult_loop :-
    repeat,
    read(Klausel),              % Lies nächste Klausel.
    process(Klausel), !.

process(end_of_file) :- !.      % Dateiende erreicht.
process(Klausel) :-
    assertz(Klausel),           % Schreibe Klausel in die Wissensbasis.
    fail.                       % Erzwinge Backtracking.
```

11.3 Ein Prädikat für alle Lösungen

In Prolog ist es möglich, durch Backtracking alle Lösungen für ein Ziel zu generieren. Es gibt aber Situationen, in denen es vorzuziehen ist, alle Lösungen direkt zur Verfügung zu haben. Zu diesem Zweck gibt es in Prolog die Systemprädikate *bagof*/3, *setof*/3 und *findall*/3.

Alle drei Prädikate haben die Form *prädikat(+Muster, +Ziel, –Ergebnisliste)*, wobei *Ergebnisliste* alle Instantiierung von *Muster* enthält, die *Ziel* erfüllen. Der Unterschied zwischen *bagof* und *setof* besteht darin, daß *setof* Mehrfachvorkommen eines Wertes entfernt. Anders als *findall* erlauben es *bagof* und *setof*, bei der Generierung der Lösungen andere Variablen in *Ziel* mit zu berücksichtigen.

Beispiel (11–5)

Angenommen die Wissensbasis enthält die folgenden sechs Fakten:

test(a, a). test(a, b). test(a, c).
test(a, b). test(b, b). test(b, d).

dann ist folgender Dialog möglich:

?- setof(X, test(X, Y), L).
X = _139
Y = a
L = [a] ;
X = _139
Y = b
L = [a, b] ;
X = _139
Y = c
L = [a] ;
X = _139
Y = d
L = [b] ;
no
?- findall(X, test(X, Y), L).
X = _139
Y = _140
L = [a, a, a, b, a, b] ;
no
?- setof(X, Y^test(X, Y), L).
X = _139
Y = _140
L = [a, b]
?- bagof(X, Y^test(X, Y), L).
X = _139
Y = _140
L = [a, a, a, b, a, b]

11.4 Interaktive Programme

In diesem letzten Abschnitt beschreiben wir zwei größere Programme, die ein gewisses Maß an Interaktion mit dem Benutzer erfordern: Das erste Prorgamm realisiert einen kleinen zeilenorientierten Editor; das zweite eine Prolog–*Shell*, die es ermöglicht, den mit dem System geführten Dialog in einer Datei zu protokollieren.

11.4.1 Ein zeilenorientierter Editor

Zur Repräsentation der Datei wird eine Struktur *file(Oben, Unten)* verwendet, mit *Oben* als Liste der Zeilen vor der aktuellen Cursorposition und *Unten* als

Liste der Zeilen, die auf die aktuelle Cursorposition folgen. Der Cursor steht also immer *zwischen* zwei Zeilen.

```
% Das Prädikat edit/0 initilisiert die Datei und startet durch den Aufruf von Edit/1 die
% Dateibearbeitung.

edit :- edit(file([ ], [ ])).
edit(File) :-
    write_prompt,                    % Ausgeben des Editor–Prompts.
    read(Kommando),                  % Lesen des nächsten Kommandos.
    edit(File, Kommando).            % Ausführen des Kommandos.

% edit/1 unterscheidet drei Fälle: Das Kommando exit führt zum Verlassen des Editors;
% andere Kommandos werden von dem Kommando–Interpreter apply/3 ausgeführt und un-
% bekannte Anweisungen führen zu einer Fehlermeldung.

edit(File, exit) :- !.               % Verlassen des Editors.
edit(File, Command) :-
    apply(Command, File, File1),     % Ausführen des Kommandos.
    !, edit(File1).                  % Edieren der veränderten Datei.
edit(File, Command) :-
    writeln([Command, ' is not applicable']),
    edit(File).                      % Unbekannte Kommandos werden ignoriert.

% Der Kommandointerpreter apply/3 führt die Kommandos zur Bearbeitung der Datei (Cur-
% sor eine Zeile hoch/runter, Zeile löschen, . . . ) aus.

apply(up, file([X|Xs], Ys), file(Xs, [X|Ys])).
apply(up, file([ ], Ys), file([ ], Ys)).
apply(up(N), file(Xs, Ys), file(Xs1, Ys1)) :-
    N > 0,
    up(N, Xs, Ys, Xs1, Ys1).
apply(down, file(Xs, [Y|Ys]), file([Y|Xs], Ys)).
apply(down, file(Xs, [ ]), file(Xs, [ ])).
apply(insert(Line), file(Xs, Ys), file(Xs, [Line|Ys])).
apply(delete, file(Xs, [Y|Ys]), file(Xs, Ys)).
apply(delete, file(Xs, [ ]), file(Xs, [ ])).
apply(print, file([X|Xs], Ys), file([X|Xs], Ys)) :-
    write(X), nl.
apply(print, file([ ], Ys), file([ ], Ys)) :-
    write('<<top>>'), nl.
apply(print(*), file(Xs, Ys), file(Xs, Ys)) :-
    reverse(Xs, Xs1),
    write_file(Xs1),
    write_file(Ys).
```

```
%
%

up(N, [ ], Ys, [ ], Ys).
up(0, Xs, Ys, Xs, Ys).
up(N, [X|Xs], Ys, Xs1, Ys1) :-
    N > 0,
    N1 is N-1,
    up(N1, Xs, [X|Ys], Xs1, Ys1).

write_file([X|Xs]) :-
    write(X), nl,
    write_file(Xs).
write_file([ ]).

write_prompt :-
    write('>>'), nl.

writeln([ ]) :-                        % Ausgabe einer Liste von Termen.
    nl.
writeln([X|Xs]) :-
    write(X),
    writeln(Xs).
```

11.4.2 Eine interaktive Shell mit Protokoll

Version 1

```
shell :-
    shell_prompt,
    read(Goal),
    shell(Goal).

shell(exit) :- !.
shell(Goal) :-
    ground(Goal), !,
    shell_solve_ground(Goal),
    shell.
shell(Goal) :-
    shell_solve(Goal),
    shell.

shell_solve(Goal) :-
    Goal,
    write(Goal), nl.
shell_solve(Goal) :-
    write('No (more) solutions'), nl.

shell_solve_ground(Goal) :-
```

```
        Goal, !,
        write('Yes'), nl.
    shell_solve_ground(Goal) :-
        write('No'), nl.

    shell_prompt :-
        write('Next command? ').
```

Version 2

```
    logg :-
        shell_flag(logg).

    shell_flag(Flag) :-
        shell_prompt,
        shell_read(Goal, Flag),
        shell(Goal, Flag).

shell(exit, Flag) :-
        !, close_logging_file.
shell(nolog, Flag) :-
        !, shell_flag(nolog).
shell(logg, Flag) :-
        !, shell_flag(logg).
shell(Goal, Flag) :-
        ground(Goal), !,
        shell_solve_ground(Goal, Flag),
        shell_flag(Flag).
shell(Goal, Flag) :-
        shell_solve(Goal, Flag),
        shell_flag(Flag).

shell_solve(Goal, Flag) :-
        Goal,
        flag_write(Goal, Flag), nl.
shell_solve(Goal, Flag) :-
        flag_write('No (more) solutions', Flag), nl.

shell_solve_ground(Goal, Flag) :-
        Goal, !,
        flag_write('Yes', Flag), nl.
shell_solve_ground(Goal, Flag) :-
        flag_write('No', Flag), nl.

shell_prompt :-
        write('Next command? ').

shell_read(X, logg) :-
        read(X),
        file_write(['Next command? ', X], 'prolog.log').
shell_read(X, nolog) :-
```

```
        read(X).
flag_write(X, nolog) :-
        write(X).
flag_write(X, logg) :-
        write(X),
        file_write([X], 'prolog.log').
file_write(X, File) :-
        telling(Old),
        tell(File),
        writeln(X),
        tell(Old).
close_logging_file :-
        tell('prolog.log'),
        told.
```

Kapitel 12

Operatoren und Differenzlisten

In den bisherigen Sitzungen haben wir ausschließlich Programme entwickelt, die auf *vollständigen* Datenstrukturen operierten. In diesem Kapitel wird beschrieben, wie durch die Verwendung unvollständiger Datenstrukturen effiziente Programme entwickelt werden können. Die wichtigste unvollständige Datenstruktur für Prolog-Programme korrespondiert mit der Datenstruktur Liste und wird als *Differenzliste* bezeichnet.

12.1 Operatoren

Bislang haben wir es ausschließlich mit vordefinierten Operatoren zu tun gehabt, wie z.B. den arithmetischen Operatoren '+', '–', '∗', ... und den Systemoperatoren ':-', '?-', ',', ';' usw. Prolog bietet dem Benutzer die Möglichkeit, eigene Operatoren zu definieren.

Ein wichtiges Argument für die Einführung benutzerdefinierter Operatoren ist die Tatsache, daß sie in vielen Fällen die Lesbarkeit und Übersichtlichkeit von Programmen erheblich verbessern. Zu beachten ist allerdings, daß mit selbstdefinierten Operatoren keine spezifischen Operationen verbunden sind, etwa wie die Operation der Addition mit dem '+'–Operator. Sie sind nur ein gestalterisches Mittel, das es erlaubt, die Syntax von Ausdrücken den eigenen Bedürfnissen anzupassen.

Zur Definition von Operatoren wird das *built–in* Prädikat *op* verwendet, das folgendes Format besitzt:

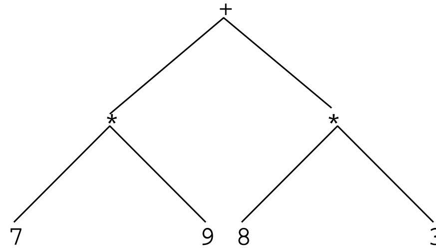
op(<*Vorrang*>, <*Assoziativität*>, <*Operator*>).

12.1.1 Vorrang

Wir haben gesehen, daß arithmetische Ausdrücke in Prolog nicht in der sonst für Prolog charakteristischen Präfixnotation angegeben werden müssen, sondern

als Infix–Ausdrücke notiert werden können:

Beispiel (12–1)
$+ (* (7, 9), * (8, 3)) \Longleftrightarrow 7 * 9 + 8 * 3$



Intern werden arithmetische Ausdrücke allerdings immer in Präfixnotation repräsentiert. Die Frage, die in diesem Zusammenhang naheliegt, ist, wie der Prologinterpreter wissen kann, wie ein (prinzipiell ambiger) Infixausdruck zu interpretieren ist. Die Lösung dieses Problems ist sehr einfach: Für jeden Operator wird angegeben, wie stark er bindet. Seine *Bindungsstärke* wird als sein *Vorrang* (engl. precedence) gegenüber anderen Operatoren bezeichnet.

In unserem Beispiel gilt, daß die Multiplikation stärker bindet als die Addition und damit die Ambiguität der Infixnotation aufgelöst wird. In Prolog wird der Vorrang eines Operators durch einen numeraischen Wert zwischen 1 und 1200 festgelegt. Gundsätzlich gilt: Je höher dieser Wert ist, um so geringer ist die Bindungsstärke des Operators. Um den Vorrang der Multiplikation gegenüber der Addition sicherzustellen, könnte z.B. '+' der Vorrang 500 und '∗' der Vorrang 300 zugewiesen werden.

12.1.2 Assoziativität

Das zweite Argument in einer Operatordefinition legt fest, um was für einen Typ von Operator es sich handelt und bestimmt die Assoziativität des Operators: Es legt fest, in welcher Reihenfolge Argumente in Ausdrücken verarbeitet werden, die mehrere Vorkommen eines Operators enthalten. Folgende sieben Operatortypen werden unterschieden:

Symbol	Position	Assoziativität
xfx	Infix	nicht assoziativ
xfy	Infix	rechts nach links
yfx	Infix	links nach rechts
fx	Präfix	nicht assoziativ
fy	Präfix	links nach rechts
xf	Postfix	nicht assoziativ
yf	Postfix	rechts nach links

Bei einer Typangabe legt f die Position des Operators und x bzw. y die der Argumente fest, wobei gilt, daß

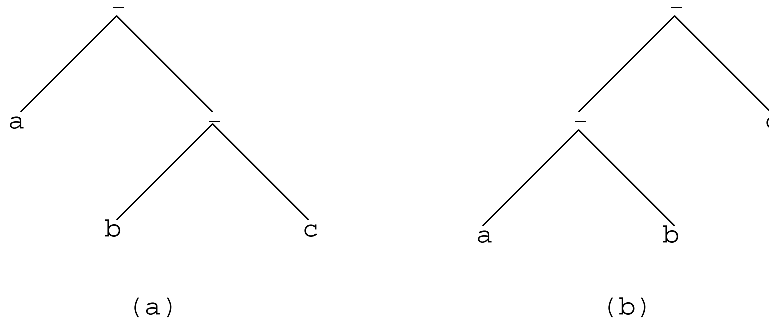
- an der durch x markierten Position nur ein Ausdruck auftreten darf, der einen geringeren Vorrang als der durch f bezeichnete Operator besitzt;

- an der durch *y* markierten Position dagegen darf auch ein Ausdruck mit gleichem oder geringerem Vorrang stehen.

Der Vorrang eines beliebigen Objektes läßt sich durch folgende Regel bestimmen:

1. Ein unstrukturiertes Objekt hat den Vorrang 0.
2. Eine in runden Klammern eingeschlossene Struktur hat den Vorrang 0.
3. Für jede andere Struktur wird der Vorrang durch den ihres Hauptoperators bestimmt.

Beispiel (12–2)
Gegeben sei der arithmetische Ausdruck a – b – c. Dieser Ausdruck ist zunächst prinzipiell ambig:



Wenn allerdings der Operator '–' als Operator des Typs *xfy* definiert wird, dann bleibt nur die Lesart (a) übrig.

Beispiel (12–3)
Wenn '∼' als ein Operator vom Typ *fx* definiert wird, dann ist der Ausdruck $\sim p$ ein syntaktisch zulässig, nicht aber der Ausdruck $\sim\ \sim p$. Um Ausdrücke dieser Form zuzulassen, muß '∼' als Operator vom Typ *fy* definiert werden.

Beispiel (12–4)
Gegeben seien folgende Operatordefinitionen:

:- op(300, xfx, plays).
:- op(200, xfy, and).

dann führen die Terme

?- Term = jimmy plays football and squash.
?- Term = susan plays tennis and basketball and volleyball.

zu folgenden Resultaten:

Term = plays(jimmy, and(football, squash)).
Term = plays(susan, and(tennis, and(basketball, volleyball))).

12.2 Differenzlisten

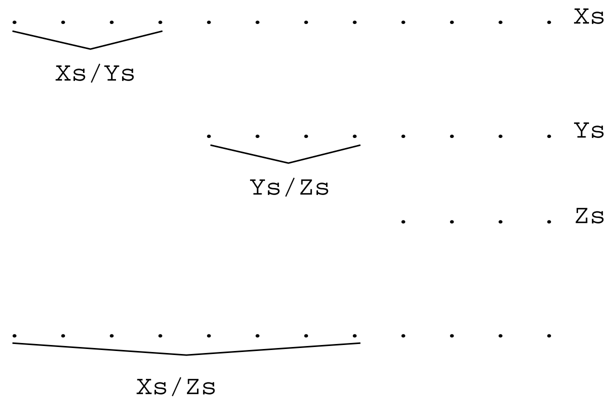
Die Folge von Objekten 1, 2, 3 kann (u.a.) entweder durch eine Liste [1, 2, 3] oder als Differenz zweier Listen repräsentiert werden:

[1, 2, 3, 4, 5] - [4, 5] bzw.
[1, 2, 3, 8] - [8] ...

Wir werden im folgenden zur Bezeichnung von Differenzlisten den Operator „/" verwenden; d.h. *Xs/Ys* bezeichnet eine Differenzliste mit dem Kopf *Xs* und dem Rest *Ys*. [1, 2, 3|Xs]/Xs ist die allgemeinste Differenzliste, die zur Repräsentation der Zahlenfolge 1, 2, 3 verwendet werden kann.

Zwischen Listen und Differenzlisten besteht ein enger systematischer Zusammenhang: Jede Liste L kann als eine Differenzliste der Form L/[ ] aufgefaßt werden. Der Vorteil von Differenzlisten besteht darin, daß Operationen auf ihnen wie z.B. *append* via Unifikation in konstanter Zeit und nicht wie bei einfachen Listen in linearer Zeit (abhängig von der Länge der Liste) realisiert werden können.

Beispiel (12–5)



Hinreichende und notwendige Bedingung für die Verknüpfung zweier Differenzlisten ist die Unifizierbarkeit des Rests der ersten mit dem Kopf der zweiten Liste; ist diese Bedingung erfüllt, sind beide Listen *kompatibel*. Natürlich gilt: Eine Differenzliste mit einer nicht-instantiierten Variable als Rest ist mit jeder anderen Differenzliste kompatibel.

Beispiel (12–6)
Wir definieren zunächst die *Append*-Relation unter Verwendung von Differenzlisten und zeigen anschließend, wie sie zum Glätten von Listen verwendet werden können:

% Verknüpfen von Differenzlisten
% append_dl(As, Bs, Cs) ←

```
% Die Differenzliste Cs ist das Ergebnis der Verkettung der beiden kompatiblen
% Differenzlisten As und Bs.

append_dl(Xs/Ys, Ys/Zs, Xs/Zs).

% Glätten von Differenzlisten
% flatten(Xs, Ys) ←
% Ys ist eine flache Liste, die alle Elemente aus Xs enthält.

flatten(Xs, Ys) :-
    flatten_dl(Xs, Ys/[ ]).
```

```
flatten_dl([ ], Xs/Xs).
flatten_dl(X, [X|Xs]/Xs) :-
    atomic(X),
    X \== [ ].
flatten_dl([X|Xs], Ys/Zs) :-
    flatten_dl(X, As/Bs),
    flatten_dl(Xs, Cs/Ds),
    append_dl(As/Bs, Cs/Ds, Ys/Zs)
```

Die letzte Klausel kann vereinfacht werden, da As/Bs nur dann mit Cs/Ds verknüpft werden kann, wenn Bs und Cs miteinander unifizierbar sind. Wir erhalten so:

```
flatten_dl([X|Xs], Ys/Zs) :-
    flatten_dl(X, Ys/Ys1),
    flatten_dl(Xs, Ys1/Zs).
```

Der folgende Trace verdeutlicht die Arbeitsweise von *flatten_dl*/2:

```
flatten([[a], [b, [c]]], Xs)
  flatten_dl([[a], [b, [c]]], Xs/[ ])
    flatten_dl([a], Xs/Xs1)
      flatten_dl(a, Xs/Xs2)                       Xs = [a|Xs2]
        atomic(a)
        a \== [ ]
      flatten_dl([ ], Xs2/Xs1)                    Xs2 = Xs1
    flatten_dl([[b, [c]]], Xs1/[ ])
      flatten_dl([b, [c]], Xs1/Xs3)
        flatten_dl(b, Xs1/Xs4)                    Xs1 = [b|Xs4]
          atomic(b)
          b \== [ ]
        flatten_dl([[c]], Xs4/Xs3)
          flatten_dl([c], Xs4/Xs5)
            flatten_dl(c, Xs4/Xs6)                Xs4 = [c|Xs6]
              atomic(c)
              c \== [ ]
            flatten_dl([ ], Xs6/Xs5)              Xs6 = Xs5
          flatten_dl([ ], Xs5/Xs3)                Xs5 = Xs3
      flatten_dl([ ], Xs3/[ ])                    Xs3 = [ ]
Xs = [a, b, c]
```

Programme, die Differenzlisten verwenden, ähneln stark *endrekursiven* Programmen. Charakteristisch für endrekursive Programme ist, daß bei ihnen der zu berechnende Wert mit Ende des rekursiven Abstiegs vorliegt; d.h. während des rekursiven Aufstiegs keine diesen Wert verändernden Operationen mehr ausgeführt werden. Programme dieses Typs verwenden meistens einen sogenannten *Akkumulator*.

Beispiel (12–7)
Die endrekursive Variante des *flatten*-Prädikats hat folgende Form:

```
flatten(Xs, Ys) :-
    flatten(Xs, [ ], Ys).
flatten([X|Xs], Zs, Ys) :-
    flatten(Xs, Zs, Ys1),
    flatten(X, Ys1, Ys).
flatten(X, Xs, [X|Xs]) :-
    atomic(X),
    X \== [ ].
flatten([ ], Xs, Xs).
```

Es gibt nur zwei Unterschiede zwischen diesem Programm und der Variante, die Differenzlisten verwendet:

1. Im endrekursiven Programm wird die Differenzliste durch zwei Argumente repräsentiert;

2. *flatten*/3 berechnet die Resultatsliste *bottom-up* (d.h. beginnend mit dem Rest der Liste); *flatten_dl*/3 dagegen berechnet sie *top-down*.

Weitere Beispiele unterstreichen die Ähnlichkeit, die zwischen endrekursiven und Differenzlistenprogrammen besteht:

Beispiel (12–8)

```
% Umkehren von Differenzlisten
% reverse(Xs,Ys) ← Ys ist die Umkehrung der Liste Xs.
reverse(Xs, Ys) :-
    reverse_dl(Xs,Ys/[ ]).
reverse_dl([ ],Xs/Xs).
reverse_dl([X|Xs],Ys/Zs) :-
    reverse_dl(Xs,Ys/[X|Zs]).

% Quicksort mit Differenzlisten
% quicksort(List,Sortedlist) :- Sortedlist ist eine geordnete Permutation von List.
quicksort(Xs,Ys):-
    quicksort_dl(Xs,Ys/[ ]).
quicksort_dl([ ],Xs/Xs).
quicksort_dl([X|Xs],Ys/Zs) :-
    partition(Xs,X,Littles,Bigs),
    quicksort_dl(Littles,Ys/[X|Ys1]),
```

```
    quicksort_dl(Bigs,Ys1/Zs).
partition([ ],Y,[ ],[ ]).
partition([X|Xs],Y,[X|Ls],Bs) :-
    X =< Y,!,
    partition(Xs,Y,Ls,Bs).
partition([X|Xs],Y,Ls,[X|Bs]) :-
    X > Y,!,
    partition(Xs,Y,Ls,Bs).
```

Es stellt sich natürlich die Frage, welche Probleme Lösungen mit Hilfe von Differenzlisten nahelegen: Gute Kandidaten bilden birekursive Programme, bei denen am Ende beide Teilergebnisse durch *append* miteinander verknüpft werden.

Beispiel (12–9)
Ein gutes Beispiel für ein Problem dieses Typs bildet ein vereinfachte Version von Dijkstra's *dutch flag Problem*. Dabei geht es darum, eine Liste von roten, weißen und blauen Objekten so zu ordnen, daß alle roten vor den weißen Objekten und die weißen vor den blauen Objekten stehen. Wenn also die Liste die Form *[red(1), white(2), blue(3), red(4), white(5)]* hat, sollte man die Liste *[red(1), red(4), white(2), white(5), blue(3)]* erhalten.

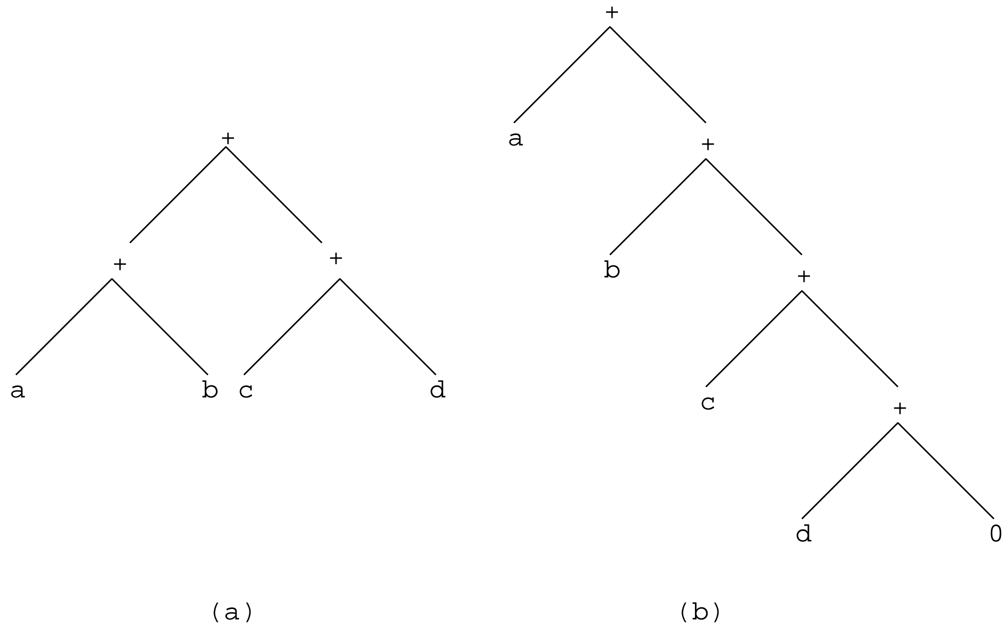
```
% dutch(Xs, RedsWhitesBlues) :-
% RedsWhitesBlues ist eine Liste von Elementen von Xs, die nach Farbe geordnet wurden:
% rot, dann weiß, dann blau.

% Version 1 - ohne Differenzlisten

dutch1(Xs, RedsWhitesBlues) :-
    distribute(Xs, Reds, Whites, Blues),
    append(Whites, Blues, WhitesBlues),
    append(Reds, WhitesBlues, RedsWhitesBlues).

% distribute(Xs, Reds, Whites, Blues) :-
% Reds, Whites, and Blues sind Listen von roten, weißen und blauen Elementen aus Xs.
distribute([red(X)|Xs], [red(X)|Reds], Whites, Blues) :-
    distribute(Xs, Reds, Whites, Blues).
distribute([white(X)|Xs], Reds, [white(X)|Whites], Blues) :-
    distribute(Xs, Reds, Whites, Blues).
distribute([blue(X)|Xs], Reds, Whites, [blue(X)|Blues]) :-
    distribute(Xs, Reds, Whites, Blues).
distribute([ ], [ ], [ ], [ ]).

% Version 2 - mit Differenzlisten

dutch2(Xs, RedsWhitesBlues) :-
    distribute_dls(Xs, RedsWhitesBlues/WhitesBlues,
```

```
        WhitesBlues/Blues, Blues/[ ]).
    % distribute_dls(Xs, Reds, Whites, Blues) :-
    % Reds, Whites, Blues sind Differenzlisten von roten, weißen und blauen Elementen aus Xs
distribute_dls([ ], Reds/Reds, Whites/Whites, Blues/Blues).
distribute_dls([red(X)|Xs], [red(X)|Reds]/Reds1, Whites, Blues) :-
    distribute_dls(Xs, Reds/Reds1, Whites, Blues).
distribute_dls([white(X)|Xs], Reds, [white(X)|Whites]/Whites1, Blues) :-
    distribute_dls(Xs, Reds, Whites/Whites1, Blues).
distribute_dls([blue(X)|Xs], Reds, Whites, [blue(X)|Blues]/Blues1) :-
    distribute_dls(Xs,Reds, Whites, Blues/Blues1).
```

12.3 Differenz-Strukturen

Auch zu anderen vollständigen Datenstrukturen als Listen können korrespondierende unvollständige Datenstrukturen angegeben werden. So erhält man als Gegenstück zu den Prolog-Strukturen *Differenzstrukturen*.

Beispiel (12–10)
Ein gutes Beispiel, an dem sich die Verwendung von Differenzstrukturen einfach demonstrieren läßt, bildet die *Normalisierung* arithmetischer Ausdrücke. Die folgende Abbildung zeigt einen arithmetischen Ausdruck in nicht-normalisierter und normalisierter Notation:




```
:- op(50, xfx, ++).                    % Operatordefinition
```

```
% normalise(Sum, NormalisedSum) :-
% NormalisedSum ist das Ergebnis der Normalisierung des Ausdrucks Sum.

normalise(Exp, Norm) :-
    normalise_ds(Exp, Norm++0).

normalise_ds(A, (A+Space)++Space) :-
    atomic(A).

normalise_ds(A+B, Norm++Space) :-
    normalise_ds(A, Norm++NormB),
    normalise_ds(B, NormB++Space).
```

12.4 Wörterbücher

Eine andere Verwendung von Differenzstrukturen erlaubt eine einfache Implementierung elementarer Lexika. Es wird vorausgesetzt, daß sie aus einer Folge von Schlüssel-Wertpaaren bestehen, jeder Schlüssel nur einmal vorkommt und jedem Schlüssel nur ein Wert zugeordnet ist.

Folgende Operationen sollen möglich sein:

- den Wert festzustellen, der unter einem gegebenen Schlüssel gespeichert ist
- das Lexikon durch neue Einträge zu erweitern

```
% Ein Lexikon wird durch eine Liste von Einträgen repräsentiert.
% lookup(Key, Dictionary, Value) :-
% Dictionary enthält für den Schlüssel Key den Wert Value.
% Dictionary wird repräsentiert als eine Liste von Paaren (Key, Value).

lookup1(Key, [(Key, Value)|Dictionary], Value) :- !.

lookup1(Key, [(Key1, Value1)|Dictionary], Value) :-
    Key \== Key1,!,
    lookup1(Key, Dictionary, Value).
```

Eine effizientere Darstellung des Lexikons erhalten wir durch die Verwendung von binären Bäumen:

```
% Das Lexikon wird durch einen binären Baum repräsentiert.

lookup2(Key, dict(Key, X, Left, Right), Value) :-
    !, X = Value.

lookup2(Key, dict(Key1, X, Left, Right), Value) :-
    Key < Key1 , !,
    lookup2(Key1, X, Left, Value).

lookup2(Key, dict(Key1, Left, Right), Value) :-
    Key > Key1, !,
    lookup2(Key, Right, Value).
```

```
% Testdaten:
dictio([(arnold, 8881), (barry, 4513), (cathy, 5950)|Xs]).
dict(4513, barry, dict(5950, cathy, L1, R1), dict(8881, arnold, L2, R2)).
```

Literaturverzeichnis

[Bratko 1990] Ivan Bratko. *Prolog programming for artificial intelligence.* Addison-Wesley, Workingham, 1990.

[Clocksin/Mellish 1990] William F. Clocksin und C.S. Mellish. *Programmieren in Prolog.* Springer Verlag, Berlin, 1990.

[Covington 1994] Michael Aaron Covington. *Natural language processing for Prolog programmers.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Deransart 1996] Pierre Deransart. *Prolog : The standard. Reference Manual.* Springer Verlag, Berlin, 1996.

[Gazdar/Mellish 1989] Gerald Gazdar und Chris Mellish. *Natural language processing in PROLOG : An introduction to computational linguistics.* Addison-Wesley, Workingham, 1989.

[Spivey 1996] J. M. Spivey. *An introduction to logic programming through Prolog.* Prentice Hall, London, 1996.

[Sterling/Shapiro 1994] L. Sterling und E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1994.

Index

append, 50, 52, 97, 131, 135
arg, 65–70, 90
assert, 118
asserta, 118
assertz, 118, 119, 121
atom, 62, 63
atomic, 62–64, 68, 87, 88, 90, 133, 134, 137

Backtracking, 33, 34, 43, 80, 90, 93, 96, 97, 99, 109, 111, 117, 118, 120, 121
Beweisbaum, 22, 31, 93, 101, 120

call, 90
clause, 117, 119
compound, 62–64, 67, 68, 70, 118
consult, 106, 108, 109, 121
cut, 93–100
cut–fail, 100

delete, 49, 50
depth-first Suche, 30, 31, 33, 43, 93

fail, 80, 93, 100, 102, 119–121
Fakt, 13, 14, 16–19, 63, 118, 119, 122
Frage, 11, 12
Fragen, 13, 15–18, 22, 103
functor, 65–71, 88, 90

Generalisierung, 16, 18
get, 105, 114
get0, 105, 114–116

Hornklausel, 11–13

Instanz, 16–22, 25, 26, 28, 29, 49
integer, 62, 63, 84
is, 56, 60

member, 25, 47, 49, 52, 53, 99, 119
mod, 55, 57
Modus Ponens, 19, 22

nonvar, 84–90
not, 93, 98, 100–102, 116, 123
numeric, 62–64

op, 127

put, 105, 114

read, 105, 109, 111, 113, 121, 123–125
real, 62, 63
reconsult, 106, 107, 109
Reduktion, 22, 23, 28, 29, 31–33
Regel, 13, 14, 18–20, 22
repeat, 120, 121

see, 104, 113, 121
seeing, 104
seen, 104, 113, 121
struct, 87, 88, 90
Substitution, 16–18, 20, 25, 26, 68
Suchbaum, 31–33, 95, 101

tell, 104, 126
telling, 104, 126
Term, 13, 14, 16, 21, 25, 26, 28, 56–58, 65–68, 84, 87, 88, 109, 111, 113, 121, 124
told, 104, 126

var, 84
Variable, 13, 15–18, 25, 27–29, 41, 56–59, 62, 63, 69, 83–85, 87–89, 91, 103, 117, 118, 131

write, 80, 82, 105, 109–111, 113, 121,
123, 124