Einführung in Prolog mit SWI-Prolog

Helmar Gust
Armin Tischler
Philipp Hügelmeyer
Patrick Tschorn
Timo Steffens

März 1999

Inhaltsverzeichnis

Kapitel 1

Einleitung

Prolog steht für „PROgramming in LOGic'ünd wurde um 1970 von Alain Colmerauer und anderen an der Universität von Aix-Marseille entwickelt. Typische Anwendungsgebiete für Prolog sind heute in nahezu allen Bereichen der Symbolischen Datenverarbeitung zu finden, z.B. in den folgenden:

- maschinelle Sprachverarbeitung
- symbolisches Gleichungslösen
- Expertensysteme
- mathematische Logik
- automatisches Beweisen
- ...

Ein Prolog-Programm läßt sich als eine Wissensbasis auffassen, über der Fragen beantwortet werden können. Prolog selbst ist eine interaktive Sprache, denn die Programmentwicklung selbst läuft eher als Dialog denn als die Eingabe von längeren Programmen in einem Stück ab: Es wird dem System das nötige Wissen vermittelt, welches es braucht, um Anfragen korrekt beantworten zu können. Im Gegensatz zu anderen Programmiersprachen (Lisp, Pascal etc.), deren Grundelemente Funktionen bzw. Prozeduren sind, gibt es in Prolog als Grundelemente Prädikate bzw. Relationen:

- Prolog ist die Implementation eines Beweisverfahrens für einen prädikatenlogischen Kalkül (Resolutionsverfahren über Hornklauseln). Ein Prolog-Programm besteht aus einer Menge von prädikatenlogischen Aussagen (einer bestimmten Form) - den Axiomen -, relativ zu denen andere Aussagen bewiesen werden können.

Das hier verwendete Prolog-System SWI_Prolog ist eine Implementation eines Standardprologsystems.

Das hier verwendete Prolog-System SWI-Prolog ist public domain und damit frei verfügbar. Es läuft auf mehreren Plattformen, u. a. unter Windows (ab Win32s), OS/2 und Linux.

Dieser Kurs ist als Tutorium zum interaktiven Studium mit dem am Institut für Semantische Informationsverarbeitung entwickelten netzwerkweiten Lernumgebung konzipiert.

Kapitel 2

Erste Schritte

2.1 Einfache Klauseln (Fakten)

Aus der Sicht von Prolog besteht die Welt aus Dingen, aus Eigenschaften von Dingen und aus Beziehungen zwischen Dingen. Ein kleines Beispiel erklärt die Zusammenhänge:

natürliche Sprache:	Übersetzungen in Prolog:
Maria ist eine Frau.	`frau(maria).`
Michaela ist eine Frau.	`frau(michaela).`
Jens ist ein Mann.	`mann(jens).`
Bernd ist ein Mann.	`mann(bernd).`
Maria ist die Mutter von Jens.	`mutter(maria,jens).`
Anke ist die Mutter von Maria.	`mutter(anke,maria).`
Bernd ist der Vater von Jens.	`vater(bernd,jens).`

In obigem Beispiel stehen links also natürlichsprachliche Sätze und rechts die Übersetzungen in Prolog. Die Ausdrücke auf der rechte Seite des Beispiels werden *Klauseln* genannt. Genauer gesagt handelt es sich hier um eine bestimmte Sorte von Klauseln, nämlich um *Fakten*. Um nun dieses Wissen dem Prolog-System zur Verfügung zu stellen, gibt es verschiedene Möglichkeiten.

Solch einfache kleine Beispiele lassen sich unmittelbar ins Editorfenster eintippen. Um eine Anfrage auszuführen, muss das Programm zunächst compiliert werden. Im dann erscheinenden Fenster kann die Anfrage eingegeben werden.

```
frau(maria).      (Im Editorfenster einfügen.)
frau(michaela).
```

Wichtig ist, sich zu merken, daß solche Klauseln immer mit einem '.' aufhören. Und der Name des Prädikats, d. h. einer Eigenschaft (z.B. 'frau') oder einer Beziehung (z.B. 'mutter') steht immer an der ersten Stelle direkt vor der öffnenden Klammer. Sind nun Fakten in der Wissensbasis eingetragen, so möchte man auch Anfragen stellen können. Beispielsweise anfragen, ob der Fakt, daß Maria eine Frau ist, in der Wissensbasis eingetragen ist. Dies geschieht folgendermaßen: Zunächst muß der Source-Code kompiliert werden. Dann kann man im erscheinenden Fenster die Anfrage eingeben.

```
frau(maria).     (Klick auf Execute führt zu:)
Yes.
```

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

```
frau(claudia).
No.
```

So sieht die Antwort aus, wenn die Anfrage nicht erfüllt werden kann. Ganz analog lassen sich auch Anfragen zu den anderen Fakten stellen.

```
mann(jens).    (Eingabe des Benutzers)
Yes. (Ausgabe des Systems)
mann(bernd). (Eingabe des Benutzers)
Yes. (Ausgabe des Systems)
```

Übung: Gib folgendes Beispielprogramm ein.

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

mann(jens).
mann(bernd).

mutter(maria, jens).
mutter(anke, maria).

vater(bernd, jens).
```

Wichtig: Solche Klauseln werden immer mit einem '.' beendet! Der Name des Prädikats, d.h. einer Eigenschaft (z.B. `frau`) oder einer Beziehung (z.B. `mutter` steht vor der Klammer. In den Argumentklammern werden die Argumente durch ein Komma getrennt.

Anschließend den Menüpunkt *Services/ Compile* wählen.

Diese Faktenmenge bildet ein vollständiges Prolog-Programm, das sofort ausgeführt werden kann.

Sind nun Fakten in der Wissensbasis eingetragen, so möchte man auch Anfragen stellen können. Beispielsweise anfragen, ob der Fakt, daß Maria eine Frau ist, in der Wissensbasis eingetragen ist. Für Anfragen ist das GOAL-Fenster da

```
 ?- frau(maria) ?
```

Auch Anfragen müssen mit einem '.' abgeschlossen werden.

Mit dem Execute-Knopf wird die Anfrage gestartet. Die Antwort erscheint im AUSGABE-Fenster.

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

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

```
?- frau(claudia).
   No
```

Diese Ausgabe zeigt an, daß die Eigenschaft Claudias, eine Frau zu sein, dem System nicht bekannt ist. Dies darf nicht mit der Information verwechselt werden, daß Claudia keine Frau ist (also der Negation der Anfrage). Wir werden später noch sehen, daß es in Prolog aus tiefliegenden logischen Gründen keine direkte Möglichkeit gibt, negierte Sachverhalte auszudrücken.

Anfrage:

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

Anfrage:

```
?- mann(maria).
   No
```

Man kann in Prolog nicht nur konkrete Angaben überprüfen (bestätigen) lassen, so wie oben '(frau maria)', sondern auch anfragen, ob es Kandidaten gibt, die beispielsweise eine bestimmte Eigenschaft erfüllen. Diese Art der Anfrage wird in Prolog mit „Platzhaltern“ formuliert. In Prolog geschieht dieses auf folgende Weise.

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

?- mann(X).
   X = jens

?- mutter(X, Y).
   X = mria, Y = jens
```

Die Anfrage `frau( X)` ist folgendermaßen zu verstehen:

> Welcher Wert kann für `X` eingesetzt werden, so daß die Anfrage erfüllt werden kann (d.h. aus dem Programm ableitbar wird)?

Eine Antwort darauf lautet `maria`, wir erhalten daher die Ausgabe `X = maria`. Das System liefert immer den ersten Eintrag, der auf die Anfrage paßt. Möchte wir alle möglichen Antworten ausgegeben haben, so erreichen wir diese durch Einschalten der Option 'ALL'.

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

?- mann(X).
   X = jens ;
   X = bernd ;
   No.
```

Einige Hinweise zum Umgang mit dem Kurssystem:

- Der Inhalt des Editor-Fenster wird nach dem Anklicken von *SERVICES/compile* insgesamt übersetzt und steht als neue Programmversion zur Verfügung.
- Anfragen werden in das GOAL-Fenster geschrieben und mit einem Klick auf *Execute* gestartet.
- Ist die Option 'All' ausgeschaltet, liefert das System die erste gefundene Lösung, (d.h. Belegung der Variablen, für die die Anfrage wahr wird).
- Ist die Option *All* eingeschaltet, werden alle Lösungen in der Reihenfolge ihrer Konstruktion aufgezählt. Vorsicht: Programme können auch unendlich viele Lösungen haben.
- Anfragen können durch einen nochmaligen Klick auf die *Execute*-Taste (die während der Ausführung einer Anfrage mit *Interrupt* beschriftet ist) abgebrochen werden.

2.1.1 Übungen

1. Gib die folgenden Fakten und Regeln über das Editor-Fenster ein.

```
weiblich(maria).
weiblich(hanna).
weiblich(heide).
weiblich(renate).
weiblich(annabella).
```

```
weiblich(erna).
weiblich(britta).
weiblich(mathilda).

maennlich(john).
maennlich(manfred).
maennlich(ingo).
maennlich(helmar).

vater(heinrich, renate).
vater(konrad, ingo).
vater(ingo, manfred).
vater(ingo, maria).

mutter(katharina, ingo).
mutter(renate, manfred).
mutter(renate, maria).
mutter(anna, renate).
```

Compilieren nicht vergessen!

(a) Experimentiere mit verschiedenen Anfragen zu diesen Fakten. Wie kann man alle Männer bzw. Frauen ausgeben? Wie kann man feststellen, ob eine Person in der Wissensbasis gespeichert ist?

(b) Wie kann man den Vater bzw. die Mutter zu einer Person erhalten? Welche Anfrage liefert die Kinder einer Person.

2.2 Kompliziertere Klauseln

Es wäre langweilig, wenn Prolog nur Klauseln in Form von Fakten und Anfragen akzeptieren würde. Denn dann hätte man keine Möglichkeit, Beziehungen (Relationen) zwischen Fakten auszudrücken und abzufragen. Daher gibt es in Prolog noch eine weitere Art von Klauseln, die *Regeln*. Mit ihnen lassen sich Beziehungen zwischen Fakten (oder Regeln untereinander) darstellen. Um die Eltern zu einer Person anzugeben, können wir die obige Wissensbasis folgendermaßen erweitern:

natürlichsprachlich: Maria und Bernd sind die Elten von Jens.

Prolog-Darstellung: `eltern(maria, bernd, jens).`

Durch die Definition des dreistelligen Faktums `eltern` tritt eine gewisse Redundanz in der Wissensbasis auf, denn daß Maria die Mutter und Bernd der Vater von Jens ist, wissen wir bereits. Durch die Verwendung von Regeln, können wir dieser Redundanz entgehen. Denn mit Regeln lassen sich gewisse Aussagen aus anderen ableiten, so daß es nicht nötig ist, alle benötigten Aussagen direkt als Fakten in die Wissensbasis einzutragen. Die Regel `eltern` läßt sich folgendermaßen in Prolog darstellen:

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

> eltern(V, M, K)
> V = bernd, M = maria, K = jens
```

Unsere Regel besagt dann also:

> `V` und `M` sind die Eltern von `K`, *falls*
> `V` der Vater von `K` ist *und*
> `M` die Mutter von `K` ist.

Das Prolog-Zeichen `:-` kann also als *falls* gelesen werden. Die Ausdrücke auf der rechten Seite dieses Zeichens werden als durch *und* verknüpft interpretiert. In unserer Regel `eltern` wird `eltern(V, M, K)` als *Kopf* und `mutter(M, K)` `vater(V, K)` als *Rumpf* der Klausel bezeichnet. Eine Regel besagt, daß der Kopf dann wahr ist, wenn sich jeder einzelne Ausdruck seines Rumpfes nachweisen läßt. In folgendem Beispiel soll die obige Wissensbasis um die Beziehung "Großmutter" erweitert werden. Wir betrachten zuerst den Fall der Großmutter mütterlicherseits. Halbformal ließe sich das folgendermaßen ausdrücken:

> `GM` ist die Großmutter von `K`, *falls*
> `GM` die Mutter einer `M` ist *und*
> `M` die Mutter von `K`.

In Prolog ausgedrückt, ergibt es folgende Regel:

```
grossmutter(GM, K) :-
  mutter(GM, M),
  mutter(M, K).
```

Im Anschluß an diese Zeilen muß das Programm übersetzt werden. Dadurch werden die Regel und alle anderen Klauseln, die sich im Editor-Fenster befinden, in die Wissensbasis eingetragen.

Das Zeichen `:-` bedeutet: Die linke Seite gilt, *falls* die rechte Seite gilt.

Da in diesem Fall nicht egal ist, durch welche Werte die Platzhalter in der Klausel `mutter` belegt werden, gibt man ihnen Namen. Nämlich `GM` und `M` in der ersten *mutter*–Klausel und `M` und `K` in der zweiten. Es ist wichtig, daß die Variable `M` sowohl im ersten als auch im zweiten Vorkommen von `mutter` den gleichen Wert hat. Andernfalls wäre die Regel `grossmutter` nicht korrekt und würde zu falschen Ergebnissen führen.

```
?- grossmutter(X, Y).
   X = anke, Y = jens
```

Wir erhalten nur eine Antwort, da in der Wissensbasis nur eine der Anfrage entsprechende Beziehung vorhanden ist.

Die Definition der Großmutter väterlicherseits, sieht ganz analog aus.

```
grossmutter(GM, K) :-
  mutter(GM, V),
  vater(V, K).
```

Ein Problem tritt auf, wenn z.B. die Relation "Bruder" definiert werden soll:

```
bruder(X, Y) :-
  mutter(M, X),
  mutter(M, Y),
  vater(V, X),
  vater(V, Y).
```

Dieses Programm schließt nicht aus, daß jemand sein eigener Bruder ist und stimmt daher mit der intuitiven Bedeutung von Bruder nicht überein. Um dies zu beheben müssen wir Ausdrücken, daß `X` ungleich `Y` sein muß. Wir erhalten dann die Klausel:

```
bruder(X, Y) :-
  mutter(M, X),
  mutter(X, Y),
  vater(V, X),
  vater(V, Y),
  not  X == Y.
```

Wir werden später sehen, daß solche negativen Bedingungen mit Vorsicht zu behandeln sind!

Prolog-Programme sind sehr modular. Jede Klausel ist ein selbständig ausführbares Stückchen Programm. Solange wir gewisse Restriktionen einhalten, die später noch ausführlich diskutiert werden, ist die Bedeutung einer Klausel unabhängig vom dem Kontext, in dem sie steht.

Eine Menge von Klauseln mit gleichem Namen des Prädikates im Kopf und gleicher Stelligkeit bezeichnet man als Definition dieses Prädikats. Unter Stelligkeit versteht man die Anzahl der Argumente. Der Name einer Klausel (also Regel oder Faktum) wird auch mit *Funktor* bezeichnet.

Aufbau von Programmen

- Ein Programm besteht aus einer Liste von Fakten und Regeln. Jede solche Liste, deren Elemente syntaktisch korrekte Fakten oder Regeln sind, ist ein ausführbares Prolog-Programm!

- Einträge zum gleichen Prädikat werden zusammengefaßt und bilden die Definition dieses Prädikats auch wenn sie nicht zusammenstehen.
- Kommentare werden durch die Zeichen `/*` und `*/` eingeschlossen. Alle Zeichen in einer Zeile nach einem `%` werden ingoriert.

In Prolog gibt es vordefinierte Operatoren, die infix stehen dürfen (also in der Form $arg_1 \odot arg_2$). Diese Operatoren betreffen insbesondere Arithmetik und Vergleiche. Die folgenden Ausdrücke dürfen an der Stelle einer elementaren Aussage auf Bedingungsseite einer Regel stehen.

Expr1 > Expr2 Expr1 ergibt eine Zahl größer als Expr2.

Expr1 < Expr2 Expr1 ergibt eine Zahl kleiner als Expr2.

Expr1 =< Expr2 Expr1 ergibt eine Zahl nicht größer als Expr2.

Expr1 >= Expr2 Expr1 ergibt eine Zahl nicht kleiner als Expr2.

Expr1 =:= Expr2 Beide Ausdrücke ergeben die gleiche Zahl.

Expr1 =\= Expr2 äquivalent zu `not(Expr1=:=Expr2)`.

Term1 == Term2 Beide terme haben die gleiche Referenz .

Term1 \== Term2 äquivalent zu `not(Term1==Term2)`.

Term1 = Term2 unifiziert Term1 with Term2

Term1 \= Term2 äquivalent zu `not(Term1=Term2)`.

Auch ',' und ';' sind Infixoperatoren.

2.2.1 Übungen

1. Erweitere das Programm 2.1.1: Lade das Programm in das Editor-Fenster und ergänze die Regel

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

   Experimentiere mit diesem Programm:

   - Mit welcher Anfrage kann man erfahren, welche Eltern welche Kinder haben? Nimm den Aufruf, der alle Lösungen liefert.
   - Welche Anfrage liefert die Namen der Kinder von Renate?

- Finde etwas über Manfred heraus. Wie heißt die Mutter von Manfred, wie der Vater? Verwende dabei eine Verknüpfung von zwei Anfragen. Wie heißt diese Verknüpfung?
- Erweitere obige Wissensbasis um eine Regel, die die Information, daß Maria die Schwester von Manfred ist, liefert. Es sollte folgendes möglich sein:

```
?- schwester(X, Y).
   X = maria, Y = manfred
```

Bei der Formulierung der Regel hilft folgende Überlegung:

> `X` ist eine Schwester von `Y`, *wenn* `X` weiblich ist *und* die Eltern, repräsentiert durch die Variablen `M` und `V`, die Eltern von `X` sind *und* die Eltern, repräsentiert durch die Variablen `M` und `V` auch *gleichzeitig* die Eltern von `Y` sind.

Erweitere die Wissensbasis um eine Regel, die eine Großmutterbeziehung (mütterlicherseits oder aber väterlicherseits realisiert. Von Manfred heißt die Großmutter väterlicherseits Katharina und mütterlicherseits Anne.

2. Formalisiere folgende Sachverhalte in Form eines Prolog-Programmes:

   (a) Männer lieben Frauen, die Computerlinguistik studieren.

   (b) Wer intelligent ist und das Abitur hat, studiert Computerlinguistik.

   (c) Maria ist eine Frau.

   (d) Maria ist intelligent.

   (a) Formuliere die Frage „Liebt John Maria" als Prolog-Anfrage. Falls das Programm korrekt geschrieben und die Frage korrekt gestellt worden ist, sollte das Prolog-System *keine* Antwort finden.

   (b) Welche Sachverhalte fehlen, um die Frage beantworten zu können? Trage diese Sachverhalte nach und probiere das Programm aus.

3. Meyers planen ein Picknick im Grünen. Dazu benutzen sie die folgende Wetterprognose für die nächste Woche:

   - Montag wolkenlos 30 Grad.
   - Dienstag regnerisch 18 Grad.
   - Mittwoch bedeckt 26 Grad.
   - Donnerstag sonnig 30 Grad.
   - Freitag unbeständig 29 Grad.
   - Samstag wolkenlos 30 Grad.

Sie möchten für das Picknick einen Tag finden, an dem alle frei haben, und das Wetter schön ist. Das Wetter ist schön, wenn es wolkenlos ist, oder die Sonne scheint. Da Hans, der Vater, Astronom ist, arbeitet er, wenn der Himmel wolkenlos ist. An anderen Tagen hat er frei.

Fritz, der Sohn, geht von Montag bis Samstag in die Schule, hat aber schulfrei, falls die Temperatur auf über 28 Grad steigt. (`X ¡ Y` ist wahr, genau dann, wenn er Wert von `X` kleiner als der Wert von `Y` ist.)

Eva, die Mutter, hat Sonntagsdienst im Krankenhaus, sonst aber frei.

Entwickle ein Prolog-Programm, das die Tage bestimmt, an denen Meyers ihr Picknick machen können.

Kapitel 3

Kurze Einführung in die Prädikatenlogik (PL1)

Programmieren bedeutet in Prolog nichts anderes als das Erstellen einer Wissensbasis. In ihr muß der Programmierer alles, was er über die Objekte der darzustellenden Welt und ihre Beziehungen zueinander weiß, in Form von logischen Formeln spezifizieren. Für Prolog ist daher die Prädikatenlogik (erster Stufe) sehr wichtig. Deshalb folgt nun eine kurze Einführung.

3.1 Syntax

Wenn wir Aussagen über einen Weltausschnitt ausdrücken wollen, so müssen wir zuerst die Objekte beschreiben, die in ihm vorkommen, d. h. wir müssen die Syntax der Bezeichner angeben. Im Prädikatenkalkül heißen diese Bezeichner der Objekte *Terme*:

Ein Term ist:	Prolog-Notation:
eine *Konstante* a,b,c,. . .,1,2,3	a,b,c,. . .,1,2,3
eine *Variable* x,y,z,Mutter	X, Y, Z, Mutter
ein zusammengesetzter Term (*Funktion*) f(x,y,z), g(a)	f(X,Y,Z), g(a)

Ein zusammengesetzter Term besteht aus einem *Funktor* (Funktionssymbol), dem in Klammern und durch Kommata getrennt eine Liste von Termen folgt. Die Anzahl der Terme, die einem Funktor folgen, heißt *Stelligkeit* des Funktors.

In Prolog-Notation müssen Funktionssymbol mit einem kleinen und Variablen mit einem großen Buchstaben beginnen.

Zwischen solchen Objekten werden Relationen (Beziehungen) mit Hilfe von *Formeln* ausgedrückt. Eine *atomare Formel* besteht aus einem *Prädikatsnamen* (`p`, `q`, `mutter`, . . .), dem in Klammern und durch Kommata getrennt eine Liste von

Termen folgt (Argumente des Prädikates).

Atomare Formeln haben wir schon in Abschnitt 2 kennengelernt, z. B. sind `frau(maria)` und `mann(jens)` atomare Formeln. In Prolog werden Funktoren und Prädikatsnamen in ihrer Schreibweise nicht unterschieden. Der Unterschied liegt jedoch darin, daß Prädikate einen Wahrheitswert haben, z.B. `frau(x)` ist entweder wahr oder aber falsch. Dagegen liefern Funktionen irgendeinen Wert, z.B. `alter(hans)` liefert sich selbst als Wert.

In Prolog werden aber Funktionsausdrücke nicht ausgewertet! In Prolog liefert `alter(hans)` sich selbst, d. h. die Struktur `alter(hans)` als Wert.

Eine Formel ist von der Form:

Formel	Beschreibung	Prolognotation
q(x,y,z)	q Prädikatsname	`q(X,Y,Z)`

Sind F und G Formeln, so auch:

Formel	Beschreibung	Prolognotation
$(F \wedge G)$	Konjunktion: F *und* G	`F , G`
$(F \vee G)$	Disjunktion: F *oder* G	`F ; G`
$(F \rightarrow G)$	Implikation: aus F *folgt* G)	`G :- F`
$(\neg F)$	Negation: *nicht* F	`not(F)`

Die Verwendung dieser sog. Junktoren, die aus einfachen Formeln komplexe Formeln bilden, ist in Prolog aber stark eingeschränkt.

Der Implikationspfeil wird im Folgenden auch in umgekehrter Orientierung benutzt, wobei die Bedeutung ganz analog aufzufassen ist. ($(G \rightarrow F)$ bedeutet ebenfalls *aus F folgt G*)

Ist F eine Formel und x eine Variable, dann sind auch

- $(\exists x : F)$ (Existenzquantor, *es existiert ein x*),
- $(\forall x : F)$ (Allquantor, *für alle x*)

Formeln.

Die Prädikatenlogik besteht aus der Menge aller atomaren Formeln und den Formeln, die nach obigen Gesetzen gebildet werden können. Im folgenden nun ein paar Beispiele:

$$(\forall x : (\exists y : (vater(y, x)))) \tag{3.1}$$

$$(\exists y : (\forall x : (vater(x, y)))) \tag{3.2}$$

Häufig werden bei größeren Formeln aus Gründen der Übersichtlichkeit auch Klammern weggelassen. Bei folgendem Ausdruck ist aber immer noch klar, wie er aufgebaut ist:

$$\forall x : \exists y : vater(y, x) \tag{3.3}$$

Jeder hat einen Vater.

In Prolog interessieren jedoch nicht alle Formeln, sondern nur diejenigen, die (zumindest prinzipiell) beantwortbar sind, d. h. es werden nur Aussagen betrachtet. In der Formel

$$mutter(x, a) \tag{3.4}$$

kommt die Variable x vor, deshalb können wir der Formel keine sinnvolle Bedeutung zukommen lassen. Anders ist es, wenn die Variable quantifiziert ist:

$$\forall x : mutter(x, a) \exists x : mutter(x, a) \tag{3.5}$$

Für diese beiden Formeln können wir die Frage, ob „die Formel wahr oder falsch ist“, sinnvoll stellen.

Prolog ist eine Untermenge der Prädikatenlogik ohne Quantoren, d. h. es gibt kein $\exists$ und kein $\forall$. Deshalb können Quantoren in Prolog auch nicht explizit verwendet werden. Dies bedeutet nicht, daß Quantifikation in Prolog keine Rolle spielt. Es gilt die Vereinbarung, daß alle Variablen in einer Klausel allquantifiziert sind. Alle oben genannten Formeln, in denen Existenzquantoren vorkommen, können allerdings so in Prolog nicht dargestellt werden. Wie man mit den Existenzquantoren in Prolog umgeht, wird in Kapitel 7.1 dargestellt.

3.2 Semantik

Die Zuordnung von Bedeutungen zu den nur syntaktisch spezifizierten logischen Ausdrücken geschieht in zwei Schritten.

Zunächst müssen wir festlegen, welche Individuen die atomaren Terme bezeichen, denn die Bedeutung eines Terms ist dasjenige Individuum, das er bezeichnet. Diese Zuornung ist willkürlich und kann weitgehen frei gewählt werden. Der Term *hans* z.B. bezeichne etwa einen bestimmten Menschen mit Namen Hans. Wir könnten auch vereinbaren, daß *hans* auf einen Stein verweist.

Freie Variablen kommen nicht vor, denn Variablen sind implizit allquantifiziert. An der Position von Variablen dürfen also beliebige Terme stehen. Wir müssen daher die Referenz einer Variablen für jeden möglichen Wert betrachten.

Ein komplexer Term, der keine Variablen enthält, referiert auf ein Individuum, das durch die Bedeutung der in ihm enthalten atomaren Terme und durch seine Struktur eindeutig bestimmt ist. Dies ist zu beachten, wenn komplexe Terme gebildet werden. So kann der Term $vater(hans)$ die intendierte Bedeutung 'derjenige Mensch, der Vater von Hans ist' haben. Da Prolog eine Logik ohne Gleicheit ist, kann aber z.B. nicht ausgedrückt werden, daß der Term $fritz$ die gleiche Bedeutung wie der Term $vater(hans)$ hat (d.h., daß Fritz der Wert der Vaterfunktion angewendet auf Hans ist). Da ein Mensch mehrere Brüder haben kann, macht es z.B. keinen Sinn, eine Funktion 'bruder' einzuführen, um Terme der Form *bruder*($hans$) zu bilden. Wenn Hans mehrere Brüder hat, kann diesem Audruck keine sinnvolle Bedeutung zugeschrieben werden. Aber eine Funktione *brueder*($hans$)., die als Wert eine Menge, nämlich die Menge der Brüder von Hans hat, macht wieder Sinn.

Die Bedeutungen von Aussagen sind die Wahrkeitswerte w und f, d. h. eine Aussage kann wahr oder falsch sein.

Der Wahrheitwert einer elementaren Aussage $q(t1, \ldots, tn)$ bestimmt sich aus der Bedeutung des Prädikates q und der Bedeutung der Terme $t1, \ldots, tn$. Z.B. ist $vater(fritz, hans)$ ($vater$ jetzt als Prädikat und nicht als Funktion interpretiert) genau dann wahr, wenn die Bedeutung von $fritz$ zur Bedeutung von $hans$ (jeweils die Individueen, auf die diese Terme verweisen) in der Relation zueinander stehen, die wir als Bedeutung des Prädikats $vater$ annehmen wollen. Daraus folgt, daß wir auch die Wahrheitswerte der elementaren Formeln durch eine geeignete Bedeutungszuordnung für die Prädikate relativ willkürlich festlegen können.

Die Wahrheitswerte komplexer Aussagen berechnen sich dann in eindeutiger Weise aus den Wahrheitswerten der enthaltenen elementaren Aussagen für alle möglichen Belegungen der in ihnen enthaltenen Variablen nach folgendem Schema:

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$
w	w	f	w	w	w
w	f	f	f	w	f
f	w	w	f	w	w
f	f	w	f	f	w

Quantoren brauchen wir hier nicht zu betrachten, denn sie kommen in Prolog nicht explizit vor. Es reicht, einen Bereich festzulegen, der als Belegung für Variablen in Frage kommt.

Um die „Bedeutung“ eines Prolog-Programms zu erkennen, muß man also wissen, welche Bedeutung der atomaren Terme und der Prädikate der Programmierer im Sinn hatte, und welchen Bereich er sich als Belegung der Variablen vorstellte. Der Programmierer sollte dies dem Leser durch eine geeignete Wahl der Bezeichner erleichtern.

Insbesondere für theoretische Überlegungen gibt es allerdings auch die Möglichkeit, sich eine „kanonische Bedeutung“ eines Prolog-Programms zu konstruieren, die unabhängig von den Intentionen des Programmierers ist.

3.3 Übersetzung natürlichsprachlicher Aussagen in Prädikatenlogik und Prolog

In diesem Abschnitt wollen wir uns Beispiele dafür anschauen, wie man von natürlichsprachlichen Aussagen zu Prädikatenlogischen Aussagen und schließlich zu Prolog-Klauseln gelangt. Ein vielzitiertes Beipiel ist das folgende:

3.3.1 Einige Beispiele

1. Beispiel: Möchte man ausdrücken, daß Klaus und Maria in der „verheiratet“ Beziehung zueinander stehen, so sagt man natürlichsprachlich:

> Klaus und Maria sind verheiratet.

In der Prädikatenlogik drückt man dieses mittels einer atomaren Formel aus:

$$verheiratet(Klaus, Maria) \tag{3.6}$$

Und in Prolog sieht dieses folgendermaßen aus:

```
verheiratet(klaus, maria).
```

2. Beispiel: Natürlichsprachliche Formulierung:

> Sokrates ist ein Mensch. Ein x ist sterblich, falls x ein Mensch ist.

Prädikatenlogische Formulierung:

$$Mensch(Sokrates) \tag{3.7}$$
$$\forall x : sterblich(x) \rightarrow Mensch(x) \tag{3.8}$$

Prolog-Notation:

```
mensch(sokrates).
sterblich(X) :-
  mensch(X).
```

3. Beispiel: Falls wir in einer heilen Welt leben, gilt die Regel:

> Jeder Vater schenkt seinem Sohn eine Spielzeugeisenbahn.

Prädikatenlogische Formulierung:

$$\begin{aligned} \forall x \forall y \forall z : (schenkt(x,y) \rightarrow \\ (Vater(x,y) \wedge \\ maennlich(y) \wedge \\ Spielzeugeisenbahn(z))) \end{aligned}$$

Prolog-Notation:

```
schenkt(X, Y, Z) :-
  vater(X, Y),
  maennlich(Y),
  spielzeugeisenbahn(Z).
```

3.4 Übungen

1. Formuliere folgende natürlichsprachliche Familienbeschreibung in prädikatenlogischer Schreibweise und in Prolog-Notation:

   Die Familie Meier besteht aus Kerstin, sie ist die Mutter, dem Vater Rainer, der Tochter Andrea und dem Sohn Uli. Außerdem gibt es noch den Hund Bello.

2. Zusätzlich sei noch bekannt, daß Kerstin und Rainer verheiratet sind und daß Andrea ihren Hund liebt.

   Formuliere diese Fakten ebenfalls in prädikatenlogischer Schreibweise und in der Prolog-Notation.

3. Da Familie Meier in einer heilen Welt lebt, gelten dort folgende Regeln:

   - Jede Mutter liebt ihr Kind.
   - Jeder Vater spielt sonntags mit seinem Sohn.

   Notiere diese Regeln ebenfalls in beiden Notationen.

3.5 Übersetzung in Prädikatenlogik und Klauselform TEIL II

Beispiel 1: Alle Personen in diesem Raum studieren Computerlinguistik.

Formalisierung:

Alle Personnen	$\forall x : person(x)$
in	$in(x$
diesem Raum	$, r1$
studieren Computerlinguistik	$\rightarrow studieren(x, cl).$

Grundschema für Allquantoren

$\forall x$	Einschränkung von x
$\rightarrow$	Kernaussage über x

Ergebnis:

$$\forall x : person(x) \wedge in(x, r1) \rightarrow studieren(x, cl). \tag{3.9}$$

Beispiel 2: Es gibt Personen in diesem Raum, die studieren Computerlinguistik.

Formalisierung:

Es gibt	$\exists x :$
Frauen	$frau(x)$
(und) hline in	$\wedge$ $in($
(diese Frauen) hline diesem Raum hline (und)	x $r1)$ $\wedge$
,die	x
studieren	$studieren(,$
Computerlinguistik	$cl)$

3.5.1 Grundschema für Existenzquantoren

$\exists x$	Einschränkung von x
$\wedge$	Kernaussage über x

Ergebnis:

$$\forall x : frau(x) \wedge in(x, r1) \wedge studieren(x, cl) \tag{3.10}$$

Beispiel 3: (Inklusionsbeziehungen) Männer sind Personen. (= Alle Männer sind Personen) Frauen sind Personen. (= Alle Frauen sind Personen)

Formalisierung:

$$\forall x : mann(x) \rightarrow person(x) \forall x : frau(x) \rightarrow person(x) \tag{3.11}$$

Beispiel 4: Alle Männer lieben eine Frau.

Formalisierung: (von oben nach unten und links nach rechts zu lesen)

Alle Männer	$\forall x : mann(x)$
lieben	$\rightarrow lieben(x, y)$
eine Frau	$\exists y : frau(y)$

Also:

$$\forall x : mann(x) \rightarrow \exists y : frau(y) \wedge lieben(x, y) \tag{3.12}$$

Es gibt eine zweite Interpretation:

Alle Männer	$\forall x : mann(x)$	
lieben	$\rightarrow lieben(x, y)$	Also:
eine Frau	$\exists y : frau(y)$	

$$\exists y : frau(y) \wedge \forall x : mann(x) \rightarrow lieben(x, y) \tag{3.13}$$

3.5.2 Transformation prädikatenlogischer Aussagen in Klauselform

Bezeichnung:

$\wedge$	logisches und
$\vee$	logisches oder
$\forall$	Allquantor
$\exists$	Existenzquantor
Q	steht für $\forall$ oder $\exists$
$F(x/t)$	in F sind simultan alle Vorkommen von x durch t zu ersetzen
F,$F1$,$F2$	beliebig komplexe Formeln

3.5.3 Wahrheitserhaltende Transformationen

Nicht alle sind invertierbar!

$$F1 \leftrightarrow F2 \Rightarrow F1 \rightarrow F2 \wedge F2 \rightarrow F1 \tag{3.14}$$
$$F1 \rightarrow F2 \Rightarrow \neg F1 \vee F2 \tag{3.15}$$
$$F1 \vee (F2 \wedge F3) \Rightarrow (F1 \vee F2) \wedge (F1 \vee F3) \tag{3.16}$$
$$F1 \wedge (F2 \vee F3) \Rightarrow (F1 \wedge F2) \vee (F1 \wedge F3) \tag{3.17}$$
$$F1 \wedge F2 \Rightarrow F2 \wedge F1 \tag{3.18}$$
$$F1 \vee F2 \Rightarrow F2 \vee F1 \tag{3.19}$$
$$F1 \wedge (Qx : F2) \Rightarrow Qx : F1 \wedge F2 \tag{3.20}$$
$$F1 \vee (Qx : F2) \Rightarrow Qx : F1 \vee F2 \tag{3.21}$$
$$\forall x : F1 \wedge F2 \Rightarrow (\forall x : F1) \wedge (\forall x : F2) \tag{3.22}$$
$$\neg(F1 \& F2) \Rightarrow \neg F1 \vee \neg F2 \tag{3.23}$$
$$\neg(F1 v F2) \Rightarrow \neg F1 \wedge \neg F2 \tag{3.24}$$
$$\neg(\forall x : F) \Rightarrow \exists x : \neg F \tag{3.25}$$
$$\neg(\exists x : F) \Rightarrow \forall x : \neg F \tag{3.26}$$
$$\tag{3.27}$$

Skolemisieren (wenn alle Quantoren außen sind):

$$\forall x_1, \ldots, \forall x_n, \exists y : F \Rightarrow \forall x_1, \ldots, \forall x_n : F(y/sf(x1, ..., xn)) \tag{3.28}$$

mit einer neuen Skolemfunktion sf

Strategie

1. Negation nach innen,

2. Quantoren nach außen,
3. Disjunktion nach innen,
4. Konjunktion nach außen,
5. Existenzquantoren eliminieren durch Skolemisieren,
6. Allquantoren weglassen,
7. Alle einzeln transformierten Aussagen eines Problems mit 'und' verknüpfen.

Es entsteht eine konjuktive Normalform: = Konjunktion von Disjunktionen von atomaren (möglicherweise negierten) Formeln

$$(\neg F1_1 \vee \ldots \vee (not)F1_{n1}) \tag{3.29}$$
$$\wedge(\neg F2_1 \vee \ldots \vee \neg F1_{n2}) \tag{3.30}$$
$$\ldots \tag{3.31}$$
$$\wedge(\neg Fm_1 \vee \ldots \vee \neg Fm_{nm}) \tag{3.32}$$

Jeder Disjunktionsteil ist eine Klausel.

Es gibt drei Fälle, die für solche Klauseln:

1. Sie enthalten nur negierte atomare Formeln.
2. Sie enthalten genau eine nicht negierte atomare Formel.
3. Sie enthalten mehr als eine nicht negierte atomare Formel.

Nur der 2. Fall entspricht normalen Prolog-Klauseln, die einfach in ein Programm eingetragen werden können, denn dann hat die Klausel die Form:

$$C \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee B_n \tag{3.33}$$

und dies entspricht der Implikation

$$C \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \tag{3.34}$$

und damit der Prolog-Klausel

```
C :- B1, ..., Bn .
```

Der 1. Fall entspricht Prolog-Anfragen.

Entstehen Klauseln des 3. Typs, dann handelt es sich um ein Problem, daß in Prolog nicht formalisierbar ist.

3.5.4 Ein logisches Puzzle

(gefunden im WWW):

> Captain Frank was out for a walk when it started to rain. He did not have an umbrella and he wasn't wearing a hat. His clothes were soaked, yet not a hair on his head got wet. How could this happen?

Formalisierung

Implizit angenommene Regel:

$$\begin{aligned} \forall x : raining \wedge \neg has\_umbrella(x) \wedge \neg has\_hat(x) \\ \Rightarrow \\ \forall y : hair(x, y) \rightarrow wet(y). \end{aligned}$$

Fakten

$$raining. \tag{3.35}$$

$$\forall y : hair(cf, y) \rightarrow notwet(y). \tag{3.36}$$

$$\tag{3.37}$$

Klauseln

```
wet(Y) :-
  hair(X, Y),
  not(has_hat(X)),
  not(has_umbrella(X)),
  raining .

raining .

not_wet(Y) :- hair(cf, Y) .
```

Konsistenztest

```
test :-
   not(wet(Y),
   not_wet(Y)) .
```

Answer

Captain Frank is completely bald.

3.5.5 „Steamroller"-Problem

> Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them.
>
> Also there are some grains, and grains are plants.
>
> Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants.
>
> Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves.
>
> Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails.
>
> Caterpillars and snails like to eat some plants. Therefore there is an animal that likes to eat a grain eating animal.

„Steamroller"-Problem in PL1

$\forall x : wolve(x) \rightarrow animal(x).$

$\forall x : fox(x) \rightarrow animal(x).$

$\forall x : bird(x) \rightarrow animal(x).$

$\forall x : caterpillar(x) \rightarrow animal(x).$

$\forall x : snail(x) \rightarrow animal(x).$

$\forall x : grain(x) \rightarrow plant(x).$

$\exists x : wolve(x).$

$\exists x : fox(x).$

$\exists x : bird(x).$

$\exists x : caterpillar(x).$

$\exists x : snail(x).$

$\exists x : grain(x).$

$\forall x : bird(y) \rightarrow \forall x : caterpillar(x) \rightarrow muchsmaller(x, y) \wedge \forall x : snail(x) \rightarrow muchsmaller(x, y).$

$\forall x : wolve(x) \rightarrow \forall x : fox(y) \rightarrow muchsmaller(y, x).$

$\forall x : foxes(x) \rightarrow \forall x : bird(y) \rightarrow muchsmaller(y, x).$

$\forall x : animal(x) \rightarrow \forall x : plant(y) \rightarrow liketoeat(x, y) \vee \forall x : animal(y) \wedge \exists z : plant(z) \wedge liketoeat(y, z) \wedge muchsmaller(y, x) \rightarrow liketoeat(x, y)$

$\forall x : wolve(x) \rightarrow \forall x : (grain(y) \vee fox(y)) \rightarrow \neg liketoeat(x, y)$

$\forall x : bird(x) \rightarrow \forall x : caterpillar(y) \rightarrow liketoeat(x, y) \wedge \forall x : snail(y) \rightarrow notliketoeat(x, y).$

$\forall x : caterpillar(x) \rightarrow \exists y : plant(y) \wedge liketoeat(x, y).$

$\forall x : snail(x) \rightarrow \exists y : plant(y) \wedge liketoeat(x, y).$

zu beweisen: $\exists x, y, z : animal(x) \wedge animal(y) \wedge grain(z) \wedge liketoeat(x, y) \wedge liketoeat(y, z)$

Klauselform

$\neg wolve(x) \vee animal(x).$

$\neg fox(x) \vee animal(x).$

$\neg bird(x) \vee animal(x).$

$\neg caterpillar(x) \vee animal(x).$

$\neg snail(x) \vee animal(x).$

$\neg grain(x) \vee plant(x).$

$wolve(sw).$

$fox(sf).$

$bird(sb).$

$caterpillar(sc).$

$snail(ss).$

$grain(sg).$

$\neg bird(y) \vee \neg caterpillar(x) \vee muchsmaller(x, y).$

$\neg bird(y) \vee \neg snail(x) \vee muchsmaller(x, y).$

$\neg wolve(x) \vee \neg fox(y) \vee muchsmaller(y, x).$

$\neg fox(x) \vee \neg bird(y) \vee muchsmaller(y, x).$

$\neg animal(x) \vee \neg plant(y) \vee liketoeat(x, y) \vee \neg animal(y1) \vee \neg plant(z) \vee \neg liketoeat(y1, z)) \vee \neg muchsmaller(y1, x) \vee liketoeat(x, y1).$

$\neg wolve(x) \vee \neg fox(y) \vee \neg liketoeat(x, y).$

$\neg wolve(x) \vee \neg grain(y) \vee \neg liketoeat(x, y).$

$\neg bird(x) \vee \neg caterpillar(y) \vee liketoeat(x, y).$

$\neg bird(x) or \neg snail(y) \vee \neg liketoeat(x, y).$

$\neg caterpillar(x) \vee plant(sf1(x)).$

$\neg caterpillar(x) \vee liketoeat(x, sf1(x)).$

$\neg snail(x) \vee plant(sf2(x)).$

$\neg snail(x) \vee liketoeat(x, sf2(x)).$

Übersetztes Goal:

$\neg animal(x) \vee \neg animal(y) \vee \neg grain(y) \vee \neg liketoeat(x, y) \vee \neg liketoeat(y, z)$

Kapitel 4

Datenstrukturen

Um mit Daten, insbesondere „großen“ Mengen von Daten, in vernünftiger Weise umgehen zu können, versieht man sie mit einer Struktur. D.h., man faßt Daten unter gewissen Gesichtpunkten zusammen. In Prolog geschieht dies durch Verwendung von Tupeln und Listen.

4.1 Funktionsausdrücke

Eine Möglichkeit, Daten fester Länge in Prolog zu repräsentieren, bildet die Verwendung von Funktionsausdrücke. Funktionsausdrücke bestehen aus einem Funktionsymbol gefolgt von den Argumenten:

```
f(a1,a2,a3,...)
```

Das Funktionssymbol wird auch als *Funktor* bezeichnet. Wie bereits angemerkt werden solche Ausdrücke in Prolog nicht ausgerechnet sondern als Datenstrukturen betrachtet. Daher werden sie im Normalfall dazu verwendet, zusammengehörige Daten zusammenzufassen. Als Beispiel ändern wir das uns schon bekannte Prädikat `eltern` ein wenig ab. Wir verwenden nun statt zweier Variablen für Vater und Mutter nur eine Datenstruktur, in der zwei Variablen als Komponenten stehen.

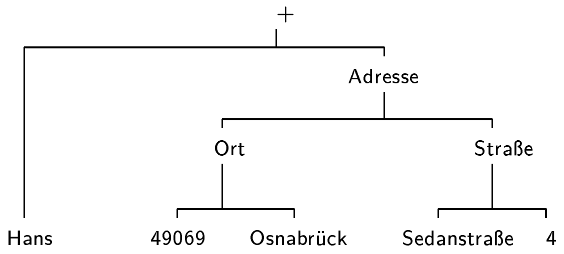
```
eltern(eltern(V, M), K) :-
  vater(V, K),
  mutter(M, K).
```

Wir betrachen nun wieder die Verwandschaftsbeziehungen aus Kapitel 2 und ändern die Elternklausel wie oben angegeben ab. Nun testen wir unsere neue Klausel und stellen die Anfrage, wer die Eltern von Jens sind.

```
? eltern(E, jens).
  E = eltern(bernd, maria)
```

Wie wir sehen, erhalten wir dieselbe Antwort wie bei unserer früheren Version auch, jedoch sieht die Anfrage anders aus. Anstatt zwei Variablen, für Vater und Mutter je eine, brauchen wir jetzt nur noch eine Variable für den komplexen Term. Darin liegt auch ein großer Nutzen in der Verwendung komplexen Termen. Denn wir brauchen die Struktur des Arguments bei der Anfrage überhaupt nicht zu kennen. Wir müssen nicht wissen, ob zuerst der Vater oder die Mutter kommt oder ob das Argument vielleicht mehr als zwei Komponenten besitzt.

Man kann komplexe Term beispielsweise auch dazu benutzen, eine Adreßdatei anzulegen. Ein Eintrag in eine Adreßdatei könnte folgendes Format haben:



Als Struktur dargestellt könnte eine solche Adresse folgendermaßen aussehen:

```
adresse(hans,
 adresse(ort(49069, osnabrück),
  strasse(sedanstrasse, 4))).
```

Eine Anfrage sieht dann folgendermaßen aus:

```
? adresse(hans, X).
  X = adresse(ort(49069, osnabrueck))
```

Es wird also `X` mit `adresse(ort(49069, osnabrueck), (strasse sedanstrasse 4)))` unifiziert.

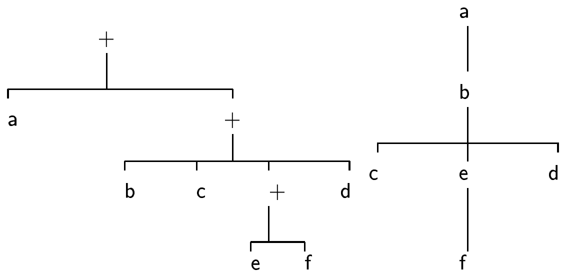
Die Verwendung von Strukturen bietet sich also immer dann an, wenn eine feste Anzahl zusammengehörender Terme zu einem Komplex zusammengefaßt werden sollen.

```
  Prädikat
  |              Funktor
  |                 |
adresse("Hans",adresse(wohnort(4500,"Osnabrück"),
               straße("Sedanstraße",4))).
```

(Die Anführungsstriche sind notwendig, da diese Terme sonst als Variablen interpretiert würden.)

Wie dieses Beispiel zeigt, lassen sich Strukturen als Bäume interpretieren und umgekehrt. Bäume sind eine der universellsten Datenstrukturen und treten in den unterschiedlichsten Kontexten auf, wie wir noch sehen werden.

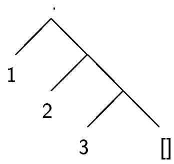
Aufgabe: Repäsentiere die folgenden Bäume als Prolog-Datenstrukturen



4.2 Listen

Während *Tupel* eine feste Länge haben, versteht man unter einer Liste eine Datenstruktur, deren Länge im Prinzip beliebig sein kann. Insbesondere gibt es Operationen, die Listen dynamisch verlängern oder verkürzen können.

Listen lassen sich auch mittels Strukturen darstellen. Dazu wählen wir eine spezielle Repräsentation, nämlich als binäre Bäume. Binäre Bäume sind Bäume, bei denen jeder Knoten entweder ein Blatt ist oder genau zwei Nachfolger hat. Die Liste mit den Elementen `1`, `2` und `3` hat dann folgende Form:



`nil` ist ein willkürlich gewähltes Symbol und repräsentiert die leere Liste. Anstelle des `nil` könnte dort jedoch auch eine Variable `R` stehen. Würde nun `R` den Wert `.(4,nil)` bekommen, so würde die Struktur die Form `.(1, .(2, .(3, (4, nil))))` annehmen.

Es gibt Prolog-Systeme, die nur diese Form von Listen zur Verfügung stellen:

- ein zweistelliges Funktionssymbol (Funktor) `.`,
- das Symbol `nil` für die leere Liste.

Nach dieser Definition sind folgende Beispiele Listen:

```
.(a,nil)                                    .
                                           / \
                                          a   nil
```

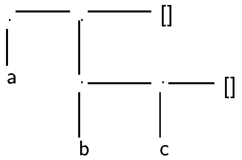
```
.(x, .(y, nil))                        .
                                      / \
                                     x   .
                                        / \
                                       y   nil
```

```
                                       .
                                      / \
.(a, .(.( b, .(c, nil)), nil)        a   .
                                        / \
                                       .   nil
                                      / \
                                     b   .
                                        / \
                                       c   nil
```

Diese Darstellungsweise des Baumdiagrammes läßt sich etwas deformieren, so daß die Listendarstellung intuitiver wird. Der Baum von links nach rechts und die Zweige hängen (wie Weinreben) herunter. In einem solchen „Wein"-Diagramm sieht die Liste `.(x, .(y, nil))` wie folgt aus:

```
.---.---nil
|   |
|   |
x   y
```

Die letzte der drei oben dargestellten Listen sieht als „Wein"-Diagramm folgendermaßen aus:



An diesem Diagramm sieht man, daß jede horizontale Wein-Ebene eine Liste ist, die aus einer bestimmten Anzahl von Elementen besteht. Die oberste Ebene besteht aus einer Liste mit zwei Elementen, von denen eines eine Liste ist. Die zweite Ebene, die aus zwei Elementen besteht, ist das zweite Element der obersten Ebene.

Diese Art der Listendarstellung, die große Ähnlichkeit mit den Datenstrukturen der Programmiersparche LISP hat, ist natürlich (man beachte aber die Syntax) auch in Prolog möglich. Da Listen aber eine sehr allgemeine und oft benötigte Datenstruktur sind, gibt es in den meisten Prolog-Systemen eine eigene Notation für Listen, die tief eingeschachtelte Strukturen vermeidet. In vielen Prolog-

Systemen gibt es neben einer speziellen Notation sogar einen eigenen Datentyp für Listen. Listen lassen sich in einen Kopf und einen Rest aufspalten. Bei der Verwendung des `.`-Funktors war das erste Argument der Kopf und das zweite Argument der Rest einer Liste. In der Standardsyntax tritt an die Stelle des `.`-Funktors die Struktur:

```
[kopf | rest]
```

Die leere Liste repräsentiert das für sich sprechende Symbol:

```
[]
```

Statt der Schreibweise

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

ist die abkürzende Schreibweise

```
[a, b, c]
```

üblich. So werden Listen auch ausgegeben. Obige Listen würden folgendermaßen aussehen:

Weitere Beispiele von Listen in der Prolog-Darstellung:

```
[]
[a, b, c]
[a, b, c | Rest]
[a, b, [c, d]]
[maria, klaus, bernd, [10, 20, 30], claudia]
[maria, klaus, bernd, [10 [20, 30] | X] | R]
[[a] [2] 3 c]
```

Ein häufiger Fehler, der im Umgang mit Listen gemacht wird, ist die Darstellung einer Liste, die aus zwei Elementen `a` und `b` besteht. Der Fehler sieht folgendermaßen aus:

```
[a | b]
```

Diese Notationen liefern zwar einen korrekten Prolog-Term, aber *keine echt Liste*, da sich auf der rechten Seite des `|` keine Liste befindet.

Richtig sind:

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

Eine Aufgabe, die sich häufig im Umgang mit Listen stellt, ist die Verkettung zweier Listen zu einer, d. h. gesucht ist ein Prädikat `append(X,Y,Z)`, das wahr wird, wenn gilt: Z = X verkettet mit Y

Beispiel: Es sei `X = [1 2 3]` und `Y = [4 5 6]`, dann muß gelten `Z = [1 2 3 4 5 6]`.

Die folgende Definition des Prädikates `append` leistet das gewünschte. (Die Richtigkeit der folgenden Regel kann man per vollständiger Induktion zeigen.)

```
append([], L, L).
append([X|R1], Y, [X|R]) :-
  append(R1, Y, R).
```

Es ist nun möglich, beliebige Listen mit obigem Prädikat zu verknüpfen.

```
?- append([1, 2, 3], [4, 5, 6], Z).
   Z = [1, 2, 3, 4, 5, 6]
```

Dabei verfährt das Prädikat nach folgender Methode: Die erste Liste wird elementweise durch Anwendung des zweiten Axioms von `append` verkürzt. Irgendwann ist die Liste die leere Liste `[]`. Dann tritt der erste Fall in Kraft. Dabei wird `Z` mit `[4, 5, 6]` instantiiert. Da sowohl die zweite Liste als auch die dritte Liste im ersten Axiom von `append` denselben Variablennamen besitzen, erhalten sie auch denselben Wert. Danach wird zur nächsthöheren Aufrufebene zurückgesprungen, wodurch vorne an die Liste `R = [4, 5, 6]` die `3` konkateniert (angehängt) wird. Durch den Rücksprung zur nächsten Aufrufebene wird die '2' und schließlich die '1' vorne angehängt, und das Ergebnis mit der verknüpften Liste ausgegeben.

Listen haben gegenüber komplexen Termen bzw. Tupeln den Vorteil, daß sie von beliebiger Länge sind. Jedoch kann man auf ein beliebiges Listenelement nicht direkt zugreifen.

Einfache rekursive Programme

```
ist_liste([]).
ist_liste([X|Rest]) :-
  ist_liste(Rest).

element(X, [X|Rest).
element(X, [Y|Rest]) :-
  element(X, Rest).

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

4.2.1 Reihenfolgeprobleme

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

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

4.2.2 Strukturaufbau bei Rekursionsabstieg bzw. Rekursionsaufstieg

Zählen der Elemente einer Liste

```
anz_element(0, []).
anz_element(N, [X|Rest) :-
  anz_element(NM1, [X|Rest]),
  Tmp is NM1 + 1,
  N == Tmp.

mengen_diff(M, [], M).
mengen_diff(M, [X|Rest], M) :-
  loesche(X, M, M1),
  mengen_diff(M1, Rest, M).
```

4.2.3 Programmierprinzipien

1. Möglichst keine Redundanz! Redundanz führt i. a. dazu, daß die gleiche Lösungen mehrfach gefunden wird (und damit auch mehrfach angezeigt wird).

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

4.3 Übungen

Die Aufgaben dieser Übung sollen ohne die Verwendung von vordefinierten Prädikaten gelöst werden.

1. Definiere das sogenannte „member"-Prädikat. Dieses rekursiv zu definierende Prädikat stellt fest, ob ein Element in einer Liste von Elementen enthalten ist oder nicht. Dabei sind zwei Fälle zu unterscheiden:

(a) Das gesuchte Element ist das 1.Element der Liste von Elementen:

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

(b) Das gesuchte Element ist irgendwo im Rest der Liste:

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

2. Definiere eine Variation des „member"-Prädikates. Dieses Prädikat soll feststellen, ob zwei Elemente konsekutiv in einer Liste von Elementen enthalten sind.

```
?- paarmember(a, b, [a, b, c, d, e]).
   Yes

?- paarmember(c, d, [a, b, c, d, e]).
   Yes

?- paarmember(a, c, [a, b, c, d, e]).
   No
```

3. Die natürlichen Zahlen lassen sich folgendermaßen definieren:

   1 ist eine natürliche Zahl. Ist x eine natürliche Zahl, dann ist $nf(x)$ eine natürliche Zahl. Dabei bezeichne $nf(x)$ den Nachfolger von x.

4. Finde auf der Basis dieser Axiome eine Prolog-Darstellung für die Natürlichen Zahlen, ohne jedoch dabei die eingebauten Zahlen zu benutzen.

5. Schreibe ein Prädikat `nat-zahl`, das wahr ist, wenn das Argument eine so dargestellte Natürliche Zahl ist, bzw. auf die Anfrage

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

   alle Natürlichen Zahlen auflistet.

6. Definiere Prädikate `summe(X, Y, Z)` und `differenz(X, Y, Z)` die wahr werden, wenn `Z` die Summe bzw. Differenz von so dargestellten Zahlen `X` und `Y` ist.

7. Schreibe ein rekursives Prädikat, das aus einer Liste ein Element löscht. Das Prädikat sollte folgendermaßen aufgerufen werden:

```
?- delete(b, [a, b, c], Liste).
   Liste = [a, c]
```

8. In der Linguistik spricht man von einem Minimalpaar, wenn sich ein Wort nur um einen Buchstaben von einem anderen Wort unterscheidet. Folgende Beispiele stellen Minimalpaare dar:

(a) Haus und Laus
(b) Kind und Rind
(c) Rind und Rand

Definiere ein Prädikat `minimalpaar`, das Minimalpaare erzeugen kann. Das Prädikat soll nicht nur das Tauschen eines Anfangsbuchstabens berücksichtigen, sondern auch die Möglichkeit, daß irgendwo in der Mitte des Wortes ein Buchstabe vertauscht werden kann (siehe Beispiel c)). Das Prädikat `minimalpaar`, muß (mindestens) vierstellig werden. Denn man benötigt eine Stelle, um den zu ersetzenden Buchstaben anzugeben, eine Stelle für den Buchstaben, mit dem ersetzt wird, eine Stelle für die Eingabeliste und eine Stelle für die Ausgabeliste.

```
?-  minimalpaar(h, l, [h, a, u, s], X).
    X = [l, a, u, s])
```

9. Definiere ein Prädikat, welches die Reihenfolge der Buchstaben einer Liste umdreht. Entspricht die erzielte Liste der Ausgangsliste, so nennt man dieses Phänomen „Palindrom", z.B. `[r, e, l, i, e, f, p, f, e, i, l, e, r]` oder `[m, a, d, a, m]`

```
?- palindrome( [m, a, d, a, m], X).
   X = [m, a, d, a, m]
```

Definiere ein Prädikat, welches aus einer Liste von Wörtern alle Palindrome herausfiltert.

10. Schreibe ein Prädikat, das aus einer Liste in folgendem Sinne eine „Menge" macht. Die mehrfachen Vorkommen des gleichen Elementes sollen gelöscht werden, so daß jedes Element nur noch einmal vorkommt. Beispiel:

```
?- menge([a, s, d, a, s, w, e, s], M).
   M = [s, e, w, a, d]
```

11. Definiere ein Prädikat, welches bei einer Anfrage alle Möglichkeiten angibt, die entstehen, wenn jedes Element einer beliebigen Liste durch einen angegebenen Term 'u' ersetzt wird.

```
?- ersetze(u, _, [a, b, c], Z).
   Z = [u, b, c];
   Z = [a, u, c];
   Z = [a, b, u];
   No
```

12. Schreibe ein Prädikat, das alle Permutationen zu einer Liste erzeugt. Eine Permutation bedeutet hier eine Liste, die dieselben Elemente enthält wie die Ausgangsliste, lediglich in einer anderen (beliebigen) Reihenfolge. Beispiel:

```
?- perm([a, s, d], P).
   P = [a, s, d];
   P = [a, d, s];
   P = [s, a, d];
   P = [s, d, a];
   P = [d, a, s];
   P = [d, s, a];
   No
```

13. Schreibe ein Prädikat, das alle Zyklen einer Liste erzeugt und dann terminiert. Ein Zyklus ist hier eine Liste, in der die Elemente alle um einen Platz bzw. mehrere Plätze verschoben sind. `[d, a, b, c]` ist ein Zyklus von `[a, b, c, d]` Anmerkung: Die Zyklen bilden also eine Teilmenge der Permutationen.

4.4 Graphen

Eine Verallgemeinerung von Bäumen sind gerichtete Graphen. Im Gegensatz zu Bäumen, dürfen bei Graphen Kanten auch wieder zusammenlaufen.

4.4.1 Ein Beispiel

Abbildung 4.1 zeigt einen einfachen Graphen, der über die Relation `kante` beschrieben wird. In dieser Relation sind alle Paare von Knoten enthalten, zwischen denen es einen Pfeil gibt:

```
kante(a, b).
kante(b, c).
kante(c, d).
kante(c, h).
kante(d, e).
kante(e, f).
kante(g, a).
kante(g, f).
kante(h, g).
```

Ein Weg in einem solchen Graphen ist eine Folge von Pfeilen, die immer Spitze auf Schwanz zusammenstoßen. In diesem Fall, wo es zwischen zwei Knoten höchstens eine Pfeil gibt, wird ein Weg durch eine Folge von Knoten beschrieben (anderenfalls müßte man den Pfeilen Namen geben). Ein Prädikat, daß alle Wege konstruiert, könnte so definiert werden:

```
weg(Anfang,  Anfang, [Anfang]).
weg(Anfang, Ende, [Anfang | Weg]):-
```

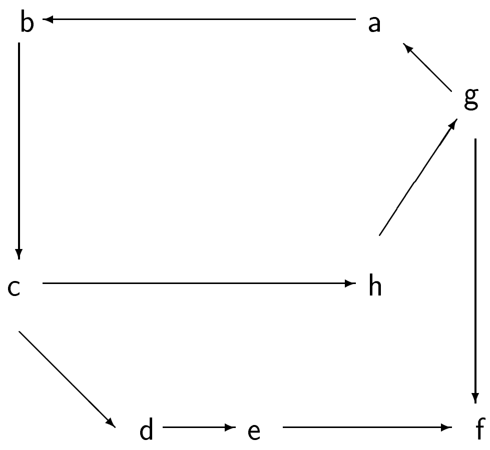


Abbildung 4.1: Ein gerichteter Graph

```
      (Kante, Anfang, Zwischenknoten),
   weg(Zwischenknoten, Ende, Weg).
```

Die erste Klausel liefert Wege der Länge 0, die zweite Klausel zerlegt einen Weg in den ersten Schritt und den Rest. Man beachte aber, daß der Graph eine Zyklus enthält, d. h. es existieren unendlich viele Wege.

Das Problem dieser Graphenrepräsentation liegt darin, daß der Graph nicht als ein Objekt existiert. Wenn Graphen als Objekte benötigt werden, muß man eine andere Darstellung wählen, etwa als Liste von Paaren:

```
[kante(c, h), kante(h, g), kante(g, a), kante(a, b)
 kante(b, c), kante(c, d), kante(d, e), kante(e, f)
 kante(g, f)]
```

Das Prädikat `weg` erhält dann folgende Definition:

```
weg(Anfang, Anfang, Graph, [Anfang]).
weg(Anfang, Ende, Graph, [Anfang|Weg]):-
  member(kante(Anfang, Zwischenknoten), Graph),
  weg(Zwischenknoten, Ende, Graph, Weg).
```

Wenn ein Graph jedoch Zyklen enthält, kann ich auch diese ganz einfach abfangen. So kann ich zum Beispiel bei dem folgenden Graphen den Weg von a nach f mit Hilfe von weg2 finden:

```
[kante(a, b), kante(a, c), kante(b, d), kante(b, f), kante(c, e), kante(c, h),
 kante(c, i), kante(d, e), kante(d, c), kante(e, g), kante(e, h), kante(f, d),
```

```
kante(i, j), kante(i, k), kante(h, e), kante(h, l), kante(j, l), kante(k, l),
kante(l,h)]
```

Um feststellen zu können, ob ich einen Knoten schon besucht habe, führe ich eine List mit den schon besuchten Konten mit.

```
  weg2(Start,Start,_).

weg2(Start,Ziel,_):-
  member(kante(Start,Ziel),Graph).

weg2(Start,Ziel,SchonBesucht):-
  member(kante(Start,Zwischen),Graph).
  not(member(Zwischen,SchonBesucht)),
  weg2(Zwischen,Ziel,[Start|SchonBesucht]).

weg2(Start,Ziel):-
  weg2(Sart,Ziel,[]).
```

Diese letzte sehr kompliziert anmutende Repräsentationsweise für gerichtete azykliche Graphen demonstriert, daß man Prolog-Terme von ihrer innersten Struktur her genau als solche Graphen auffassen kann.

4.5 Differenzlisten

Bei der Verwendung von großen Listen stellt man sehr schnell fest, daß Prädikate, die nicht auf den Kopf bzw. den Rest einer Liste zugreifen, sehr viel Rechenzeit benötigen. Ein Beispiel dafür ist `append`. Denn um zwei Listen miteinander zu verknüpfen, ist es nötig, daß von einer Liste Element für Element abgespalten wird, bis schließlich die Liste leer ist, um anschließend die Elemente an die andere Liste vorne anzuhängen. Möchte man ein derartiges Problem effizienter lösen, so bietet sich häufig die Verwendung von Differenzlisten an. Eine Differenzliste kann man sich als eine Liste, bestehend aus zwei Listen, vorstellen, wobei die Differenz dieser beiden Listen die eigentliche Differenzliste darstellt. Wir wollen uns den Begriff der Differenzliste anhand der Liste `[a, b, c]` verdeutlichen. Folgende Darstellungsweisen sind für diese Liste möglich (so sieht die Darstellung von Differenzlisten in Prolog allerdings nicht aus): Die zweite Liste muß jeweils ein Rest der oberen Liste sein. Deshalb stellen die obigen Listen alle dieselbe Liste, nämlich [a b c], dar. Auf diese Weise gibt es also unendlich viele Möglichkeiten ein und dieselbe Liste darzustellen. Haben wir nun noch eine weitere Liste `[1, 2, 3]` gegeben, so könnte man diese folgendermaßen darstellen:

```
[a, b, c, d, e, f]
         [d, e, f]
```

```
[a, b, c, d]
         [d]
```

```
[a, b, c]
       []
```

Abbildung 4.2: Alternative Differenzlistendarstellungen

```
[1, 2, 3]
       []
```

Abbildung 4.3: Eine minimale Differenzliste

Um nun zwei Differenzlisten zu verketten, wählt man eine Darstellung, bei der der zweite Teil (also die Restliste) der ersten Differenzliste dem ersten Teil der zweiten Differenzliste entspricht. Dann bildet der erste Teil der ersten Differenzliste und der zweite Teil der zweiten Differenzliste das Ergebnis der Verkettung der beiden Differenzlisten. Die beiden Diffenrenzlisten aus Abbildung 4.4 passen, so wie dort dargestellt, nicht zusammen, denn `[]` von `dliste_1` ist nicht identisch mit `[1, 2, 3]` von `dliste_2`. Ändert man aber die Darstellung der ersten Liste wie folgt, so lassen sich die beiden Differenzlisten sehr leicht verketten: Der Trick dabei ist also, daß die Restliste variabel ist. Man wählt die Restliste der `dliste_1` so, daß sie der ersten Liste von `dliste_2` entspricht. Denn Liste `dliste_1` repräsentiert nun immer noch die Liste `[a b c]`, allerdings lassen sich nun die beiden Differenzlisten sehr leicht miteinander verketten. Denn der zweite Teil von `dliste_1` entspricht dem ersten Teil von `dliste_2`.

```
dliste_1          dliste_2

[a, b, c]         [1, 2, 3]
       []                []
```

Abbildung 4.4: Unverträgliche Differenzlisten

Gesucht ist nun ein Prädikat, mit dem man zwei Differenzlisten verknüpfen kann, also ein Prädikat mit folgendem Kopf:

```
dappend(dl(X1, X2),dl(Y1, Y2),dl(Z1, Z2)).
          /     |     /     \     |
          Liste_1    Liste_2    \Verknüpfung der beiden Listen
```

Zur Verkettung sind nur die oben beschriebenen Unifikationen durchzuführen:

```
Z1 = X1
Z2 = Y2
X2 = Y1
```

Damit ergibt sich:

```
dappend(dl(Z1, X), dl(X, Z2), dl(Z1, Z2)).
```

In praktischen Anwendungen nun tritt jedoch so ein Prädikat nicht explizit auf, sondern diese Gleichsetzungen (Identifikationen) werden in die Programmstruktur mit aufgenommen. Häufig werden die Komponenten der Differenzlisten nichteinmal zu einem Tupel zusammengefaßt, so daß es schwierig sein kann, Differenzlisten überhaupt in einem Programm zu erkennen.

Wir hatten gesehen, daß es zu einer Liste beliebig viele Differenzlistendarstellungen gibt. Differenzlisten wären bei weitem nicht so interessant, wenn es nicht auch eine eindeutig bestimmte allgemeinste Darstellung geben würde, aus der alle anderen sich durch Instantiierung ergeben. In dieser allgemeinsten Darstellung ist die zweite Komponente eine freie Variable: Differenzlisten haben in Prolog insbesondere bei der Programmierung von Grammatiken eine wichtige Bedeutung.

$$\texttt{dliste\_1} \qquad\qquad \texttt{dliste\_2}$$

$$\begin{bmatrix} \texttt{[a, b, c |R1]} \\ \hfill\texttt{R1} \end{bmatrix} \qquad \begin{bmatrix} \texttt{[1, 2, 3 |R2]} \\ \hfill\texttt{R2} \end{bmatrix}$$

$$\texttt{dliste\_3}$$

$$\begin{bmatrix} \texttt{[a, b, c, 1, 2, 3|R2]} \\ \hfill\texttt{R2} \end{bmatrix}$$

Abbildung 4.5: Die allgemeinste Darstellung von Differenzlisten

Kapitel 5

Grammatiken

Zur Beschreibung formaler aber natürlicher Sprachen wurden die unterschiedlichsten Grammatikformalismen entwickelt. Die grundlegenden Arbeiten hierzu stammen von Chomsky Ende der fünfziger Jahre. Formale Grammatiken werden üblichrweise als Produktionssysteme (Ersetzungsregen) formuliert.

An eine Grammatik einer Sprache werden folgende Anforderungen gestellt:

- Die Sätze der Sprache sollen von der Grammmatik erkannt werden.
- Die Grammatik soll in der Lage sein, alle grammatikalisch richtige Sätze zu erzeugen.

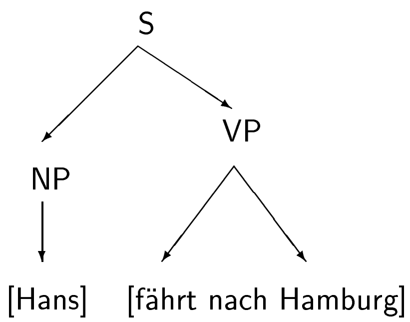
Beim Analysiere von Sätzen (parsen) soll den Sätzen u.U. eine Bedeutung zugeordnet werden.

Im Kontext eines Dialogsystems werden z.B. natürlichsprachliche Anfragen in eine Wissensrepräsentationssprache übersetzt, eine Antwort berechnet und diese Antwort wird wieder zurück in die natürliche Sprache übersetzt.

Wir werden hier ein (sehr) kleines Dialogsystem entwickeln und der Einfachheit halber als Wissensrepräsentationssprache Prolog selber verwenden.

5.1 Eine klitzekleine Grammatik

Unsere kleine Grammatik soll einen Satz (S) bestehend aus einer Nominalphrase (NP) und einer Verbalphrase (VP) erkennen (parsen) bzw. erzeugen (generieren) können. Ein Satz soll folgende Struktur besitzen:



Dazu benötigen wir etwa die folgenden Grammatikregeln:

```
S  --> NP VP
VP --> V PP
PP --> P NP
NP --> [Hans]
NP --> [Hamburg]
V  --> [fährt]
P  --> [nach]
```

Um die terminale Symbole (Wörter) von den nichtterminalen Symbolen (Kategorien) unterscheiden zu können, setzen wir die terminalen Symbole in eckige Klammern. Dies hat praktische Konsequenzen, denn wir können diese Ausdrücke später als Listen in Prolog interpretieren.

Prolog eignet sich sehr gut für die Programmierung von Grammatiken. Es wurde ursprünglich eben zu diesem Zweck entwickelt. Es ist in den meisten Prolog-Systemen möglich, die obige Grammatik wie dort notiert einzugeben. Um aber zu verstehen, was passiert, machen wir einen kleinen Umweg, indem wir zunächst zeigen, daß Grammatikregeln etwas mit Prolog-Klauseln zu tun haben.

Die Regel

```
S --> NP VP
```

können wir uminterpretieren, indem wir die nichtterminalen Symbole als Prädikate über Wortlisten betrachten.

```
s(X) :-
  np(Y),
  vp(Z),
  append(Y, Z, X).
```

Eine Liste `X` stellt einen Satz dar, falls es eine Nominalphrase `Y` gibt und es eine Verbalphrase `Z` gibt und `Y` verkettet mit `Z` gerade `Z` ist.

Diese Definition eignet sich sowohl zum Generieren, als auch zum Analysieren von Sätzen. Wendet man dieses Axiom an, um zu prüfen, ob eine angegebene

Liste im Sinne unserer Definition einen Satz darstellt, so würde die Definition folgendermaßen zu lesen sein:

Eine Liste `X` stellt einen Satz dar, falls es eine Nominalphrase `Y` gibt und es eine Verbalphrase `Z` gibt und sich die Liste `X` in `Y` und `Z` spalten läßt.

Terminale Grammatikregeln haben die einfache Form:

```
np([hans]).
```

Die komplette Grammatik als Prolog-Programm:

```
   s(x) :-
     np(Y),
     vp(Z),
     append(Y, Z, X).

   vp(X) :-
     v(Y),
     pp(Z),
     append(Y, Z, X).

   pp(X) :-
     p(Y),
     np(Z),
     append(Y, Z, X).

   np([hans]).
   np([hamburg]).

   v([faehrt]).

   p([nach]).

?- s(X).
   X = [hans, faehrt, nach, hans] ;
   X = [hans, faehrt, nach, hamburg] ;
   X = [hamburg, faehrt, nach, hans] ;
   X = [hamburg, faehrt, nach, hamburg] ;
   No.
```

Die Grammatik erzeugt also neben dem intendierten Satz "Hans fährt nach Hamburg" auch einige Sätze, die zwar syntaktisch nicht falsch, semantisch aber zumindest merkwürdig sind. Wie man dies verhindern kann, wollen wir noch etwas zurückstellen.

Das obige Programm ist nicht sehr effizient, u.a., weil die Verwendung des *append* recht viel Rechenzeit kostet. Erst recht dann, wenn die Listen, die verkettet, bzw. gespalten werden sollen, lang sind. Um nun das *append* aus der obigen Definition loszuwerden, verwendet man Differenzlisten.

Wir übersetzen als Beispiel das erste Axiom des obigen Programms in eine Version mit Differenzlisten. Dabei wird eine Variable durch ein Tupel bestehend aus zwei Komponenten ersetzt, denn Differenzlisten bestehen ja aus zwei Teilen:

```
s(X1, X2) :-
  np(Y1, Y2),
  vp(Z1, Z2),
  dappend(dl(X1, X2), dl(Y1, Y2), dl(Z1, Z2)).
```

Da aber nun `dappend` nichts weiter macht, als einige dieser Variablen zu unifizieren, können wir `dappend` weglassen und die entsprechenden Variablen miteinander identifizieren. Wir erhalten:

```
s(var(X1, X2)) :-
  np(dl(X1, X2)),
  vp(dl(Y1, Y2)).
```

Diese Klausel hat folgende mehr oder weniger anschauliche Interpretation:

`X1` beginnt mit einem Satz und hat als Rest `X2`, falls `X1` mit einer Nominalphrase beginnt, der verbleibende Rest `Y2` mit einer Verbalphrase beginnt und der verbleibende Rest von `Y2` gerade `X2` ist.

Das oben angesprochene Problem, semantisch nicht wohlgeformte Sätze auszuschließen, läßt sich durch Einführen weiterer Argumente, die z.B. semantische Informationen enthalten, lösen. Insgesamt ergibt sich dann folgende Grammatik:

```
s(dl(X1, X2)) :-
  np(dl(X1, Y), Sorte),
  vp(dl(Y, X2), Sorte).

vp(dl(X1, X2), Sorte) :-
  v(dl(X1, Y), Sorte),
  pp(dl(y, X2), Sorte).

pp(dl(X1, X2), Sorte) :-
  p(dl(X1, Y), Sorte, Sorte2),
  np(dl(Y, X2), Sorte2).
```

5.2 DCG-Grammatiken

Der hier beschriebene Grammatiktyp ist unter dem Namen *Definite-Clause-Grammar* (DCG) bekannt und steht in nahezu jedem Prolog-System mit einer eigenen Syntax zur Verfügung, so daß man sich zumidest bei der Regeleingabe nicht um die Differenzlisten zu kümmern braucht.

Die Grammatik des letzten Abschnittes hat dann die Form:

```
s           --> np(Sorte), vp(Sorte).

vp(Sorte)  --> v(Sorte, Sorte2), pp(Sorte2).

pp(Sorte)  --> p(Sorte, Sorte2), np(Sorte2).

np(mensch) --> [hans].
np(stadt)  --> [hamburg].

v(mensch, ziel) --> [faehrt].
p(ziel, stadt)  --> [nach].
```

Die Anfrage muß aber wie im letzten Abschnitt gestellt werden.

5.2.1 Übungen

1. Schreibe ein Prolog-Programm, welches unter anderem die folgenden Sätze erzeugt und erkennt:
   - „Peter liebt Maria."
   - „Die Männer lieben die Frauen."
   - „Das Kind kennt den Clown."
   - „Die Kinder kennen die Tänzerin."
2. Erstelle auf Grundlage von 5.8.1 eine Grammatik, welche folgende Restriktionen erfüllt:

   Verb und Subjekt sowie Artikel und Nomen passen in Kasus, Genus und Numerus zusammen!

Kapitel 6

Arithmetik

Für viele Anwendungen ist es notwendig mit Zahlen umzugehen. Dazu stellt Prolog eine Reihe von arithmetischen Operationen zur Verfügung, die wir uns jetzt der Reihe nach ansehen wollen. Einige der vordefinierten Operatoren könne für grundlegende arithmetische Operationen verwendet werden.

`+`	Addition
`-`	Subtraktion
`*`	Multiplikation
`/`	Division
`//`	Ganzzahlige Division
`mor`	Rest der Ganzahligen Division (Modulo)

Dabei ist zu beachten, daß es eine Ausnahme ist, daß Prolog tatsächlich bei diesen Opeartoren berechnungen vereanlaßt. Aber selbst bei diesen Operatoren genügt es nicht, daß Prolog weiß, wie die Operatoern ausgewertet werden sollen. Die Ausswertung muß erst noch angestoßen werden.

```
?- X = 1 + 2.
```

Diese Anfrage wertet das System nicht aus, sondern antwortet

```
X = 1 + 2.
```

und nicht $X = 3$, wie man vielleichterwartet hätte. Der Ausdruck $X = 1 + 2$ ist ein Term, in dem + der Funktor und 1 und 2 die beiden Argumente sind. In dem oberigen Term gibt es nichts, was das Prologsystem veranlaßt, den Term auszuwerten. Um dies aber trotzdem anzuregen, gibt es einen vordefinierten operator, is, der die Auswertung erzwingt. Der richtige Aufruf zur Berechung von X ist also:

```
?- X is 1 + 2.
```

Dieser Ausdruck wird dann ausgewertet zu:

```
X = 3
```

Das linke Argument des is-Operators ist ein einfaches Argument. Das rechte Argument ist ein arithmetischer Ausdruck, der aus arithmetischen Operatoren, Zahlen und Variabelen zusammengesetzt seien kann. Da der is-Operator eine Auswertung erzwingt, darf es zum Zeitpunkt der Auswertung keine ungebundenen Variabelen mehr geben. Der Vorrang der Argumente der operatoren ist so gewählt, daß die Assozivität der Argumente die gleiche ist wie in der Mathematik. Beachtet werden muß, das +, -, *, /,.. von links nach rechts ausgewertet werden. So wird zum Beispiel

```
X is 8 - 4 - 2.
```

interpretiert wie

```
X is (8 - 4) - 2.
```

Zur Arithmetik gehören auch die Vergleichsoperatoren. Ähnlich wie der is-Operator erzwingen die Vergleichsoperatoren eine Auswertung.

```
?- 2 * 2 > 3.

yes
```

Vergleichsoperatoren in Prolog sind:

`X > Y`	X ist größer als y
`X < Y`	X ist kleiner als Y
`X >= Y`	X ist größer oder gleich Y
`X =< Y`	X ist kleiner oder gleich Y

Wir wollen nun den Gebrauch von Operatoren an einem Beispiel deutlich machen. Es seien Zwei Zahlen X und Y gegeben. bei der Berechnung ihres ggT T seien drei Fälle unterschieden.

1. Sind X und Y gleich, das ist T gleich X.

2. Ist X kleiner als Y, dann ist T gleich dem größten gemeinsamen Teiler von X und der Differenz Y - X.

3. Ist X größer als Y, dann ist T gleich dem größten gemeinsamen Teiler von X und Y, wobei X und Y vertauscht werden.

Diese drei Regeln können leicht durch drei Klauseln ausgedrückt werden.

```
ggt(X, X, X).

ggt(X, Y, T):-
  X < Y,
  Y1 is Y - X,
  ggt(X, Y1, T).

ggt(X, Y, T):-
  X > Y,
  ggt(Y, X, T).
```

6.1 Übungen

1. Schreibe ein Prädikat, das die Anzahl der Elemente einer Liste ermittelt.
2. Schreibe ein Prädikat, das das i-te Element einer Liste ausgibt.
3. Schreibe ein Prädikat, welches die Fibonaccizahlen berechnet. Die Fibonaccizahl von 0 ist 1, von 1 ist ebenfalls 1 und allgemein ergibt sich eine Fibonaccizahl aus der Summe ihrer beiden unmittelbaren Vorgänger. Die Fibonaccizahlen lauten also: 1,1,2,3,5,8,13,21,34,55,89,...
4. Denke an ein Dominospiel. Stelle Dir eine Reihe von aneinandergelegten Dominosteinen vor, z.B.

```
+---+---++---+---++---+---++---+---++---+---+
| 5 | 2 || 3 | 1 || 5 | 1 || 3 | 2 || 1 | 2 |
+---+---++---+---++---+---++---+---++---+---+
```

5. Finde eine Prolog-Darstellung, die eine solche Reihe von Dominosteinen in geeigneter Weise repräsentiert.
6. Eine Reihe ist korrekt gelegt, wenn die Steine so gelegt sind, daß zwei Steine an den Seiten, an denen sie sich berühren, denselben Zahlenwert haben. Schreibe ein Prädikat, welches die Korrektheit einer solchen gegebenen Reihe testet.
7. Schreibe ein Prädikat, das das Umdrehen eines Steines durchführt.
8. Erstelle ein Prädikat, welches zu einer beliebigen Reihe von Dominosteinen alle möglichen Permutationen erzeugt. Beachte hierbei, daß sich eine Permutation nicht nur aus der Vertauschung zweier Steine, sondern auch schon dadurch ergibt, daß ein Stein gedreht wird. Wie erhälst man aus einer beliebigen Reihe die Permutationen, die eine im obigen Sinne korrekte Reihe darstellen. Verwende zum Testen der Prädikate die oben angegebene Reihe.

9. Denke Dir ein Sortierverfahren, das von der Idee her ähnlich arbeitet wie das Dominospiel, d.h. erzeuge zu einer gegebenen Liste von Zahlen alle Permutationen und teste, ob die Permutation, die gerade erzeugt worden ist, sortiert ist. Schreibe ein Prädikat für das obige Sortierverfahren.

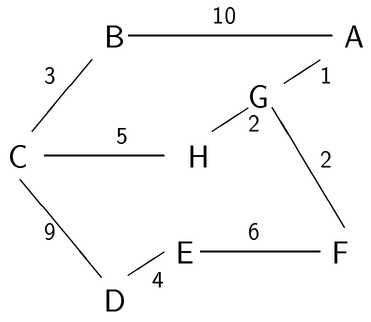
10. Schreibe ein Prädikat für den Bubblesort. Der Bubblesort arbeitet nach dem folgenden Prinzip: Man denke sich die Zahlen als unterschiedlich große Blasen, größere Zahlen sind größere Blasen. Die größte Blase steigt am schnellsten nach oben, dann die nächst größere u.s.w. Auf diese Weise wird also die Liste sortiert. Beipiel: Die mit '$\frac{1}{4}$-' gekennzeichneten Zahlen werden mit ihrem Vorgänger verglichen. Falls sie größer ist als ihr Vorgänger, werden die beiden Zahlen getauscht, sonst nicht. In jedem Fall wird der „Zeiger" um eins hochgerückt, d.h. die nächst höhere Position der Liste betrachtet. Bei den nächsten Schritten wird entsprechend verfahren.

11. Der Quicksort-Algorithmus funktioniert folgendermaßen: Gegeben sei die Liste [3 7 1 4 2]. Wähle ein Element der Liste aus (in Prolog ist es am einfachsten das erste Element der Liste zu nehmen). In diesem Falle also die 3. Teile die Liste in zwei Teillisten. In die eine Liste packe die Elemente, die kleiner sind als 3 und in die andere die Liste die Elemente, die größer oder gleich 3 sind. So erhält man also [1 2] und [3 7 4]. Nun verfahre man rekursiv genauso mit den neu entstandenen Listen.

12. Schreibe ein Prädikat, das den Quicksort-Algorithmus darstellt, ohne Differenzlisten zu verwenden.

13. Schreibe ein Prädikat, das nochmals den Quicksort-Algorithmus darstellt, jedoch diesmal unter Verwendung von Differenzlisten.

14. Betrachte den folgenden ungerichteten Graphen:



15. Die Buchstaben seien Städte und die Zahlen an den Graphen seien die Entfernungen in Kilometern zwischen den beiden entsprechenden Städten. Wie kann man einen solchen Graphen in Prolog repräsentieren.

16. Schreibe ein Prädikat, welches alle Wege inklusive der Entfernung zwischen zwei Städten ausgibt. Dabei ist darauf zu achten, daß kein Zyklus durchlaufen wird, bzw. sich das Programm nicht in einem Zyklus verfängt.

17. Schreibe ein Prädikat, welches den kürzesten Weg zwischen zwei Städten ausgibt.

18. Ein Musikverein möchte bei einem Umzug durch einen Torbogen marschieren. Doch der Dirigent, der voran geht, stellt fest, daß die Kapelle, die in quadratischer Formation marschiert, so nicht durch den Torbogen paßt. Er beschließt daraufhin, die Gruppe in drei kleinere ebenfalls quadratische Gruppen aufzuteilen. Die Anzahl der Musikanten der Kapelle ist übrigens die kleinste Zahl, die sich aus der Addition von drei Quadratzahlen ergibt. Aus wievielen Musikanten besteht die gesamte Kapelle und wie teilt der Dirigent die Kapelle vor dem Torbogen auf? Erstelle ein Prolog-Programm, welches diese Probleme löst.

Kapitel 7

Arbeitsweise des Prolog-Interpreters

Wie der Interpreter vorgeht, bei dem Versuch, Anfragen zu beantworten, soll im nächsten Abschnitt geklärt werden. Die wichtigsten Mechanismen dabei sind die Unifikation und das Backtracking.

Wenn vom System eine Anfrage beantwortet werden soll, so sucht es in der Wissensbasis nach einem Faktum oder dem Kopf einer Regel, die mit dieser *unifiziert* (verschmilzt). Eine Anfrage und ein Klauselkopf (Faktum oder Kopf der Regel) unifizieren, wenn es möglich ist, Variablen so zu ersetzen, daß beide Ausdrücke gleich werden.

Ein Beispiel: Befindet sich die Verwandtschaftsbeziehungen aus 2. in der Wissensbasis und die Anfrage lautet,

```
?- mutter(M, jens).
```

so wird die Wissensbasis von vorne nach hinten durchsucht, bis eine Klausel gefunden wird, deren Kopf das zweistellige Prädikat `mutter` und dessen zweites Argument `jens` oder eine noch nicht belegte Variable ist, kurz gesagt, dessen zweites Argument mit `jens` unifiziert. Befindet sich an der zweiten Stelle des Kopfes der Klausel eine Variable, so wird jedes weitere Vorkommen dieser Variablen in der Klausel mit `jens` instantiiert. Diese Anfrage ist bezüglich des obigen Beispieles nur mit `mutter(maria, jens)` unifizierbar, die Variable `M` muß also durch den Wert `maria` ersetzt werden.

Wir betrachten folgendes Prolog-Programm:

```
mutter(anke, maria).
mutter(maria, jens).

großmutter(G, E) :-
```

```
    mutter(G, X),
    mutter(X, E).
```

(G ist die Großmutter von E, wenn G die Mutter von einer X ist, und diese die Mutter von E) Stellen wir nun die Anfrage:

```
?- grossmutter(M, jens).
```

Der Kopf der Regel `grossmutter` ist mit der Anfrage unifizierbar. Wir setzen dazu für `E` den Wert `jens` ein und die Variablen `G` und `M` werden gleichwertig, d.h. erhält eine der Variablen später einen Wert, so bekommt diesen automatisch auch die andere Variable. Als spezielle Information erhalten wir nun:

```
grossmutter(G, jens) :-
  mutter(G, X),
  mutter(X, jens).
```

Nun wird vom Prolog-System versucht, die neu entstandenen Teilziele (Subgoals) `mutter(G, X)` und `mutter(X, jens)` zu beweisen und zwar in der angegebenen Reihenfolge. Dabei unifiziert `mutter(G, X)` mit dem Kopf des Faktums `mutter(anke, maria)`. Dadurch erhält `G` den Wert `anke` und `X` den Wert `maria`. Da die Wertzuweisung für jedes Vorkommen der Variablen in einer Klausel gilt, folgt unter anderem, daß für das zweite Teilziel (Subgoal) nun `mutter(maria, jens)` bewiesen werden muß. Aber dieses steht ja als Faktum in der Wissensbasis. Also ist der Beweis erfolgreich abgeschlossen worden, und das System gibt folgendes aus:

```
grossmutter(anke, jens).
```

7.1 Aussagen in Normalform

Die Prädikatenlogik gestattet es, den gleichen Sachverhalt durch verschiedene aber logisch äquivalente Aussage auszudrücken. Dies erschwert die Beweiskonstruktion, die ja auf den syntaktischen Strukturen operiert. Unser Ziel ist es daher zunächst, beliebige logische Aussagen auf eine besonders einfache syntaktische Struktur zu bringen, ohne dabei ihre Wahrheitsbedingungen zu ändern.

In diesem Abschnitt wird kurz skizziert, wie beliebige Formeln der Prädikatenlogik in eine spezielle Normalform, die Klauselform, überführt werden.

Eine atomare Formel (elementare Formel) ist ein Prädikatsname gefolgt von seinen Argumenten.

Ein *Literal* ist eine atomare Formel (positives Literal) oder das Negat einer atomaren Formel (negatives Literal), z.B. $\neg mutter(a, b)$.

Eine *Klausel* ist ein Literal oder die Disjunktion (Oder-Verknüpfung) mehrerer Literale, in denen alle Variablen durch Allquantoren gebunden sind; z.B. ist

$$\forall X_1 : \forall X_2 : \ldots \forall X_k : (\neg A1 \vee \neg A2 \vee \ldots \vee \neg A_n \vee B_1 \vee \ldots \vee B_m) \tag{7.1}$$

eine Klausel, wenn $X_1, \ldots, X_k$ alle in den Literalen $\neg A_1, \ldots, \neg A_n, B_1, \ldots, Bm$ vorkommenden Variablen sind. Man möchte erst einmal jede Aussage der Prädikatenlogik 1.Stufe auf folgende Normalform bringen:

$$Quantorenpraefix : Klausel_1 \odot Klausel_2 \odot \ldots \odot Klausel_n \tag{7.2}$$

Solch eine Form wird auch Pränexnormalform genannt. Sie kann aus jeder prädikatenlogischen Formel durch äquivalente Umformungen der folgenden Art erreicht werden.

1. Schritt: $\Leftrightarrow, \Rightarrow$ eliminieren

$$\begin{aligned} FG &\Rightarrow F \\ FG &\Rightarrow \neg FG \end{aligned}$$

2. Schritt: Gültigkeit der Negationszeichen wird verkleinert

$$\begin{aligned} \neg(FG) &\Rightarrow \neg F \neg G \\ \neg(FG) &\Rightarrow \neg F \neg G \\ \neg\neg F &\Rightarrow F \\ \neg(\forall X : F) &\Rightarrow \exists X : \neg F \\ \neg(\exists X : F) &\Rightarrow \forall X : \neg F \end{aligned}$$

3. Schritt: Quantoren($\forall, \exists$) nach vorne ziehen. Die Variable Z komme in A und B nicht vor; Q ist aus $\forall, \exists$.

$$\begin{aligned} \forall X : A(X) \forall X : B(X) &\Rightarrow \forall X : (A(X)B(X)) \\ QX : A(X)B &\Rightarrow QZ : (A(Z)B) \\ QX : A(X)B &\Rightarrow QZ : (A(Z)B) \\ AQX : B(X) &\Rightarrow QZ : (AB(Z)) \\ AQX : B(X) &\Rightarrow QZ : (AB(Z)) \end{aligned}$$

4. Schritt: Den quantorenfreien Ausdruck als Konjunktion von Disjunktionen von Literalen darstellen

$$\begin{aligned} (AB)C &\Rightarrow (AB)(BC) \\ A(BC) &\Rightarrow (AB)(AC) \end{aligned}$$

Um nun aus der durch äquivalente Umformungen entstandenen Pränexnormalform zur Klauselform zu gelangen, müssen noch die Existenzquantoren beseitigt werden. Dies geht nicht mehr mittels äquivalenter Umformungen, sondern durch die Einführung von Skolemfunktionen. Durch sie lassen sich die Existenzquantoren folgendermaßen eliminieren:

> Eine existenzquantifizierte Variable X wird durch eine neue (noch nicht verwendete) Funktion ersetzt. Diese Funktion hängt von allen allquantifizierten Variablen ab, in deren Geltungsbereich sich X befindet.

Auf diese Weise erhält man die oben beschriebene Klauselform. Nun kann man per Konvention alle Allquantoren weglassen, wenn man alle freien Variablen als implizit allquantifiziert ansieht. Man bezeichnet eine Variable in einer Formel als frei, wenn sie nicht im Wirkungsbereich eines Quantors steht.

Beispiel: Umformung einer prädikatenlogischen Aussage in Klauselform mit den zuvor beschriebenen Schritten.

$$\forall x : (q(x) \exists y : \forall z : (q1(y,z) q2(y,z)) \tag{7.3}$$
$$\forall x : (q(x) \neg\exists y : \forall z : (q1(y,z) q2(y,z)) \tag{7.4}$$
$$\forall x : (q(x) \forall y : \neg\forall z : (q1(y,z) q2(y,z)) \tag{7.5}$$
$$\forall x : \forall y : (q(x) \exists z : \neg(q1(y,z) q2(y,z)) \tag{7.6}$$
$$\forall x : \forall y : \exists z : (q(x) \neg q1(y,z) \neg q2(y,z)) \tag{7.7}$$
$$\forall x : \forall y : (q(x) \neg q1(y, f(x,y)) \neg q2(y, f(x,y))) \tag{7.8}$$
$$(q(x) \neg q1(y, f(x,y)) \neg q2(y, f(x,y))) \tag{7.9}$$
$$\tag{7.10}$$

7.2 Resolution

Für Axiomensysteme dieser Form gibt es ein sehr effizientes Beweisverfahren: Das *Resolutionsverfahren* von Robinson (1965). Dieses Verfahren beruht auf der *indirekten Beweisführung* (Widerlegungsverfahren).

Es soll gezeigt werden, daß eine Aussage A aus einem Axiomensystem ϕ ableitbar ist.

- Voraussetzung: Das Axiomensystem ϕ ist widerspruchsfrei.
- Die zu beweisende Aussage A wird negiert, in Klauselform übersetzt und als neues Axiom hinzugenommen.
- Läßt sich nun ein *Widerspruch* aus $\neg A$ ableiten, so ist die ursprüngliche Aussage A wahr (d. h. aus ϕ ableitbar).

Der Widerspruch erfolgt nach folgender Ableitungsregel:

$$(A \wedge \neg B) \vee (B \wedge C) \rightarrow (A \wedge C) \tag{7.11}$$

D.h. Tritt die gleiche Aussage in einer Klausel negiert und in einer anderen Klausel nicht negiert auf, so kann eine neue Klausel aus den restlichen Literalen dieser Klauseln gebildet werden. Gelingt es, die leere Klausel abzuleiten, so ist der Widerspruch gezeigt, denn die leere Klausel ist nur ableitbar aus Klauseln der Form ¬B und B. Die nichtnegierten Literale werden auch als positive Literale und die negierten Literale als negative Literale bezeichnet.

In Prolog sind nun nicht beliebige Klauseln zugelassen, sondern nur solche, die höchstens ein positives Literal besitzen. Klauseln dieser Form heißen *Hornklauseln*.

7.3 Hornklauseln

Prolog verwendet eine Untermenge der prädikatenlogischen Ausdrükke, die sogenannten Hornklauseln. Diese Hornklausellogik hat den Vorteil, daß sie entscheidbar ist und in der Praxis an Mächtigkeit ausreicht. Die Hornklausellogik ist eine Einschränkung der Prädikatenlogik, da eine Hornklausel nur ein positives Literal enthält. Eine Hornklausel ist also von der folgenden Form:

$$\begin{array}{ll} & C, \neg P_1, \ldots, \neg P_n \quad (7.12) \\ \textit{oder} & C \quad (7.13) \\ \textit{oder} & \neg P_1, \ldots, \neg P_n \quad (7.14) \\ & \quad (7.15) \end{array}$$

1. Der erste Fall entspricht der Implikation (oder Regel)

   $$C \leftarrow P_1, \ldots, P_n. \tag{7.16}$$

   Das positive Literal ist die Konklusion, und die negativen Literale sind die Prämissen.

2. Der zweite Fall entspricht dem elementaren Fakt C.

3. Der dritte Fall entspricht in Prolog den Anfragen:

   ```
   ?- B1, ... ,Bn.
   ```

Wichtig ist, daß C und alle P_i elementare Aussagen sind.

Weitergehende Informationen zu diesem Kapitel inbesondere und zu den Grundlagen von Prolog allgemein, finden sich beispielsweise in (Sperschneider, Antoniou 1991).

7.4 Unifikation

Die Wissensbasis in Prolog besteht aus Axiomen, die nur ein positives Literal haben. Diese Axiome lassen sich nach dem Prädikatsnamen des positiven Literals (der Konklusion) zusammenfassen. Die zu einem Prädikatsnamen gehörende Menge von Axiomen fassen wir als Definition des Prädikates auf. Zu einer gegebenen negierten elementaren Aussage läßt sich nun sehr einfach testen, ob ein Axiom existiert, das diese Aussage nicht negiert enthält (also als Konklusion in der Implikation).

Aus Axiomen, die freie Variablen enthalten, können neue Axiome gewonnen werden, indem Variablen umbenannt oder einige dieser Variablen mit Werten belegt werden. Das bedeutet dann den Übergang zu einer spezielleren Aussage, da freie Variablen implizit allquantifiziert sind. Entsprechend können in einer zu beweisenden Aussage, die ja negiert als Axiom betrachtet wird, freie und daher implizit allquantifizierte Variablen vorkommen und mit Werten belegt werden.

Für den Beweis einer Aussage kommen also nicht nur Axiome in Frage, die exakt diese Aussage als positives Literal besitzen, sondern auch solche Axiome, deren positives Literal durch Belegen von freien Variablen in dem Axiom und in der zu beweisenden Aussage mit dieser Aussage identisch gemacht werden können.

Dieser Vorgang heißt Unifikation und ist das Herzstück der meisten automatischen Beweiser. Die Belegungsvorschrift für die Variablen heißt *Unifikator*. Dieser Unifikator kann als eine Liste von Paaren, bestehend aus einem Variablennamen und einem Wert (der auch wieder Variablen enthalten kann), aufgefaßt werden. Eine interessante Frage ist nun, ob es mehrere Unifikatoren geben kann. Dies ist in der Tat der Fall, denn jede Spezialisierung (Einsetzen von Werten für freie Variablen) eines Unifikators ist wieder ein Unifikator. Für Prolog gilt jedoch, daß, falls überhaupt einen Unifikator existiert, es genau einen allgemeinsten gibt. Dies bedeutet, daß alle anderen eine Spezialisierung dieses einen Unifikators sind. Dies ist eine wichtige Eigenschaft von Prolog, die in allgemeineren Systemen nicht immer erhalten bleibt. Z.B. sind die beiden Ausdrücke

```
verkette(X1, [X2, X3], [Y, b]).
verkette([], L, L).
```

unifizierbar, denn durch die Variablenbelegung

```
X1 := []
L  := [X2, X3]
X2 := Y
X3 := b
```

werden beide Ausdrücke gleich.

Dies ist auch der allgemeinste Unifikator. Aus ihm lassen sich unendlich viele Unifikatoren durch das Einsetzen beliebiger Terme für `Y` ableiten. Das Prolog-System findet immer den allgemeinsten Unifikator. Ein Unifikator u (d. h. eine

Variable-Wert-Paarliste) können wir auch als eine Substitutionsvorschrift auffassen. Das Ergebnis der Anwendung einer solchen Substitition u auf eine Aussage A bezeichnen wir formal mit Au. Als Komposition von zwei Substitutionen u und v bezeichnen wir die Verkettung der Variable-Wert-Paar-Listen und schreiben dafür $v \circ u$ oder auch einfach uv. Dabei müssen eventuell Variablen umbenannt werden. Denn ein Variablenname könnte sowohl in der ersten als auch in der zweiten Liste vorkommen, welches unbeabsichtigte Effekte haben könnte.

7.4.1 Das Unifikationsverfahren

Vergleich:	Aktion:
Variable → Variable	Identifiziere beide Variable.
Term → Variable	Die Variable erhält den Term als Wert.
Variable → Term	Die Variable erhält den Term als Wert.
Konstante → Konstante	Test auf Gleichheit.
Struktur → Struktur	Test auf Gleichheit der Funktoren, anschließend Unifikation der Argumente.

Beispiel: Die folgenden beiden Ausdrücke

```
T1 = liste(X, [Y, []])
T2 = liste(a, [Z, W])
```

sind durch den Unifikator $u_1 = X/a, Y/a, Z/a, W/$ unifizierbar („können gleich gemacht werden“). Der Unifikator ist hier als eine Liste von Substitutionen dargestellt, die der Reihe nach auf beide Ausdrücke 'T1' und 'T2' angewendet werden. Zuerst muß die erste Substitution X/a angewendet werden, danach sehen die beiden Terme folgendermaßen aus:

```
T1{X/a} = liste(a,[Y,[]])
T2{X/a} = liste(a,[Z,W])
```

Nun wird die zweite Substitution angewendet:

```
T1{X/a}{Y/a} = liste(a,[a,[]])
T2{X/a}{Y/a} = liste(a,[Z,W])
```

Nach dieser Substitution erhält man folgendes:

```
T1{X/a}{Y/a}{Z/a} = liste(a,[a,[]])
T2{X/a}{Y/a}{Z/a} = liste(a,[a,W])
```

Und zu guter letzt ergibt sich:

```
T1{X/a}{Y/a}{Z/a}{W/[]} = liste(a,[a,[]])
T1{X/a}{Y/a}{Z/a}{W/[]} = liste(a,[a,[]])
```

Auf diese Weise sind die beiden Terme also unifiziert worden, sozusagen miteinander „verschmolzen" worden. Ein anderer Unifikator wäre beispielsweise $u2 = X/a, Y/Z, W/$.

Dieser Unifikator ist allgemeiner als $u1$. denn bei $u2$ wurden möglichst viele Variablen beibehalten. Ein Unifikator u von zwei Ausdrücken A und B heißt *allgemeinster Unifikator*, wenn es zu jedem Unifikator v von zwei Ausdrücken A und B eine Substitution s gibt, so daß die Hintereinanderschaltung von u und s den Unifikator v ergibt, in Zeichen $v = us$ $(v = v \circ s)$.

Prolog kann die existenzquantifizierte Konjunktion elementarer Aussagen beweisen, da

$$\neg\exists X_1, \ldots, X_m : A_1 \ldots A_n \tag{7.17}$$

gleich

$$\forall X_1, \ldots, X_m : \neg A_1 \ldots \neg A_n \tag{7.18}$$

ist und damit der Hornklausel $\neg A_1, \ldots, \neg A_n$ (ohne positives Literal) entspricht. $X_1, \ldots, X_m$ seien dabei alle in $A_1 \ldots A_n$ vorkommenden freien Variablen.

7.5 Das Beweisverfahren von Prolog

Die Negation einer Anfrage

$$\exists X_1, \ldots, X_m : A_1 \ldots A_n \tag{7.19}$$

ergibt

$$\forall X_1, \ldots, X_m : \neg A_1 \ldots \neg An \tag{7.20}$$

Dieses entspricht der Hornklausel

$$\neg A_1, \ldots, \neg A_n. \tag{7.21}$$

Starte das Verfahren mit $L := [A_1, \ldots, A_n]$, $s :=$ (leere Substitution)

1. Wenn L=[], dann Rückgabe von s (erfolgreicher Beweis).
2. Anderenfalls wähle ein A_i aus L. $L := A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n$ $\phi :=$ Liste der Axiome des Prädikates von A_i.
3. Suche ein Axiom $C \leftarrow P_1, \ldots, P_j$ in AXIOME, so daß der Kopf C unifizierbar ist mit Ais; berechne den Unifikator u; setze ϕ auf die Restliste der Axiome.

4. Falls kein solches Axiom mehr existiert: Rückgabe von FAIL. (Abbruch des Beweises dieser Aussage.)
5. Anderenfalls wende das Verfahren rekursiv auf die Liste der Prämissen an:

   Rette L, ϕ und s; setze $L := P_1, \ldots, P_j$; $s := u$; Aufruf des Verfahrens; restauriere L, ϕ und s.

   Das Verfahren liefert entweder FAIL oder eine Substitution v_1.
6. Falls FAIL, gehe nach 3.
7. Anderenfalls wende das Verfahren rekursiv auf L an:

   Rette L, ϕ, s und v_1; $s := sv_1$; Aufruf des Verfahrens; restauriere L, ϕ, s und v_1.

   Das Verfahren liefert entweder FAIL oder eine weitere Substitution v_2.
8. Falls FAIL, gehe nach 3.
9. Anderenfalls ist der Beweis gelungen: Rückgabe der Ergebnissubstitution: $v := sv_1v_2$

Dieses Verfahren liefert zu einer existenzquantifizierten Konjunktion elementarer Aussagen entweder FAIL, falls kein Beweis gefunden wurde, oder eine Substitution 's', falls ein Beweis gefunden wurde. Dies bedeutet, daß nicht nur eine Entscheidung über die Beweisbarkeit getroffen wird, sondern, daß im Falle der positiven Entscheidung eine Variablenbelegung der existenzquantifizierten Variablen berechnet wird. Sind Y1, ... ,Yl alle die in $(A_1, \ldots, A_n)$ vorkommenden freien Variablen, so liefert der Beweis:

$$\forall Y_1, \ldots, Y_l : (A_1 \ldots A_l)s \tag{7.22}$$

Es handelt sich also trotz der indirekten Beweisführung um ein konstruktives Verfahren! Dies ist äußerst wichtig, denn sonst könnte ein Prolog-Programm keine Ergebnisse berechnen.

Dieses Verfahren ist an zwei Stellen nicht deterministisch.

- Es kann mehrere Axiome geben, die 2. erfüllen, und
- die Reihenfolge, in der mehrere Prämissen bewiesen werden, ist unbestimmt.

Prolog wählt gemäß der Eingabereihenfolge das erste Axiom, das 3. erfüllt, aus. Es arbeitet die Aussagenliste L (der Prämissen, also L:=[P1,...,Pj]), ebenfalls in der Eingabereihenfolge ab, und zwar so, daß zuerst versucht wird, einen vollständigen Beweis für die erste Prämisse zu suchen, anschließend für die zweite usw. Diese Strategie kann natürlich in Sackgassen führen. In so einem Fall muß die Beweisgeschichte (oder zumindestens ein Teil davon) wieder aufgehoben werden, damit an einem früheren Beweisstadium zurückgekehrt und von dort ein anderer Beweisversuch (sofern möglich) unternommen werden kann. Dieses im Falle einer Sackgasse nötige Vorgehen wird *Backtracking* genannt.

7.6 Backtracking

Ein weiterer grundlegender Mechanismus von Prolog-Interpretern ist das *Backtracking*. Das Backtracking beruht darauf, daß bei der Unifikation eines Goals (Zieles) mit einem Klauselkopf häufig mehrere Alternativen bestehen, die ebenfalls mit dem Goal unifiziert werden können. Solche Zustände, zusammen mit den bis dahin durchgeführten Variablenbindungen, nennt man *Choicepoints*. Schlägt einmal ein Teilziel fehl, so kehrt der Interpreter zum zuletzt besuchten Choicepoint zurück und versucht, mit einer alternativen Klausel das Ziel zu beweisen. Existiert an diesem Choicepoint keine alternative Klausel mehr, so wird zum nächst höher gelegenen Choicepoint zurückgegangen usw. Eine Rückkehr ist verbunden mit der Freigabe der Variablen, die seit dem Choicepoint instantiiert wurden.

Der Backtrackingmechanismus spielt in zweierlei Hinsicht eine wichtige Rolle. Zum einen kann beim Beweisen der Anfrage eine „falsche“ Unifikation vorgekommen sein, d. h. eine Unifikation, die in eine Sackgasse führt, so daß durch Backtracking eine Alternative versucht werden muß. Zum anderen ist Backtracking wichtig, wenn man sich mehr als eine Lösung ausgeben lassen möchte, denn dann muß das System die alternativen Unifikationen an den Choicepoints versuchen.

Zur Verdeutlichung des Backtrackings betrachten wir folgendes Beispiel:

```
mutter( maria, bernd).
mutter( anke, maria).
mutter( maria, jens).
mutter( bern, jens).

grossmutter( G, E) :-
   mutter( G, X),
   mutter( X, E).

grossmutter( G, E) :-
   mutter( G, X),
   vater( X, E).
```

Wir stellen diesselbe Anfrage wie in auf Seite 60, diesmal allerdings möchten wir alle Lösungen erhalten:

```
grossmutter( M, jens).
```

In diesem Fall unifizieren die Köpfe beider `grossmutter`-Regeln mit der Anfrage. Das Prolog-System nimmt jedoch zuerst immer das erste Vorkommen. Daraufhin müssen nun die Subgoals `mutter(G, X)` und `mutter( X, E)` bewiesen werden. Das erste Subgoal `mutter( G, X)` unifiziert mit `mutter( maria, bernd)`', dadurch erhält `X` den Wert `bernd`. Nun muß also noch das Subgoal `mutter( bernd, jens)` bewiesen werden, was allerdings nicht gelingt. Deshalb wird zum Choicepunkt zurückgegangen und das erste Subgoal mit `mutter( anke, maria)` unifiziert. Nun muß das zweite Subgoal mit `mutter( maria, jens)` bewiesen werden, was ja als Faktum in der Wissensbasis steht. Also wird `grossmutter( anke, jens` als Lösung ausgegeben. Da wir alle Lösungen haben wollen, wird erneut zum letzten Choicepunkt gegangen und es werden noch die restlichen zwei alternativen Unifizierungen für das erste Subgoal durchgeführt, `mutter( maria, jens)` und `vater( bernd, jens)`, die allerdings keine weiteren Lösungen bringen. Diese unnützen Alternativen für das Unifizieren des ersten Subgoals können wir uns in diesem Fall sparen, wenn wir die `grossmutter`-Regel folgendermaßen ändern:

```
grossmutter( G, E) :-
  mutter( X, E),      % Die beiden Bedingungen wurden
  mutter( G, X).      % in der Reihenfolge getauscht
```

Dadurch erreichen wir nämlich, daß nicht alle `mutter`–Fakten, die sich in der Wissensbasis befinden, mit dem ersten Subgoal, diesmal wäre es `mutter( X, jens)`, unifizieren. Denn dadurch, daß die zweite Variable mit einem Wert belegt wurde, kommen nur noch die `mutter`-Klauseln in Frage, die `jens` als zweites Argument besitzen. Wir ändern die zweite `grossmutter`–Regel analog, so daß wir im weiteren von folgendem Prolog-Programm ausgehen werden:

```
mutter( maria, bernd).
mutter( anke, maria).
mutter( maria, jens).
vater( bern, jens).

grossmutter( G, E) :-
  mutter( X, E),
  mutter( G, X).

grossmutter( G, E) :-
  vater( X, E),
  mutter( G, X).
```

Das erste Teilziel der ersten `grossmutter`–Klausel `mutter( X, jens)`[1], unifiziert mit `(mutter anke maria)`. Dadurch erhält `X` den Wert `maria`. Da `mutter( anke, maria)` in der Wissensbasis steht, ist mit `grossmutter( anke,`

[1](`E` ist durch die Anfrage `grossmutter( M, jens).` mit `jens` instantiiert.

`jens)` eine Lösung gefunden worden. Da nun mit der ersten Klausel des Prädikates `grossmutter` keine weiteren Alternativen mehr bestehen, wird zum nächsthöheren Choicepoint zurückgegegangen, und die entsprechenden Variablenbelegungen werden rückgängig gemacht. Hier unifiziert nun das erste Subgoal mit `vater( bernd, jens)` und die Variable `X` wird mit `bernd` unifiziert. Da aber keine Mutter von `bernd` in unserer Wissensbasis eingetragen ist, bricht hier der Gesamtbeweis ab, da es keine weiteren Choicepoints mehr gibt, d. h. es sind alle Möglichkeiten versucht worden.

Um uns nun zu verdeutlichen, daß die Reihenfolge der Subgoals eine Rolle spielt, rufen wir obiges Beispiel einmal auf:

```
grossmutter( M, jens).
M = anke;
M = maria;
no.
```

Wir merken uns die 55 Millisekunden, tauschen die Reihenfolge der Prämissen in den 'Großmutter'-Klauseln und stellen erneut dieselbe Anfrage:

```
grossmutter( M, jens).
M = anke;
M = maria;
no.
```

An dem unterschiedlichen Zeitverbrauch sieht man sehr deutlich, daß die Reihenfolge der Prämissen eine Rolle spielt. Würde man jedoch zu einer Person die Enkel wissen wollen, so würde man die Anfrage `(großmutter maria *)??` stellen und das Zeitverhalten würde vielleicht ganz anders aussehen.

Das Verhalten eines Prolog-Systems sollte unabhängig sein von der Reihenfolge der Eingabe der Axiome und innerhalb der Axiome unabhängig von der Reihenfolge der Prämissen. Daß dies nicht der Fall ist, haben wir schon an obigem Beispiel gesehen. Der Grund dafür liegt in der oben beschriebenen sequentiellen Arbeitsweise des Prolog-Interpreters. Denn dieser verfährt nach der Strategie 'depth first, left to right' (Tiefensuche links nach rechts). Folgendes Beispiel demonstriert noch einmal wie wichtig die Eingabereihenfolge sein kann.

7.6.1 Bedeutung der Eingabereihenfolge

Wir betrachten folgendes Prolog-Programm, welches die Subtraktion zweier Zahlen definiert:

```
subtrahiere(Z,0,Z).
subtrahiere(X,Y,Z):-
  sum(X1,1,X),                   % X1+1=X <-> X-1=X1
  sum(y1,1,Y),
```

```
    subtrahiere(X1,Y1,Z).
```

In diesem Programm wird die Substraktion zweier nichtnegativer Zahlen auf das schrittweise Dekrementieren beider Zahlen zurückgeführt. Wenn wir von dem ersten Argument `X` das zweite Argument 0 abziehen, also nichts abziehen, so ist `X` das Ergebnis. In dem Fall, wo das zweite Argument größer als 0 ist, dekrementieren (um 1 erniedrigen) wir beide Argumente. Auf die neuen, so entstandenen Werte, wenden wir dann wieder `subtrahiere` an. Dies wird solange wiederholt, bis das zweite Argument 0 ist. Dann greift das erste Axiom und liefert das Ergebnis, nämlich das erste Argument.

So funktioniert unser Beispiel. Tauschen wir jedoch die Reihenfolge der beiden Axiome,

```
subtrahiere (X,Y,Z) :-
    sum(X1,1,X),                % X1+1=X <-> X-1=X1
    sum(Y1,1,Y),
    subtrahiere(X1,Y1,Z).
subtrahiere (X,0,X).
```

so führt der Prolog-Interpreter eine Endlosschleife aus. Denn es wird immer erst versucht, das (in der Eingabereihenfolge) erste Axiom anzuwenden, was in diesem Fall auch immer gelingt. Das bedeutet, daß rekursive Definitionen kritisch sein können bezüglich der Eingabereihenfolge, es aber nicht sein müssen. In diesem Zusammenhang sollte man sich folgendes merken, was auch für die Fälle, in denen die Programme unabhängig von der Reihenfolge der Axiome funktionieren, sinnvoll ist:

> Die rekursiven Axiome sollte man zum Schluß und Abbruchbedingen vorher angeben. Denn unter gewissen Umständen kann der Prolog-Interpreter dann die Rekursion sogar in eine Iteration umwandeln, welches die Laufzeit günstig beeinflußt.

Kurz zusammengefaßt gilt folgende Faustregel:

> Wenn möglich sollte der rekursive Aufruf nur im letzten Axiom und dort in der letzten Prämisse auftreten.

7.7 Übungen

1. Formuliere mit Hilfe von Prolog Fakten und Regeln, die folgende Beziehungen beschreiben:
    - „Fritz ist größer als Egon“,
    - „Egon ist größer als Peter“ etc.

Folglich ist Fritz auch größer als Peter.

Beim Aufruf des Programms sollen alle aus den Fakten ersichtlichen Größer-Beziehungen ausgegeben werden.

Kapitel 8

Kontrollstrukturen und Backtracking

Im folgenden wollen wir uns mit dem Kontrollfluß in einem Prolog-Programm beschäftigen. Um sich den Kontrollfluß zu verdeutlichen, gibt es einige verschiedene Darstellungsweisen.

8.1 Und-/Oder-Bäume

Wir betrachten zunächst Beweisbäume (genauer sogenannte Und-/Oder-Bäume). Auf die folgende Weise wollen wir einem Prolog-Programm einen Beweisbaum zuordnen:

1. Die zu beweisende Aussage bildet die Wurzel des Baumes.
2. Für jedes Axiom der Definition des Prädikats dieser Aussage wird ein Nachfolgeknoten eingetragen. Diese Knoten sind durch „oder“ verbunden.
3. Für jede Prämisse eines Axioms wird ein Nachfolgeknoten an diesem Axiom eingetragen. Diese Knoten sind durch „und“ verbunden.
4. Dieser Prozeß wird für die Prämissen (die ja ebenfalls zu beweisende Aussagen sind) wiederholt.

Die Blätter so eines Beweisbaumes entsprechen den Axiomen, die keine Prämissen besitzen, also immer wahr sind. Wir betrachten folgendes Prolog-Programm:

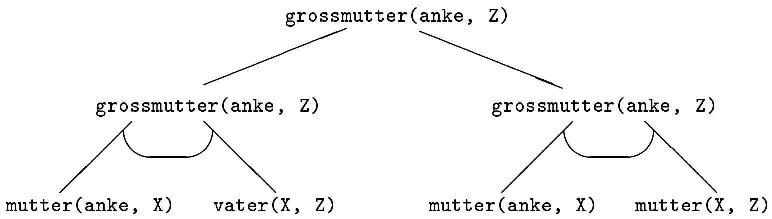
```
mutter(anke, maria).
mutter(maria, jens).
```

```
grossmutter(G, E) :-
  mutter(G, X),
  vater(X, E).

grossmutter(G, E) :-
  mutter(G, X),
  mutter(X, E).
```

Wir möchten nun ein Enkelkind von Anke finden. Dazu stellen wir folgende Anfrage:

```
?- grossmutter(anke, Z).
```

Für obiges Prolog-Programm ergibt sich nach unserer Definition folgender Und- / Oder-Baum:



Das Prolog-System versucht nun an einem Oder-Knoten einen Nachfolger (Alternative) zu beweisen und an einem Und-Knoten alle Nachfolger zu beweisen. Ist an einer Stelle im Baum die Fortsetzung des Beweises nicht mehr möglich, so wird bis zu einem Oder-Knoten zurückgegangen (Backtracking), bei dem noch eine Alternative möglich ist. Ein Oder-Knoten wird auch Choicepoint genannt (s.o.). Das Prolog-System arbeitet sich dabei von oben nach unten und von rechts nach links durch den Baum (depth first, left to right).

In unserem Beispiel gibt es nur einen Oder- und zwei Und-Knoten. Um unsere Anfrage `grossmutter(anke, Z)` zu beweisen, versucht nun das Prolog-System gemäß der Strategie, zuerst den linken Nachfolger im Oder-Knoten zu beweisen. Dazu müssen zwei Prämissen bewiesen werden, `mutter(...)` und `vater(...)`. Die erste Prämisse kann leicht bewiesen werden, da sie ja so in der Wissensbasis steht. Bei der zweiten jedoch schlägt der Beweisversuch fehl, da sich keinerlei Axiom der Form `vater(...)` in der Wissensbasis befindet. Deshalb findet nun Backtracking statt, d. h. es wird im Baum bis zum Oder-Knoten zurückgegangen und versucht, den anderen Nachfolger der Anfrage zu beweisen. Dazu sind ebenfalls zwei Prämissen zu beweisen. Die erste läßt sich beweisen mit `mutter(anke, maria)`. Jetzt muß noch die zweite Prämisse bewiesen werden, wobei nun `Z` den Wert `maria` hat. (Wir erinnern uns, daß sämtliche Vorkommen einer Variablen innerhalb eines Axioms den gleichen Wert zugewiesen bekommen, sofern die Variable durch Unifikation an irgendeiner Stelle innerhalb des Axioms einen Wert erhält.) Das Axiom `mutter(anke, maria)` steht aber in der Wissensbasis und so ist der Beweis erfolgreich.

8.2 CUT und FAIL

Um nun die Auswertung eines Prädikates explizit steuern zu können, gibt es die Prädikate 'cut' (in Standard-Prolog `!` geschrieben) und `fail`. Die Auswertung von `!` gelingt immer, jedoch werden im Beweisbaum die Alternativen am übergeordneten 'oder'-Knoten abgeschnitten. Dadurch wird erreicht, daß die noch nicht getesteten Alternativen an diesem 'oder'-Knoten bei einem späteren Backtracking nicht mehr gefunden werden. In diesem Sinne entspricht der Cut einer Tür, die nur auf der einen Seite eine Klinke besitzt. Hat man diese Tür einmal durchschritten, so gibt es kein zurück mehr. Analog ergeht es dem Prolog-System, wenn es auf einen Cut stößt. Ansonsten wird der Beweis jedoch normal weitergeführt.

Das Prädikat `fail` mißlingt immer. Es erzwingt also immer ein Backtracking. Die Kombination von `!` und `fail` erweist sich in der Praxis als recht nützlich. Denn wird ein `fail`, das unmittelbar hinter einem 'cut' steht, erreicht, so wird dadurch sofort der Beweis des Goals (Ziels) abgebrochen. Deshalb kann mit der Kombination aus 'cut' und `fail` eine Art Negation realisiert werden (genannt „negation on failure"). Diese Negation entspricht jedoch nicht der logischen Negation (der Prädikatenlogik). Die logische Negation der Prädikatenlogik ist in Prolog auch nicht möglich, da die durch sie entstehenden Axiome keine Hornklauseln mehr sind. In diesem Sinne ist beim Gebrauch der folgenden Definition von 'not' Vorsicht geboten.

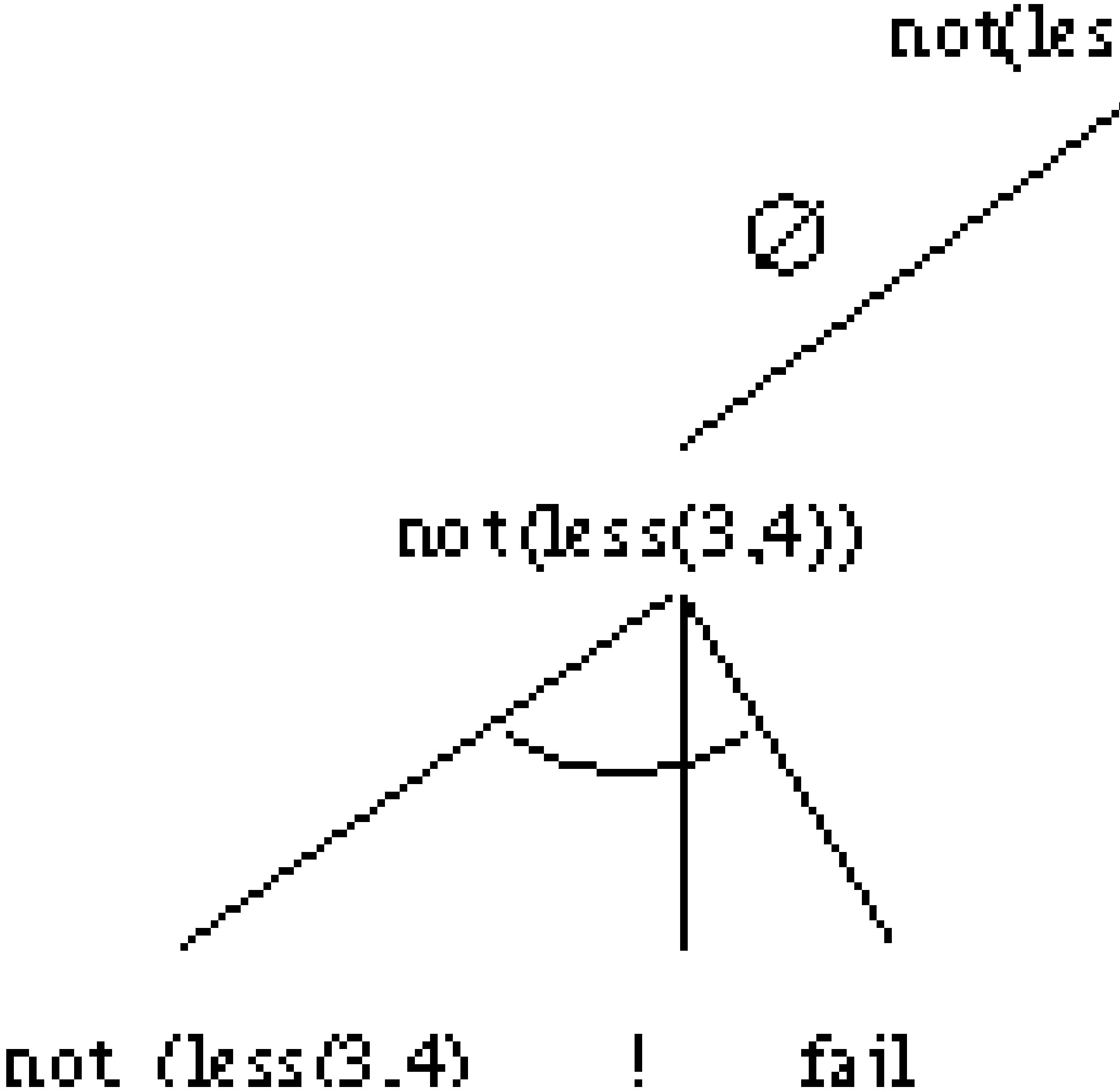
```
not(Aussage) :-
  Aussage, !,
  fail.

not(_).
```

Zur Auswertungszeit von `Aussage` muß die Aussage einen Wahrheitswert besitzen, d. h. es dürfen keine uninstantiierten Variablen vorkommen, da in diesem Falle das System nicht entscheiden kann, ob die Aussage wahr oder falsch ist. Deshalb wird in diesem Fall für `not` immer die Unerfüllbarkeit angenommen. Zurück zu unserem Beispiel. Kann `Aussage` bewiesen werden, so sorgt `fail` dafür, daß diese Klausel von `not` mißlingt. Da 'fail' hinter dem `!` steht, wird der gesamte Rest des Beweisbaumes abgeschnitten und der Beweis somit abgebrochen. D.h. es wird auch nicht mehr die zweite Klausel (das zweite Axiom) von `not` versucht. Mißlingt jedoch der Beweis von `Aussage`, so wird die erste Klausel vor Erreichen des `!` verlassen. Deshalb steht für den Beweis von `not` noch die zweite Klausel zur Verfügung. Diese hat jedoch keine Prämissen und ist somit immer wahr. Zusammengefaßt bedeutet dies, die Negation mit `not` ist nur unter folgenden sehr starken Restriktionen äquivalent zur logischen Negation:

- `Aussage` darf zur Auswertungszeit keine freie Variable enthalten.

- Die Wissensbasis muß vollständig sein, d.h., was nicht beweisbar ist, ist falsch („closed world asumption").

Die Funktionsweise des `!` machen wir uns nun an einem Beispiel mit dem oben defininierten Prädikat `not` deutlich. Und zwar betrachten wir die Anfrage `not( less( 3, 4)).` und den dazugehörigen Und-oder-Baum:

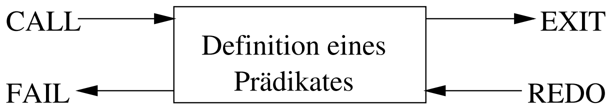


Der Prolog-Interpreter versucht gemäß der Strategie (depth first, left to right) den linken Teil des Oder-Knotens zu beweisen. Dazu muß er die erste Klausel von `not` beweisen. Deshalb versucht er, deren Prämissen der Reihenfolge nach zu beweisen. `less( 3, 4)` gelingt, ebenso `!` . Dieses Prädikat erzeugt jedoch den

Nebeneffekt, daß der darüber befindliche Oder-Knoten abgeschnitten wird. Hier angedeutet durch 'Ø', d.h. beim Backtracking kann nicht mehr über 'Ø' hinaus zurückgesetzt werden. Dadurch bleiben aber keine Alternativen mehr übrig. Deshalb ist der gesamte Beweis gescheitert. Betrachten wir nun die Anfrage `not( less( 4, 3))`, so sieht der Baum analog zu oben aus, jedoch mißlingt der erste Beweisversuch schon beim Beweis der Aussage `less( 4, 3)`. In diesem Fall wird also nicht mehr der `!` überschritten, was zur Folge hat, daß beim Backtracking, die zweite Alternative des Oder-Knotens erreicht wird. Diese gelingt jedoch, wodurch die gesamte Aussage wahr wird.

8.3 Das Vierportmodell

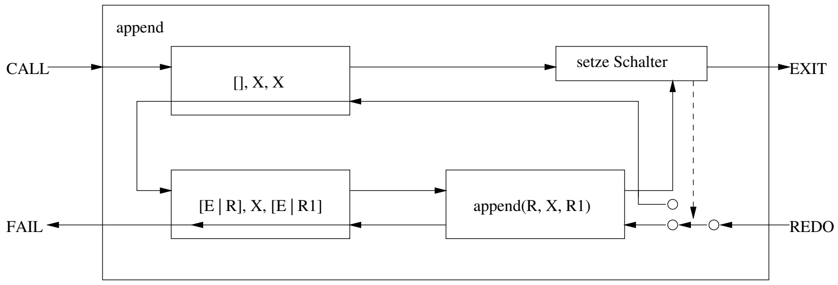
Eine weitere Möglichkeit den Kontrollfluß in einem Prolog-Programm darzustellen, bietet das Vierportmodell (Clocksin, Mellish 1981), mit welchem sich besonders die Wirkung des Cuts demonstrieren läßt. Dieses basiert im Unterschied zu Beweisbäumen auf einem prozeduralen Modell. Bei diesem Modell ordnet man jeder Definition eine Box mit zwei Eingängen (CALL und REDO) und zwei Ausgängen (FAIL und EXIT) zu:



In die Box einer Definition werden nun eine Reihe weiterer Boxen eingetragen. Es wird zuerst eine Box für die Unifikation mit der Konklusion (mit dem Klauselkopf) eingetragen, und danach je eine Box für jede Prämisse. Normalerweise ist nun der EXIT-Ausgang einer Box mit dem CALL-Ausgang der nächsten Box verbunden. Besteht eine Definition aus mehreren Axiomen, so muß jedoch sichergestellt sein, daß bei einem möglichen Backtracking bei dem Axiom weiter gemacht wird, von dessen EXIT Ausgang die Definition verlassen worden ist. D.h. es muß beim Betreten durch den REDO-Eingang der Definition auch der REDO-Eingang dieses Axioms gewählt werden. Diese Entscheidung wird durch einen Umschalter symboliert, der beim Verlassen der Defintion auf das entsprechende Axiom gesetzt wird. Es ergibt sich dann für das Prädikat 'append'

```
append([], X, X).

append([E|R], X, [E|R1]) :-
   append(R, X, R1).
```

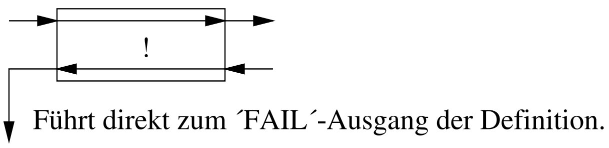
folgende Box:



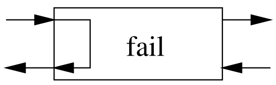
Die Box mit dem Inhalt 'setze Schalter' symbolisiert die Steuerung des oben beschriebenen Umschaltens des Schalters (unten rechts in der Box dargestellt. Die Box oben links steht für das Axiom 'append([], X, X).' und die beiden Boxen unten stehen für das Axiom 'append([E | R], X, [E | R1]) :- append(E, R, R1).'. In Prolog ist die Unifikation eindeutig, d.h. wenn sie gelingt, existiert genau eine allgemeinste Variablenbelegung. Gelangt nun der Kontrollstrom durch den REDO-Eingang zu einer Unifikationsbox (die beiden links dargestellten Boxen), so wird er direkt zum FAIL-Ausgang weitergeleitet, denn es kann keine andere allgemeinste Variablenbelegung gefunden werden. Nach obiger Vorstellung kann man mit einer Kette solcher Boxen eine Liste von Aussage darstellen, wobei man den EXIT-Ausgang der einen mit dem CALL-Eingang der nächsten und den FAIL-Ausgang mit den REDO-Eingang der vorhergehenden Box verbindet. Es ergibt sich also für eine Liste, bestehend aus drei Aussagen, folgendes Bild in unserer Modelldarstellung:

Nehmen wir einmal an, wir möchten die obigen drei Aussagen beweisen und der Beweisvorgang sei bei der 'Aussage 2' angekommen. Diese Box wird nun durch den CALL-Eingang betreten. Dabei muß man sich jetzt die Box durch eine Kopie der zugehörigen Definitionsbox ersetzt denken. Es gibt nun zwei Möglichkeiten, entweder der Beweis gelingt, dann wird die Box durch den EXIT-Ausgang verlassen und die dritte Box durch den CALL-Eingang betreten, oder der Beweis mißlingt. Im letzteren Fall wird die zweite Box durch den FAIL-Ausgang verlassen und die erste Box durch den REDO-Eingang betreten. Dieser Vorgang wird mit BACKTRACKING bezeichnet, d.h. der Beweisvorgang wird zurückverfolgt und die Variablenbelegungen, die auf diesem Beweisstück (auf dem Hinweg) durchgeführt worden sind, werden rückgängig gemacht. Nun wird ein alternativer Beweis für 'Aussage 2' versucht. Gelingt dieser, so wird die Box wieder durch den EXIT-Ausgang verlassen und die nächste Box durch den CALL-Eingang betreten. Der gesamte Beweisvorgang läuft solange, bis der EXIT-Ausgang der letzten Aussage, in diesem Fall der 'Aussage 3', erreicht worden ist. Dann ist für den gesamten Beweisversuch eine Lösung gefunden worden. Oder er läuft solange bis alle Alternativen versucht worden sind, und die erste Box ('Aussage 1') durch den FAIL-Ausgang verlassen wird. In diesem Fall ist der gesamte Beweisversuch fehlgeschlagen.

Die Prädikate '!' und 'fail' entsprechen den folgenden Boxen:



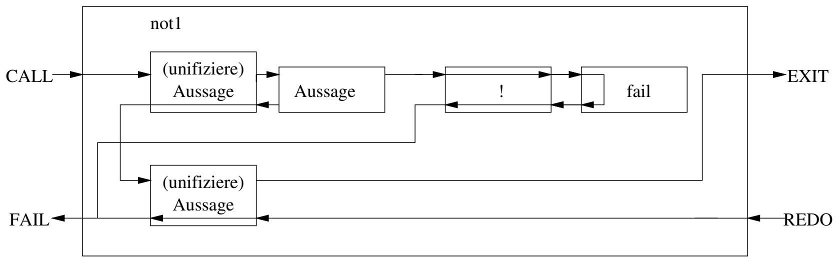
Der FAIL-Ausgang der Cuts ist also nicht mit dem REDO-Ausgang der vorhergehenden Box verbunden, sondern direkt mit dem FAIL-Ausgang der Definition des Prädikates (der „größten" Box).



Wird die 'fail'-Box durch den CALL-Eingang betreten, so wird sie sofort durch den FAIL-Ausgang verlassen. 'fail' erreicht also niemals seinen EXIT-Ausgang. Kommen wir jetzt noch einmal auf unser Beispiel mit 'not' zurück. Die Definition hatte ja folgende Gestalt:

```
not(Aussage) :-
    Aussage, !, fail .
not(_).
```

Mit dem Vierportmodell können wir 'not' wie folgt darstellen:



Der obere Teil der Box stellt das erste Axiom dar. Ist 'Aussage' beweisbar, so wird der Cut('!') überschritten und 'fail' erreicht. Dieses löst ein Backtracking aus, es wird also die 'fail'-Box durch den FAIL-Ausgang verlassen und die 'cut'-Box durch den REDO-Eingang betreten. Dieser Eingang führt aber direkt zum FAIL-Ausgang der 'cut'-Box und von dort direkt zum FAIL-Ausgang der großen Box, d.h. der Box, die die Definition des Prädikates 'not' darstellt. Dadurch wird jeglicher alternativer Beweisversuch unterdrückt.

Mißlingt jedoch der Beweisversuch von 'Aussage', so wird der '!' nicht erreicht, denn die 'Aussage'-Box wird durch den FAIL-Ausgang verlassen, welcher mit dem CALL-Eingang der ersten (und einzigen) Box des zweiten Axioms der Definition von 'not' verbunden ist. Die Unifikation gelingt und so wird die gesamte Defintion durch den EXIT-Ausgang verlassen.

Wie man schon an diesen Beispielen sieht, sind Definitionen, die einen '!' enthalten sehr empfindlich gegenüber der Vertauschung von Axiomen und auch gegenüber der Vertauschung von Prämissen, in denen ein '!' vorkommt. Daher ist es sehr ratsam, sofern man sich einen guten Programmierstil angewöhnen möchte, auf die Verwendung von '!', so gut es geht, zu verzichten.

Nicht immer läßt sich die Verwendung eines Cuts vermeiden, jedoch ist es in vielen Fällen möglich, 'not' anstelle eines Cuts zu verwenden. Denn häufig wird der Cut in folgendem Definitionsschema verwendet:

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

Diese beiden Axiome „besagen“: Gilt 'die Aussage 'a', so soll Aussage 'c' dann gelten, wenn auch 'b' gilt. Anderenfalls soll Aussage 'c' dann gelten, wenn 'd' gilt. Vom logischen Standpunkt aus gesehen, ist dies aber nicht die Interpretation der obigen Axiome. Denn bei der deklarativen Interpretation kann ein '/' einfach ignoriert werden, denn ein '/' ist immer wahr. Dieses könnte man aber auch folgendermaßen ausdrücken:

```
c :- a, b.
c :- not a, d.
```

Abgesehen von der suggestiveren Form hat diese Darstellung noch den Vorteil, das sie unabhängig ist von der Reihenfolge der Axiome. Die deklarative und die prozedurale Interpretation stimmen jetzt überein. Jedoch ergibt sich auch ein Nachteil, denn 'a' wird eventuell zweimal ausgewertet, welches unerwünschte Nebeneffekte haben kann, z.B. wenn in 'a' Ausgaben auf den Bildschirm stattfinden etc.

Ein weiteres Problem, welches man beachten muß, stellen eventuell vorkommende freie Variablen in 'a' dar. Denn 'not' sollte nicht auf Ausdrücke angewendet werden, in denen freie Variablen vorkommen. Eine, wenn auch nicht immer befriedigende Lösung, stellt die Verwendung des Prädikates von 'numbervars' dar. Dieses erhält (in SWI-Prolog) vier Argumente

```
% numbervars(+Term, +Functor, +Start, -End)

?- numbervars(foo(A, B, A), this_is_a_variable, 0, End).

A = this_is_a_variable(0)
B = this_is_a_variable(1)
End = 2
```

Es unifiziert alle freien Variablen, die in 'Term' vorkommen, mit dem Term 'Funktor(Zahl)', wobei Zahl mit 'Begin' initialisiert wird und nach jeder unifizierten Variable inkrementiert wird. Der Wert von 'Ende' ist die letzte verwendete Zahl plus 1. Auf diese Weise können alle noch freien Variablen belegt werden.

Es ist oftmals wichtig zu wissen, ob eine Variable frei oder aber gebunden ist. Um dieses festzustellen, gibt es ein 'built-in'-Prädikat 'var'. Dieses erhält als Argument eine Variable und testet, ob diese frei ist.

8.4 Übungen:

1. Betrachte folgende Datenbasis:

```
holiday( tuesday, july4).
holiday( tuesday, fair).
holiday( saturday, fair).

weather( sunday, fair).

weekend( sunday).
weekend( saturday).

make( potatoe_salad, Day) :-
  picnic( Day).

picnic( Day) :-
  weather( Day, fair),
  weekend( Day).
picnic( Day) :-
  holiday( Day, july4).
```

   Beim Aufruf von

```
make(potatoe_salad, When).
```

   erhält man die Antwort

```
When = saturday;
When = sunday;
When = tuesday;
no.
```

   Modifiziere das Verhalten von Prolog durch Setzen des Cuts `!` so, daß folgendes gilt:

2. Auf die Anfrage

   ```
   make( potatoe_salad, When).
   ```

   erscheint

   ```
   no.
   ```

3. Es erscheint

   ```
   When = saturday;
   no.
   ```

4. Es erscheint

   ```
   When = saturday;
   When = sunday;
   no.
   ```

   Bei allen drei Teilaufgaben ist zu begründen, warum das Prolog-System so reagiert.

5. Bei der Lösung der folgenden Aufgaben sollen die Prädikate `append` und `not` *nicht* benutzt werden. Es ist davon auszugehen, daß die Eingabeargumente Mengen sind, d.h. es ist nicht nötig, sie vorher zu Mengen zu machen.

6. Definiere ein Prädikat, welches die Schnittmenge zweier Mengen bildet. Die Schnittmenge S von zwei Mengen besteht aus den Elementen, die sowohl in der einen als auch in der anderen Menge enthalten sind.

7. Definiere ein Prädikat, welches die Vereinigungsmenge zweier Mengen bildet. Eine Menge M ist die Vereinigung zweier Mengen, wenn sie alle Elemente der einen sowie alle Elemente der anderen Menge enthält, und sonst kein Element.

Kapitel 9

Programmierstrategien

9.1 Allgemeines zur Programmstruktur

Es gibt im wesentlichen zwei Typen von Prädikatsdefinitionen:

- Viele Klauseln ohne Bedingungen und mit wenigen oder keinen Variablen. Solche Prädikate fungieren meist als interne „Datenbank“.
- Wenige Klauseln mit Bedingungen und Variablen und fast keinen konstanten Argumenten,

Im Beispiel von Kapitel 2 ist 'vater' und 'mutter' vom ersten Typ, 'großvater' und 'großmutter' vom zweiten Typ. Prädikatsdefinitionen, die keinem der Typen zuzuordnen sind, sollten kritisch betrachtet werden.

Typische Definitionsstruktur für Prädikate vom zweiten Typ:

```
q(...).
    .              Behandlung einiger weniger Spezialfälle
    .              (z.B. Rekursionsanfang, "Abbruchbedingung")
    .

q(...) :-
    .
    .              wenige Klauseln
    .
    q(...).        Rekursion meist in der letzten Klausel
...
```

Die Summe der Prämissen einer Prädikatsdefinition sollte nicht zu groß werden. Werte viel größer als 5 sind kritisch. In solchen Fällen sollte überlegt werden, ob nicht konzeptuell zusammengehörende Prämissen zur Definition eines

Hilfsprädikats zusammenfaßbar sind. Prolog-Programme erreichen sehr schnell eine Komplexität, die vom Programmierer nur noch schwer durchschaubar ist. Dabei sind Prolog-Programme eigentlich auf mehreren Ebenen sehr gut modularisierbar:

- Jede Klausel hat eine eigenständige Bedeutung (falls versteckte Negationen vermieden werden , siehe 9.5).
- Jede Prädikatsdefinition ist ein kleines „Modul“ für sich. Sie kann eigenständig getestet und inkrementell erweitert werden.

9.2 Transitiver Abschluß von Relationen

Eine typische Anwendung der oben beschriebenen Programmstruktur tritt beim transitiven Abschluß einer Relation auf. Der naive Ansatz zur Definition einer transitiven Relation durch ein einziges Prädikat führt zu einer Endlosschleife:

```
q(a, b).
q(b, c).
...
q(X, Y) :-
  q(X, Z),
  q(Z, Y).
```

Die Basis der Relation (das Hassediagramm) und der transitive Abschluß müssen durch zwei getrennte Prädikate repräsentiert werden.

```
q1(a, b).
q1(b, c).
...

q(X, Y) :-
  q1(X, Y) .

q(X, Y) :-
  q1(X, Z),
  q(Z, Y).
```

9.3 Vermeidung von ';' und '='

Ein ';' ('oder') im Prämissenteil läßt sich grundsätzlich eliminieren. Der Preis dafür ist die Einführung einer neuen Klausel oder eines neuen Prädikates.

Beispiel:

```
grossvater(X, Y) :-                 % X ist Großvater von Y, wenn
  vater(X, Z),                      % X der Vater von Z ist und
  (vater(Z, Y) ; mutter(Z, Y)).     % Z Vater oder Mutter von Y ist
```

1. Version ohne ';':

```
grossvater(X, Y) :-                 % X ist Großvater von Y, wenn
  vater(X, Z),                      % X Vater von Z
  vater(Z, Y).                      % und Z Vater von Y ist

grossvater(X,Y)  :-                 % X ist Großvater von Y, wenn
  vater(X, Z),                      % X Vater von Z
  mutter(Z, Y).                     % und Z Mutter von Y ist
```

Die erste Prämisse beider Klauseln ist gleich. Dies bedeutet, daß ein beim Abarbeiten der ersten Klausel gelöstes Teilproblem beim Backtracking nochmal gelöst werden muß.

2. Version ohne ';':

```
grossvater(X, Y) :-
  vater(X, Z),
  elternteil(Z, Y).

elternteil(X, Y) :-
  vater(X, Y).

elternteil(X, Y) :-
  mutter(X, Y).
```

Diese Version ist von der Abarbeitung her äquivalent zu der Version mit `;`. Insbesondere verschachtelte `;`'s führen zu unübersichtlichen Programmen.

Oft ist die Verwendung von `=` überflüssig.

```
foo(X, Y) :-   % foo(Arg1, Arg2) ist wahr, wenn Arg1 = Arg2
    X = Y.
```

Das obige Beispiel kann man ohne `=` eleganter ausdrücken:

```
foo(X, X).     % foo(Arg1, Arg2) ist wahr, wenn Arg1 = Arg2
```

9.4 Generieren und Testen

Viele Programmierprobleme lassen sich als Suchprobleme in einem geeignet definierten Alternativenraum auffassen. Solche Probleme sind in einen Generierungs- und einen Testteil zerlegbar:

Der Generierungsteil zählt alle möglichen potentiellen Lösungen auf (spannt den Alternativenraum auf).

Der Testteil testet, welche der potentiellen Lösungen wirkliche Lösungen sind.

9.4.1 Das Dominoproblem

Die typische Lösung funktioniert nach der oben genannten Methode:

Ein Prädikat generiert aus der Ausgangskonfiguration durch Vertauschen und Drehen der Steine alle möglichen Konfigurationen.

Ein zweites Prädikat testet zu einer gegebenen Konfiguration, ob die Übergangsbedingungen erfüllt sind.

Die Lösungen erhalten wir durch die Konjunktion beider Prädikate:

```
?- gen(Konfiguration), test(Konfiguration).
```

Sind 'gen' und 'test' logisch sauber programmiert, so muß für die vertauschten Anfragen

```
?- test(Konfiguration), gen(Konfiguration).
```

das System die gleichen Lösungen berechnen. Oftmals ist die zweite Lösung aber um Größenordnungen effizienter, da der Generatorteil nicht blind, sondern nur noch solche Konfigurationen generiert, die die Bedingungen des Tests erfüllen und damit tatsächliche Lösungen sind.

9.5 Verwendung von '!' und versteckte Negation

Es gibt zwei Arten der Verwendung des Cutoperators '!'

- *sichere* (grüne) Cuts: Die Bedeutung des Programms ändert sich nicht, wenn '!' entfernt wird. '!' dient hier nur der Effizienzsteigerung.
- *unsichere* (rote) Cuts: Die Bedeutung des Programms ändert sich, wenn '!' entfernt wird. Hinter diesen Cuts verbirgt sich eine versteckte Negation.

Eine zu 'not' äquivalente Negation läßt sich implizit realisieren.

```
q(...) :-
    .                    Test auf die Situation,
    .                    in der q mißlingen soll
    .
```

```
    !, fail .

 q( ...) :-              restliche Definition
    .
    .
```

Eine andere typische Programmstruktur mit versteckter Negation:

```
q( ...) :-
  test(...),
  !,
  ... .

q(...) :-     Hier wird angenommen,
  ...         daß der Test mißlungen ist
```

Wenn die Definitionen zu verschachtelt sind, ist diese implizite Negation schlecht zu erkennen. Die eigenständige Bedeutung der Klauseln geht verloren. Diese Programmstruktur sollte also nur vorsichtig verwendet werden.

Formulierung mit expliziter Negation und ohne Cut:

```
q(...) :-
  test(...),
  ... .

q(...) :-
  not(test(...)),
  ... .
```

Jetzt sind die Klauseln wieder vertauschbar. Der Nachteil besteht darin, daß der Test, falls er mißlingt, zweimal ausgewertet werden muß.

9.6 Hilfsargumente

Häufig läßt sich die Effizienz von Programmen durch die Einführung von Hilfsargumenten erheblich steigern. Differenzlisten sind ein Beispiel dafür. Diese Hilfsargumente sollten durch eine weitere Klausel nach außen abgeschottet werden.

9.6.1 Beispiel: Grammatik

Ist S das Startsymbol, so sollte die Definition von S so aussehen:

```
s(Satz) :- s(Satz, []) .    Abschottung des 2.
```

```
                            Differenzlistenarguments
s --> np, vp .
   .                        Grammatikregeln
   .
   .
```

9.6.2 Beispiel: Das schnellste 'reverse'-Prädikat

```
reverse(Liste, Ergebnis) :-                Abschottung des
  reverse(Liste, [], Ergebnis).            Hilfsarguments

reverse([], Ergebnis, Ergebnis).

reverse([E|Rest], Hliste, Ergebnis) :-
  reverse(rest, [E|Histe], Ergebnis).
```

In diesem Beispiel wird die Liste im ersten Argument beim Rekursionsabstieg auf das Hilfsargument umgestapelt und dann im dritten Argument als Ergebnis zurückgeliefert.

9.7 Metainterpreter

Es ist sehr einfach, in Prolog selbst einen Prolog-Interpreter zu programmieren

```
beweise([]).

beweise([Goal|Goals]) :-
  klausel(Head, Subgoals),
  unifiziere(Goal, Head),
  beweise(Subgoals),
  beweise(Goals).

unifiziere(X, X).

klausel(...)       Wissensbasis
  ....
```

Dieser Metainterpreter benutzt die Kontrollstruktur und die Unifikationsroutine des unterliegenden Prolog-Systems. Diese beiden Komponenten lassen sich aber auch ausprogrammieren. Dabei können Kontrollstrukturen und Unifikationsverfahren verwendet werden, die von der klassischen Prolog-Realisation abweichen (etwa Breitensuche). Aber auch ganz andere Logiken, mit anderen Beweiskalkülen, lassen sich auf ähnliche Weise realisieren (etwa die gesamte Prädikatenlogik).

In KI-Programmen ist es daher ein übliches Verfahren, nicht Prolog direkt als Wissensrepräsentationssprache zu wählen, sondern

- eine dem Problem angepaßte Repräsentationssprache zu spezifizieren,
- Die Ausdrücke dieser Sprache als Prolog-Terme zu repräsentieren und
- den Kalkül zum Umgang mit den Ausdrücken dieser Sprache als Metainterpreter zu realisieren.

9.8 Prozedurale Programmstrukturen

Nicht alle Programmteile lassen sich mit dem deklarativen Anspruch von Prolog realisieren. Ein- / Ausgabe ist ein Beispiel für einen Bereich, der prozedural realisiert werden muß.

Prolog stellt kein eigenes Programmkonstrukt für die prozedurale Programmierung zur Verfügung. Man muß die starre Abarbeitungsmethode von Prolog für Klauseln für die Realisierung prozeduraler Programmteile benutzen. Befehle werden als Prädikate mit Seiteneffekten realisiert.

Es gibt zwei Möglichkeiten für Schleifen

1. Rekursion

   ```
   q(... ).                 Schleifenabbruch

   q(... ) :-
       .                    Schleifenrumpf
       .
       .
     q(...) .
   ```

   Die Argumente von q übernehmen die Funktion der Schleifenvariablen.

2. Backtracking

   ```
   ...
   q(...)
       .                    Schleifenrumpf
       .
       .
     e(...)                 Schleifenende
   ...
   ```

q muß in dieser Version als Generator für die Schleifenvariablen fungieren, e testet auf das Schleifenende. Eine Variante:

```
 ...
 q(...)
    .                   Schleifenrumpf
    .
    .
   fail
```

Das Schleifenende wird durch ein 'fail' von q realisiert.

Kapitel 10

Komplexere Programme

Im folgenden wollen wir uns einige umfangreichere Beispielprogramme anschauen. Dazu fehlen uns allerdings noch ein paar Features, die Standard-Prolog zur Verfügung stellt.

Wenn in einem Programm z.B. Daten durch Kommunikation mit dem Benutzer oder durch Berechnung des Programms entstehen, so möchte man diese oftmals in die Wissensbasis aufnehmen, so daß sie ab diesem Zeitpunkt für den Beweis neuer Ziele zur Verfügung stehen. Häufig handelt es sich hier um einfache Fakten, wie `edge(a, b)` oder `an_der_reihe(spieler_1)`

Es können aber im Verlauf eines Programms auch Regeln erzeugt werden, die im weiteren Programmverlauf angewendet werden sollen. Auch solche komplexeren Regeln lassen sich in Standard-Prolog in die Wissensbasis eintragen. Das Faktum `edge(a, b)`, welches bedeuten soll, daß eine Kante zwischen a und b besteht, läßt sich folgendermaßen in die Wissensbasis eintragen:

```
assert(edge(a, b)). (Eingabe des Benutzers)
edge(X, Y). (Eingabe des Benutzers)
X=a (Ausgabe des Systems)
Y=b
```

Das Prädikat `append` könnte man auf folgende Weise in die Wissensbasis eintragen:

```
assert(work:pend([], L, L)).
assert((pend([X|R1], Y, [X|R]) :-
      pend(R1, Y, R))).
```

Es wird hier `pend` genannt, um sicher zu stellen, daß es nicht schon in der Wissensbasis vorhanden war. Das hier einmal das Modul 'work' angegeben wurde und einmal nicht, macht in diesem Fall keinen Unterschied, da wir kein anderes Modul aktiviert haben und so das Modul 'work' aktiv ist. Gibt man einen

Modulnamen an, so wird in diesem Modul das angegebene Axiom eingetragen, ansonsten wird in das aktuelle (gerade aktive) Modul eingetragen. Nun wollen wir testen, ob unser Prädikat auch wirklich in die Wissensbasis eingetragen worden ist:

```
%
pend([1 2 3], [2 3 4], X).
X=[1 2 3 2 3 4]
```

10.1 Nimm-Spiel

An dem folgenden Programm sollen mehrere wichtige Programmierstrategien in Prolog demonstriert werden.

Der Spielverlauf wird auf einer sehr abstrakten Ebene spezifiziert: Zwei Spieler ziehen abwechselnd (modelliert durch die Prädikate 'ziehen1' und 'ziehen2'. Das Prädikat 'zug' definiert die möglichen Züge.

Die optimale Spielstrategie kann auf einer ebenso abstakten Ebene spezifiziert werden:

Ein Zug ist für einen Spieler optimal, falls er in eine Spielsituation führt, in der der Gegner keinen optimalen Zug spielen kann.

Realisiert ist im folgenden Programm eine Version des sogenannten Nimmspiels, welches aus folgenden Regeln besteht:

- Die Anfangssituation besteht aus einer Reihe von Haufen, die jeweils aus einer vorzugebenden Anzahl von Streichhölzern bestehen, wobei jeder Haufen eine andere Anzahl an Streichhölzern umfassen kann.
- Ein Zug besteht darin, aus einem beliebigen Haufen mindestens ein Streichholz zu nehmen.
- Gewonnen hat, wer das letzte Streichholz nimmt.

```
nimm :-
        vctell('Nimm-Spiel'),
        assert(spielstatus(noch_leer)),
        anfang(Anfang),                 % Anfangskonfiguration holen
        vcwrite('Anfang: '),
        vcwrite(Anfang), vcnl,
        eingabe2(Anfang),               % Erster Zug des Computers
        spiele.                         % in die Schleife eintreten


spiele :-
```

```
        retract(spielstatus(ende)),   % Gelingt das?
        vcwrite('Tschuess...<RETURN>'),    % Ja, also verabschieden...
        vcread(_),                    % Auf RETURN warten
        vctold.
spiele :-
        vcwrite('Dein Zug: '),        % Prompt ausgeben
        vcread(Zug),                  % Zug einlesen
        vcnl,
        eingabe(Zug),!,       % verarbeiten
        spiele.                       % und in die Schleife schicken



eingabe(Nachher):-
        spielstatus(Vorher),
        zug(Vorher, Nachher),   % also ein korrekter Zug
        ziehen1(Nachher).
eingabe(_Nachher):-
        vcwrite('Das war kein korrekter Zug!'),
        vcnl.
eingabe2(Anfang) :-           % los gehts, der Computer ist am Zug
        ziehen1(Anfang).


ende(Endmeldung) :-
        % Ende des Spiels
        retract(spielstatus(_)),
        assert(spielstatus(ende)),
        vcwrite(Endmeldung), vcnl.
% der Computer ist am Zug ; unabhängig vom konkreten Spiel .
ziehen1(Vorher) :-
        optimal1(Vorher, Nachher), !,
        % schlecht für den Gegner, es gibt einen optimalen Zug
        ziehen2(Nachher).
ziehen1(Vorher) :-
        % Pech , kein optimaler Zug, also irgend einen
        zug(Vorher, Nachher), !,
        ziehen2(Nachher).
ziehen1(_Ende) :-
        % noch mehr Pech, verloren
        ende('Du hast gewonnen.').



% Der Mensch ist am Zug ; unabhängig vom konkreten Spiel.
ziehen2(Vorher) :-
```

```
        vcwrite('Mein Zug: '),
        vcwrite(Vorher),
        vcnl,
        zug(Vorher, _A),
        retract(spielstatus(_)),
        assert(spielstatus(Vorher)).
ziehen2(_Ende) :-
        % so sollte es sein
        ende('Ich habe gewonnen.').




% optimal für Spieler 1 ; unabhängig vom konkreten Spiel .
optimal1(Vorher, Nachher) :-
        zug(Vorher, Nachher),
        not optimal2(Nachher, _X).

% optimal für Spieler 2 ; unabhängig vom konkreten Spiel .
optimal2(Vorher, Nachher) :-
        zug(Vorher, Nachher),
        not optimal1(Nachher, _X).


% Anfangskonfiguration ; abhängig vom konkreten Spiel .
anfang([[i,i,i],[i,i],[i]]).




% mögliche Züge ; abhängig vom konkreten Spiel .
zug(Vorher, Nachher) :-
    ersetzen(Alter_haufen, Neuer_haufen, Vorher, Nachher),
    nehmen(Alter_haufen, Neuer_haufen).



ersetzen(A, B, [A|Rest], [B|Rest]).
ersetzen(A, B, [X|Rest1], [X|Rest2]) :-
      ersetzen(A, B, Rest1, Rest2).


nehmen([_E|Rest], Rest).
nehmen([_E|Rest], Rest_vom_rest) :-
     nehmen(Rest, Rest_vom_rest).
```

10.2 Übungen:

1. Finde ein anderes Spiel für zwei Personen, bei dem die Spieler abwechselnd Spielzüge unternehmen, bis einer der Spieler gewonnen hat. Implementiere es nach der obigen Strategie. .
2. Ändere das Programm so, daß der Rechner gegen sich selbst spielt, d.h. er soll die Spielzüge von beiden Spielern übernehmen.

10.3 Schaltkreisgenerator

Die Konstruktion elektronischer Schaltungen, insbesondere auch die Konstruktion von integrierten Schaltungen ist ein Bereich, der heute ohne Computerunterstützung kaum noch möglich wäre.

Das folgende Programm konstruiert die Realisierung boolescher Funktionen mit Bausteinen (IC's) einer vorgegebenen Bausteinbibliohek auf einem beschränkten Platz (die maximale Anzahl von Bausteinen wird vorgegeben).

Das Programm versucht dabei, bereits vorhandene Schaltungsteile mehrfach auszunutzen und nicht vorhandene boolesche Operationen durch äquivalente Umformungen vermittels vorhandener Gatter (elektronische Realisierungen boolescher Operationen) zu realisieren.

```
  baugruppe(Baugruppe, Funktionen) :-
      konstruiere_liste(Schaltung, Baugruppe, Funktionen),
      write('Baugruppe: '), nl, nl,
      write('Baustein'), nl,
      tab(3), write('Gatter'), nl,
      tab(7), write('Eingaenge'), nl,
      ausgabe(Baugruppe), nl,
      write('Funktionen :'), nl, nl,
      ausgabe_funktionen(Schaltung, Funktionen) ,!.


% Bausteinbiliothek .
% ttl7400 enthält vier nand Gatter mit je zwei Eingängen .

  baustein(ttl7400, [ s(nand,[_1, _2]), s(nand,[_3, _4]), s(nand,[_5, _6]),
```

```
s(nand,[_7, _8]) ] ).


% aequivalente Umformungen boolescher Operationen .

  aequivalent( s(not, [s(and, Argumente)]), s(nand, Argumente) ).
  aequivalent( s(and, Argumente), s(not, [s(nand, Argumente)])) .
  aequivalent( s(not, [Argument]), s(nand, [Argument, Argument])).
  aequivalent( s(or, [Arg1, Arg2]), s(nand, [s(not, [Arg1]), s(not, [Arg2])])
).
  aequivalent( s(xor, Argumente), s(and, [s(or, Argumente), s(not, [s(and,
Argumente)])])).



% konstruiert eine liste boolescher Funktionen .

  konstruiere_liste([], _Baugruppe, []).

  konstruiere_liste([Schaltung | RestS], Baugruppe, [Funktion | RestF]) :-
konstruiere( Schaltung, Baugruppe, Funktion),
konstruiere_liste( RestS, Baugruppe ,RestF).



% realisiert eine Funktion (3. Argument)

  konstruiere( inp( Input), _Baugruppe, inp( Input)).

  konstruiere((B_NR, G_NR, s(Operation, Eingaenge)), Baugruppe,
s(Operation, Argumente)) :-
        baustein( Typ, Gatterliste),
        enthaelt( Gatterliste, _1, s(Operation, Eingaenge)),
        konstruiere_liste( Eingaenge, Baugruppe, Argumente),
        enthaelt( Baugruppe, B_NR, ic(Typ, Gatterliste1)),
        baustein( Typ, Gatterliste1),
        enthaelt( Gatterliste1, G_NR, s(Operation, Eingaenge)).


  konstruiere( Schaltung, Baugruppe, Funktion) :-
        aequivalent( Funktion, Funktion1),
        konstruiere( Schaltung, Baugruppe, Funktion1).



% das gleiche wie element oder member , liefert zusaetzlich den index .
```

```
  enthaelt(Liste, Pos, Element) :- enthaelt1(Liste, Element, 0, Pos).

  enthaelt1([X|_], X, Akku, Pos) :-  % in Akku befindet sich der letzte
        Pos is Akku + 1.               % Index
% der gesuchte Index ist Akku
% + 1

  enthaelt1([_|R], X, Akku, Pos) :-
Akku2 is Akku +1,% Akku um 1 erhöhen, Ergebnis in Akku2
        enthaelt1(R,X,Akku2,Pos). % mit Akku2 als Index in die
% Rekursion...



% ausgabe durch eine verschachtelte Backtrackingschleife .

  ausgabe(Baugruppe) :-
enthaelt(Baugruppe, N, Baustein),
nl, write(N), tab(1),
ausgabe_baustein(Baustein).

  ausgabe(_1).



  ausgabe_baustein(Baustein) :-
var(Baustein),
write('frei'), nl, !, fail.

  ausgabe_baustein(ic(Typ, Gatterliste)) :-
      write(Typ), nl,
      ausgabe_gatter(Gatterliste).



  ausgabe_gatter(Gatterliste) :-
      enthaelt(Gatterliste, N, s(Operation, Eingaenge)),
      tab(3), write(N), tab(1), write(Operation), nl,
      ausgabe_eingaenge(Eingaenge).



  ausgabe_eingaenge(Eingaenge) :-
      enthaelt(Eingaenge, _N, Eingang),
      ausgabe_eingang(Eingang).
```

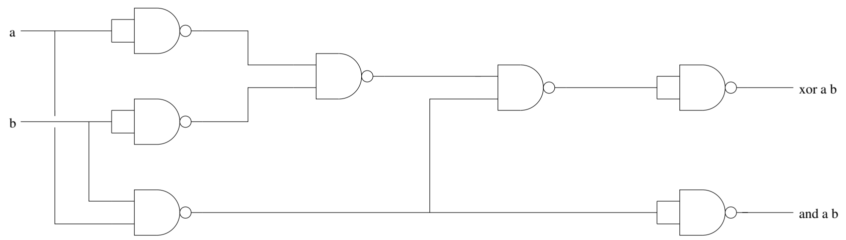
```
  ausgabe_eingang(X) :-
      var(X),
      tab(7), write('frei'), nl, !, fail.

  ausgabe_eingang(inp(X)) :-
      tab(7), write('Input : '), write(X), nl, !, fail.

  ausgabe_eingang((B_NR, G_NR, _1)) :-
      tab(7), write('VON BAUSTEIN '), write(B_NR), write(' GATTER '),
      write(G_NR), nl, fail.



  ausgabe_funktionen([], []).

  ausgabe_funktionen([(B_NR, G_NR, _1) | Schaltung], [Funktion |
Funktionen]) :-
write('BAUSTEIN '), write(B_NR),
        write(' GATTER '),  write(G_NR),
        write(' :'), tab(3),
        write(Funktion),  nl,
        ausgabe_funktionen(Schaltung, Funktionen).
```

Anfrage:

```
baugruppe([_,_,_], [s(xor, [inp(a), inp(b)]), s(and,[inp(a), inp(b)])]).

Baugruppe:

Baustein
   Gatter
       Eingaenge

1 ttl7400
   1 nand
       Input : a
       Input : a
   2 nand
       Input : b
       Input : b
   3 nand
```

```
        VON BAUSTEIN 1 GATTER 1
        VON BAUSTEIN 1 GATTER 2
    4 nand
        Input : a
        Input : b

2 ttl7400
    1 nand
        VON BAUSTEIN 1 GATTER 3
        VON BAUSTEIN 1 GATTER 4
    2 nand
        VON BAUSTEIN 2 GATTER 1
        VON BAUSTEIN 2 GATTER 1
    3 nand
        VON BAUSTEIN 1 GATTER 4
        VON BAUSTEIN 1 GATTER 4
    4 nand
        frei
        frei

3 frei

Funktionen :

BAUSTEIN 2 GATTER 2 :   s(xor, [inp(a), inp(b)])
BAUSTEIN 2 GATTER 3 :   s(and, [inp(a), inp(b)])
```

Die konstruierte Schaltung:



10.4 Eine umfangreichere Grammatik

Hier soll eine etwas umfangreicher DCG-Grammatik für einen Ausschnitt des Deutschen vorgestellt werden. Diese Grammatik ist natürlich nicht in der Lage, kompliziertere syntaktische Konstruktionen korrekt zu analysieren. Insebsondere die freie Wortstellung im Deutschen macht grundsätzliche Probleme bei der Formulierung von DCG-Grammatiken.

```
s --> np,vp,opps.
s --> s,[und],s.

opps --> [].
opps --> pp, opps.

pp --> p, np.

np --> np, pp.
np --> pn.
np --> det, n1.

n1 --> n0.
n1 --> a, n1.

n0 --> n.
n0 --> n, [,], rs, [,].

rs --> relp, opps, iv.
rs --> relp, np, opps, tv1.

vp --> iv.
vp --> tv1, np.
vp --> tv2, np, np.
```

Lexikalische Regeln:

```
iv --> [schlaeft].

tv1 --> [liebt].

tv2 --> [gibt].

det --> [der].
det --> [die].
det --> [das].

pn --> [hans].
pn --> [hamburg].

np --> [frauen].
```

c—c natürlichsprachliche Struktur:
Bedeutungsstruktur:np1 v np2
→ bv(bnp1, bnp2)
np: Nominalphrase
bv: Bedeutung von v
v : Verbalphrase
bnp1: Bedeutung von bnp1
bnp2: Bedeutung von bnp2
name
→ name falls 'name' ein Name ist

n
→ p(X) falls 'p' ein Prädikat ist, das die Bedeutung 'n' darstellt

art n
→ ein_bestimmtes(X, p(X)) falls 'art' ein bestimmter Artikel im Singular ist und 'p(X)' die Bedeutung von 'n' ist

art n
→ alle(X, p(X)) falls 'art' ein bestimmter Artikel im Plural ist und 'p(X)' die Bedeutung von 'n' ist

```
relp --> [der].
relp --> [die].
relp --> [das].

p --> [von].
p --> [nach].
p --> [mit].
```

10.5 Übungen

1. Erweitere die Grammatik um die Möglichkeit, die Bedeutung des Satzes zu erfassen und in die Wissensbasis einzutragen. Das Übersetzen in die Bedeutungsstruktur soll folgendes bedeuten: Beispiel: Hans liebt die Frauen →lieben(Hans, alle(X, frau(X)))

2. Schreibe ein Prädikat, das einen umgangssprachlichen Satz in die oben erklärte Bedeutungsstruktur übersetzt und in die Wissensbasis einträgt. Formuliere ebenfalls ein Prädikat, mit dem überprüft werden kann, ob die Bedeutungsstruktur eines umgangssprachlich eingegebenen Satzes bereits in der Wissensbasis gespeichert ist.

3. Erweitere das Programm derart, daß der gesamte Dialog über ein zu erzeugendes Fenster geführt werden. D.h. der Anwender soll über das erzeugte Dialogfenster einen natürlichsprachlichen Satz eingeben können, von dem dann die Bedeutungsstruktur in die Wissensbasis eingetragen wird. Entprechend soll eine Anfrage gestellt werden können, ob ein im Fenster eingegebener Satz vom System bewiesen werden kann.

Anhang A

Sammlung weiterer Übungen

1. Schreibe ein Prädikat, welches eine Liste (von Listen) in übersichtlicher Form im folgenden Sinne ausgibt:

   Gegeben sei z.B. folgende Liste:

   ```
   [a b [c d [e] f] g]
   ```

   Ausgegeben werden soll folgendes:

   ```
   a
   b
      c
      d
         e
      f
   g
   ```

2. Aufgabe wie 10.2.1. Jedoch sollen diesmal die Klammern der Listen in geeigneter Weise mit ausgegeben werden.

3. Schreibe ein dreistelliges Prädikat findall(G, X, L), welches zu einer Variablen 'X' eines beliebigen Goals G eine Liste 'L' mit allen möglichen Lösungen erzeugt. Das Prädikat 'findall' läßt sich z.B. mit dem Prädikat 'permutiere' oder 'element' folgendermaßen überprüfen:

   ```
   findall(X,element(X, [a b c]), L).
   findall(X,element(X, [a b c]), [a b c]).
   ```

Hinweis: Speichere die Ergebnisse in der Wissensbasis zwischen, wobei darauf zu achten ist, daß keine Veränderungen an Definitionen vorgenommen werden, die noch in Arbeit sind, z.B. beim Backtracking noch benötigt werden.

Anhang B

Literatur

Clocksin/Mellish: Programmieren in Prolog, Berlin, Springer, 1990

Bratko, I.: PROLOG Programming for AI, Bonn, Add. Wesley, 1987

Pereira, F.: PROLOG and Natural Language Analysis, Stanford CA, CSLI, 1987 (Lecture Notes; 10)

König/Seiffert: PROLOG für Computerlinguisten, Tübingen, Francke, 1989 (UTB; 1525)

V. Sperschneider; G. Antoniou: Logic: A foundation of computer science, 1991.