Grundkurs PROLOG für Linguisten

Esther König, Roland Seiffert

ursprünglich erschienen als:
UTB Linguistik, ISBN 3-7720-1749-5
Francke Verlag, Tübingen, 1989

Online-Fassung vom 20. Juni 2003

Inhaltsverzeichnis

Teil I

Grundbegriffe

Kapitel 1

Einführung

delirant logici
Asterix: Die Logiker

Seit vor wenigen Jahren Japan PROLOG zum zentralen Bestandteil seines ehrgeizigen Projektes „Fifth Generation Computing Systems“ gemacht hat, findet diese Programmiersprache plötzlich eine rasende Verbreitung. Die Idee des „*Pro*grammierens in *Log*ik“ – daher der Name PROLOG – soll in allen Bereichen der Künstlichen Intelligenzforschung zu großen Fortschritten führen. Typische Einsatzgebiete für PROLOG sind heute:

- Expertensysteme
- automatisches Beweisen
- maschinelle Übersetzung
- natürlichsprachlicher Zugang zu Datenbanken oder Expertensystemen
- Systeme zur Entwicklung von Grammatiken für natürliche Sprache
- Intelligente Textverarbeitung
- Dialogsysteme
- textverstehende Systeme

In manchen dieser Anwendungsbereiche, z.B. bei der maschinellen Übersetzung, gibt es bereits Programme, die kommerziell eingesetzt werden. Andere Bereiche, wie z.B. Textverstehen, sind noch im Stadium der Grundlagenforschung.
PROLOG ist wesentlich älter als man üblicherweise vermutet. Bereits 1972 wurde der erste PROLOG-Interpreter an der Universität von Aix-Marseille entwickelt. Die wichtigsten Namen, die hier zu nennen sind, sind Colmerauer, Kowalski und Roussel. Ziel der damaligen Entwicklungen war eine Programmiersprache, die gleichzeitig zur Sprachanalyse und zur Implementierung eines Fragebeantwortungsverfahrens geeignet war.

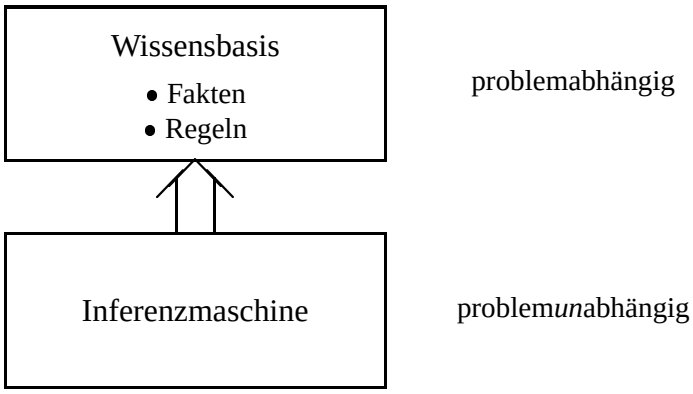


Abbildung 1.1: Modell der PROLOG-Programmierung

1.1 PROLOG-Programmierung

In einer klassischen Programmiersprache wie z.B. PASCAL besteht ein Programm aus einer Folge von *Anweisungen* oder *Befehlen*, die Schritt für Schritt exakt festlegen, was der Computer *tun* soll. In einem Programm wird also genau die Prozedur zur Lösung eines Problems beschrieben; deshalb nennt man diese Sprachen auch *prozedurale Programmiersprachen*.
Die zentrale Frage bei der Erstellung eines PASCAL-Programmes lautet also:

> Was muß ich *tun*, um das gegebene Problem zu lösen?

In PROLOG dagegen bedeutet Programmieren das *Erstellen einer Wissensbasis*, oft auch *Datenbasis* genannt. In dieser Wissensbasis wird alles, was der Programmierer über die Eigenschaften von Objekten einer anwendungsbezogen gewählten Miniwelt und über ihre Beziehungen untereinander weiß, in Form von *logischen Formeln* spezifiziert. Nachdem die Wissensbasis gefüllt ist, besteht die Aufgabe des Programmierers nun im *Stellen von Anfragen* an das PROLOG-System. Jetzt wird die *Inferenzmaschine* aktiv. Die an das System gestellte Anfrage ist nämlich wieder nur eine logische Formel und die Aufgabe der Inferenzmaschine ist es, einen *Beweis* für diese Formel zu finden.
Die zentralen Fragen für den PROLOG-Programmierer sind also:

> Was weiß ich über das gegebene Problem?
>
> Welche Fakten gelten?
>
> Welche Gesetzmäßigkeiten gelten?

Anschließend muß dies alles nur noch in Form logischer Formeln formuliert werden, eine geeignete Anfrage gestellt werden und dann ist es die Aufgabe des PROLOG-Systems, den Weg zur Lösung zu finden.
Es darf niemanden erschrecken, daß jetzt so viel von „logischen Formeln“ und „beweisen“ die Rede war. Es ist absolut nicht nötig, ein tiefes Verständnis für die PROLOG zugrunde liegenden logischen Probleme zu haben, um Programme entwickeln zu können. Wir werden dies an den einführenden Beispielen in den nächsten Abschnitten sofort sehen. Allerdings gibt es einige Probleme bei der PROLOG-Programmierung, die man nur verstehen kann, wenn man fundierte Kenntnisse der Prädikatenlogik besitzt. Im Rahmen einer Einführung spielt dies allerdings keine Rolle.

1.2 Fakten, Regeln und Anfragen

Die Syntax von PROLOG ist sehr einfach. Es gibt nur drei Arten von Ausdrücken in einem PROLOG-System: *Fakten*, *Regeln* und *Anfragen*. Aus Fakten und Regeln wird die Wissensbasis aufgebaut; Anfragen werden dem System als Aufgaben gestellt. Fakten und Regeln bezeichnet man auch als *Axiome*. Axiome sind in der Logik Formeln, von denen man voraussetzt, daß sie wahr sind. Alle Aussagen in PROLOG werden in der besonders einfachen Form der *Klauseln* formuliert. Für Prädikatenlogikexperten: In Klauselform werden Formeln ohne explizite Angabe von Quantoren ausgedrückt. Der Existenzquantor wird durch Skolemisierung eliminiert und alle Variablen in einer Klausel sind implizit allquantifiziert.
Als einführendes Beispiel wollen wir Wissen über die Familie Maier in einer PROLOG-Wissensbasis darstellen.
Familie Maier ist die Traumfamile unseres Bundesfamilienministeriums. Sie besteht aus einem glücklichen Elternpaar, zwei Kindern und einem Hund. Dies alles und noch viel mehr wollen wir in PROLOG ausdrücken.

1.2.1 Fakten

Eigenschaften von Objekten und Beziehungen zwischen ihnen werden in PROLOG als *Fakten* ausgedrückt.
Zunächst stellen wir dar, welche Personen es in der Familie gibt:

```
person(lore).
person(gerd).
person(gitte).
person(uli).
```

Nicht zu vergessen, der Hund Fido:

```
hund(fido).
```

Außerdem wissen wir, daß Gerd der Vater von Gitte und Uli ist. In PROLOG können wir das so formulieren:

```
vater(gerd,gitte).
vater(gerd,uli).
```

Die Bedeutung des Symbols `vater` ist PROLOG natürlich unbekannt. Wir hätten genauso schreiben können

```
x0815(gerd,gitte).
x0815(gerd,uli).
```

und uns merken, daß `x0815` für uns „ist-Vater-von“ bedeutet. Ebenso ist es mit der Reihenfolge der Argumente. Es ist nur unsere Interpretation der Fakten, daß `vater(gerd,uli)` bedeutet „Gerd ist der Vater von Uli“ und nicht umgekehrt „Uli ist der Vater von Gerd“. Wir müssen also die Bedeutung der Prädikate und ihrer Argumente zu Anfang festlegen und sie dann immer genau so verwenden. Es ist deshalb empfehlenswert, im PROLOG-Programm einen *Kommentar* einzufügen, der bei der Einführung eines Prädikates im Klartext aussagt, was gemeint ist.
Ein *Kommentar* ist ein beliebiger Text in einem Programm, der besonders gekennzeichnet ist und von dem PROLOG weiß, daß er lediglich eine Information für den Leser des Programms darstellt und von

PROLOG ignoriert werden darf. In PROLOG hat man meistens zwei verschiedene Möglichkeiten, Kommentare in ein Programm einzufügen. Erstens, durch Einschließen des Kommentars zwischen `/*` und `*/`. Zweitens, durch Schreiben eines `%` wird alles nach dem %-Zeichen bis zum Zeilenende als Kommentar aufgefaßt.
Die Fakten über Familie Maier könnten also mit Kommentaren versehen ungefähr so aussehen:

```
/* Die Personen in Familie Maier sind:
   Lore, Gerd, Gitte und Uli */
person(lore).
person(gerd).
person(gitte).
person(uli).

/* Die Frauen: */
weiblich(lore).
weiblich(gitte).
/* Die Maenner: */
maennlich(gerd).
maennlich(uli).

/* Der Hund heisst Fido */
hund(fido).

/* Die Spielzeugeisenbahn */
spielzeugeisenbahn(d318).

/* Eltern-Kind-Beziehungen */
vater(gerd,gitte).    % gerd ist der vater von gitte
vater(gerd,uli).
mutter(lore,gitte).   % lore ist die mutter von gitte
mutter(lore,uli).
```

1.2.2 Regeln

Beziehungen zwischen Objekten, die nur unter bestimmten Bedingungen gelten, werden in PROLOG durch *Regeln* dargestellt.
Da Familie Maier in einer heilen Welt lebt, gibt es auch noch den alten Brauch, daß jeder Vater seinem Sohn eine Spielzeugeisenbahn schenkt. Diese Regel können wir etwas umformulieren zu:

> X schenkt Y ein Z, *falls* X Vater von Y ist *und* Y männlich ist *und* Z eine Spielzeugeisenbahn ist.

In dieser Form können wir die Regel sofort nach PROLOG übersetzen. Die *falls*-Beziehung heißt in der Logik *Implikation* und wird in PROLOG als `:-` geschrieben. Die *und*-Beziehung heißt *Konjunktion* und wird einfach durch ein Komma `,` in PROLOG ausgedrückt.

Insgesamt erhalten wir also die Regel:

```
/* Jeder Vater schenkt seinem Sohn eine
   Spielzeugeisenbahn */
schenkt(X,Y,Z) :-         % X schenkt Y ein Z, falls
   vater(X,Y),            % X Vater von Y ist und
   maennlich(Y),          % Y maennlich ist und
   spielzeugeisenbahn(Z). % Z eine Spielzeugeisenbahn ist
```

Einen solchen Eintrag in die Wissensbasis nennt man eine *Klausel*. Der Teil vor dem `:-`-Symbol heißt *Kopf* der Klausel, der Teil danach *Rumpf*. Ein einzelner Fakt ist ebenfalls eine Klausel. Wir sagen hier, daß der Rumpf *leer* ist. Wir fassen alle Klauseln mit demselben Kopf zusammen und nennen sie *Prädikat*.

1.2.3 Anfragen

Es ist ja schön und gut, immer mehr Wissen in unser PROLOG-System hineinstecken zu können. Aber natürlich möchten wir auch irgendeinen Nutzen aus den Fakten und Regeln, die in der Wissensbasis gespeichert sind, ziehen können. Kurzum, wir möchtem dem System ein paar Fragen stellen.

- Ist Fido ein Hund?
- Wer ist die Mutter von Uli?
- Wer ist der Vater von Gitte und Uli?
- Wer bekommt von Gerd die Spielzeugeisenbahn „d318“?

Wenn man ein PROLOG-System startet, so meldet sich PROLOG – nach einiger Zeit und einigem anderem Text – immer mit dem Symbol `?-`. Dies deutet an, daß PROLOG bereit ist, vom Benutzer eine Anfrage entgegenzunehmen. Wir können hinter diesem Fragesymbol einen beliebigen Klauselrumpf, d.h. einen Fakt oder eine Konjunktion von Fakten, eingeben. Wenn wir die Eingabe mit einem Punkt und durch Drücken der „Return-Taste“ abgeschlossen haben, wird PROLOG versuchen, diese Formel zu beweisen. Wir wollen also die obigen Fragen einmal an PROLOG stellen:

```
?- hund(fido).
yes
?- mutter(X,uli).
X = lore
?- vater(X,gitte), vater(X,uli).
X = gerd
?- schenkt(gerd,X,d318).
X = uli
```

Das großgeschriebene `X` in den Anfrage und auch andere großgeschriebene Symbole in den obigen Regeln sind sogenannte Variablen. Für Variablen versucht PROLOG einen Wert zu finden.
Wie PROLOG die Lösungen für diese Anfragen herausfindet, werden wir in den nächsten Kapiteln genau untersuchen.

1.3 Übungen

Das Ziel dieser Übungen ist es, sich ein wenig mit der Systemumgebung auf Ihrem Übungsrechner vertraut zu machen. Die wichtigsten Teile eines PROLOG-Systems sind

- der PROLOG-Interpreter
- der Editor

Der PROLOG-Interpreter ist in der Lage, eine Wissensbasis aus einer Datei zu laden und Beweise über dieser Wissensbasis durchzuführen. Der Editor ist ein Programm, mit dem Texte, insbesondere natürlich PROLOG-Programmtexte, am Bildschirm eingegeben und in einer Datei abgespeichert werden können. Leider ist die Bedienung dieser Programme nicht einheitlich. Sie müssen also versuchen, jemanden zu finden, der sich mit Ihrem PROLOG-System schon auskennt und ihn um eine kurze Einführung bitten.

Übung 1.1 Geben Sie das PROLOG-Programm aus diesem Kapitel mit dem Editor ein und speichern Sie es in einer Datei ab. Starten Sie dann den PROLOG-Interpreter. Laden Sie die Datei als Wissensbasis in das System. Stellen Sie die Anfragen dieses Kapitels.

Übung 1.2 Erweitern Sie die Wissensbasis um Fakten und Regeln für folgendes Wissen:

- Lore und Gerd sind verheiratet.[1]
- X ist Großvater von Y, falls Z der Vater von Y ist und X der Vater von Z ist. Vergessen Sie nicht zusätzliche Personen für die Großeltern einzuführen.
- Lore liebt Gerd und Gerd liebt Lore[2]. Gitte liebt Fido.

Stellen Sie neue Anfragen.

[1] Wir leben in einer heilen Welt!
[2] dito

Kapitel 2

Strukturen

Um Informationen irgendwelcher Art darzustellen, werden Strukturen benötigt. Information und Strukturiertheit sind in einem gewissen Sinn äquivalente Begriffe. Das Rauschen eines Radios vermittelt keine Information, die Ansagen des Nachrichtensprechers als hochstrukturierte Abfolge von Signalen tun dies sehr wohl. Der Unterschied zwischen Informationsträchtigkeit und Informationslosigkeit läßt sich an so einfachen Dingen wie Zeichenketten verdeutlichen. Mit der folgenden, „zufälligen“ Buchstabenreihe wissen wir wenig, wenn nicht zu sagen, nichts anzufangen:

tsnlridaoeemtnioitvmnrf

Dagegen ist folgende Zeichenkette für uns informationsvermittelnd, weil sie nach den Gesetzmäßigkeiten der deutschen Sprache (und der ihr vom Lateinischen überkommenen Regeln) gebildet ist:

informationsvermittelnd

Wir beobachten, daß einerseits die Grundstrukturen, die atomaren Einheiten für mögliche Strukturen festgelegt werden müssen. Im Beispiel der Buchstabenketten sind dies die Buchstaben. Natürlich könnte die Ebene der atomaren Einheiten auch „tiefergelegt“ werden und Buchstaben könnten selbst zum Beispiel als in Bezug auf ihr Aussehen strukturiert angesehen werden. Andererseits muß es auch Vorschriften geben, wie Teilstrukturen zueinander in Beziehung gesetzt werden können und damit größere, komplexere Strukturen ergeben. So gibt es in unserem Beispiel der Buchstabenketten (oder eigentlich der durch sie repräsentierten Phoneme) Gesetzmäßigkeiten darüber, wie Silben der deutschen Sprache gebildet werden und darauf aufsetzend, Regeln, wie Silben zu Wörtern zusammengefügt werden können:

in for ma tions ver mit telnd

In der Sprachwissenschaft als empirische Wissenschaft geht es darum, die Regeln zu beschreiben, die der betreffenden Sprache vermutlich zugrunde liegen. In der Informatik und Mathematik geht man den anderen Weg: Wir dürfen nach unserem eigenen Belieben systematisch Strukturen aufbauen. Dazu werden wir möglichst einfache, aber ausdrucksfähige Strukturen wählen wollen.
Um in PROLOG Information zu kodieren, gibt es nur eine sehr einfache Art von Strukturen, die *Terme*. Es lassen sich jedoch sehr viele mathematische Strukturen als Terme darstellen. Im folgenden werden wir die Begriffe „Term“ und „Struktur“ austauschbar verwenden.
Terme werden in drei Klassen eingeteilt. Ein Term ist entweder *atomar* oder *komplex* oder *variabel*.

2.1 Atomare Strukturen

Atomare Strukturen werden per Schreibkonvention gekennzeichnet. In der Regel entsprechen in PROLOG atomare Strukturen „Wörtern“, die mit einem Kleinbuchstaben beginnen oder aus Ziffern bestehen. Mit Kleinbuchstaben beginnende „Wörter“ heißen *Atome*. Mit Ziffern beginnende (und folglich nur aus Ziffern bestehende) „Wörter“ heißen *Zahlen*. Manchmal werden atomare Strukturen auch *Konstanten* genannt. Zählen wir einige Beispiele für atomare Strukturen auf:

```
fido
1238
microProcessor
x4711
```

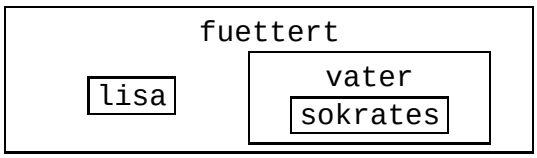
Um Strukturen deutlich voneinander abzugrenzen, kann man sie sich von einem Kästchen umrahmt denken oder sie sich als den „Inhalt“ eines Kästchens vorstellen:

2.2 Komplexe Strukturen

Ein *komplexer Term* (meist kurz auch „Term“ genannt) besteht aus einem *Funktor* und aus einer festen, geordneten Menge von *Argumenten*,. die selbst wiederum Terme sind. Der Funktor muß ein Atom sein. Nach dem Funktor stehen zwischen einer öffnenden und einer schließenden Klammer die Argumente des Terms. Falls mehrere Argumente auftreten, müssen diese durch Kommas getrennt werden. Die Reihenfolge der Argumente ist von Bedeutung! Beispiele für komplexe Strukturen:

```
sterblich(sokrates)
füttert(lisa,fido)
füttert(lisa,vater(sokrates))
```

Wie das dritte Beispiel zeigt, können Terme beliebig tief verschachtelt sein. In der Kästchen-Notation wird diese Verschachtelung besonders deutlich:



2.3 Variable Strukturen

Für Variablen besteht zur Unterscheidung von atomaren Strukturen die Schreibkonvention, daß sie durch „Wörter“ dargestellt werden, die mit einem Großbuchstaben oder dem Unterstreichungszeichen_ beginnen. Beispiele für Variablen:

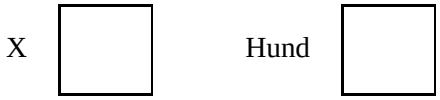
```
X
X4711a
Wer
Hund
HundeHuette
```

```
_
_fido
_Hund
```

Der allein auftretende Unterstreichungsstrich _ heißt *anonyme Variable*. Anonyme Variablen werden verwendet, wenn durch die Stelligkeit eines Prädikats zwar ein Argument verlangt wird, der Name und der Wert dieses Arguments aber in der betreffenden Klausel nicht interessant ist.
Wie auch schon aus der Schulmathematik bekannt, sind Variablen Platzhalter für beliebige Strukturen der entsprechenden formalen Sprache, der wir uns bedienen, wie zum Beispiel der„Sprache“ der arithmetischen Ausdrücke oder eben der Sprache PROLOG. Beispiel aus der Arithmetik:

$$x^2 = -1$$

Variablen kann man sich für die Verarbeitung im Programm als Namen von Kästchen vorstellen, in die zu irgendeinem Zeitpunkt die Belegung, der Wert der Variablen untergebracht wird. Der anonymen Variablen entspricht ein unbenannter Kasten.



Die Sprachkonvention sei, daß Variablen, in deren „Kästen“ noch nichts steht, *uninstantiierte Variablen* heißen.
In PROLOG werden sowohl Prädikate selbst als auch die Argumente von Prädikaten als Termstrukturen dargestellt. Zum Beispiel kann `sterblich(sokrates)` sowohl als Faktum in die Datenbasis eingetragen werden, als auch als Struktur manipuliert werden, d.h. als Argument in einem Prädikat auftreten:

```
chaos(   sterblich(   sokrates ),  lisa      )
         Funktor      Argument
         Argument                   Argument
```

2.4 Übungen

Übung 2.1 In PROLOG gibt es bereits „eingebaute“ Prädikate, die es erlauben, zu überprüfen, ob eine Struktur atomar oder variabel ist: `atomic` und `var`. Beispiele für die Verwendung:

```
?- atomic(werda).
yes
?- atomic(Z).
no
?- var(Z).
yes
?- atomic(a(B)).
no
?- var(a(B)).
no
```

Klassifizieren Sie folgende PROLOG-Terme in eine der drei oben genannten Klassen *atomar*, *komplex* oder *variabel*. Überprüfen Sie die Richtigkeit Ihrer Antworten gleich mit Ihrem PROLOG-System mittels der entsprechenden Prädikate.

1. `Fido`
2. `nummer_24`
3. `hallo_jungs`
4. `hallo(jungs)`
5. `Jungs`
6. `x`
7. `Y`

Übung 2.2 Versuchen Sie, die Information folgender Sätze kompakt als PROLOG-Terme darzustellen:

1. Annette ist mit Johann verheiratet.
2. Ottos Schreibtisch enthält als Schreibmaterial fünf Kugelschreiber, 57 Blatt Papier und als sonstige Dinge acht Zeitschriftenartikel und einen Flaschenöffner.
3. Grammatikalische Kategorien sind Verben, Nomen, Adjektive, Präpositionen, Artikel, Adverben und Partikel. Verben werden unterteilt in ein-, zwei-, drei- und vierstellige Verben. Adjektive werden unterschieden nach pränominal und prädikativ.

Kapitel 3

Unifikation

Um Anfragen an PROLOG-Wissensbasen beweisen zu können, müssen PROLOG-Terme miteinander verglichen werden. Sie werden dabei nicht nur verglichen, sondern, falls möglich, auch tatsächlich miteinander identisch gemacht. Dieses Identisch-Machen wird *Unifikation* genannt. Das Wort„Unifikation“ wurde nicht ohne Grund gewählt: unus heißt eins, facere heißt machen. Unifizieren heißt dann zu eins machen, vereinen, vereinigen. Zur Einführung in die Problemstellung stellen wir uns folgende Wissensbasis, bestehend aus nur einem Fakt, vor:

```
weiblich(lore).
```

Folgende Anfrage läßt sich positiv beantworten, da genau dieser Fakt in der Wissensbasis steht:

```
?- weiblich(lore).
yes
```

Jedoch wird die Frage nach dem Zutreffen des Faktes

```
?- weiblich(lisa).
no
```

negativ beantwortet, weil in der Wissensbasis nicht `lisa` sondern `lore` als Argument des Prädikats `weiblich` erwähnt wird. Es ist keine Unifikation zwischen den beiden verschiedenen Atomen möglich. Falls wir aber das Argument noch nicht auf ein bestimmtes Atom festlegen, sondern eine Variable dafür angeben, bekommen wir eine positive Antwort mit der Information, durch welches Atom die Variable *ersetzt* werden muß:

```
?- weiblich(Wer).
Wer = lore
```

Die drei Beispiele zeigten Aspekte der Unifikation von Variablen und Atomen. Im folgenden wollen wir das Verfahren der Unifikation bezüglich aller verschiedenen Arten von Termen darstellen. Dieses Verfahren testet nicht nur, sondern ersetzt, falls es eine Möglichkeit dazu gibt, die Variablen zweier Terme derart, das beidesmal derselbe Term dasteht. Es liefert sozusagen die„Überdeckung“ zweier Terme, falls diese existiert.
Intuitiv sind zwei atomare Terme trivialerweise gleich, wenn sie eben die gleichen PROLOG-Atomare sind. Eine Variable kann mit jedem beliebigen Term gleichgesetzt werden. Der Vergleich verschachtelter Terme ist etwas komplizierter: Es müssen die Funktoren und die Argumentezahl (*Stelligkeit*) der beiden Terme verglichen werden, wenn dann auch die Argumente in der Reihenfolge ihres Auftretens jeweils mittels derselben Variablenersetzungen unifizierbar sind, sind die beiden Terme insgesamt unifizierbar.

Die Unifikation zweier Terme scheitert in folgenden Fällen:

- es handelt sich um zwei verschiedene atomare Strukturen
- ein Term ist atomar, der andere komplex
- die Unifikation zweier komplexer Terme scheitert, wenn eines der folgenden Kriterien erfüllt ist:
  - die Funktoren der beiden Terme sind verschieden,
  - die Stelligkeiten der beiden Funktoren sind verschieden
  - mindestens ein Argumentepaar von Argumenten gleicher Position im Term läßt sich nicht unifizieren.

Die Aufzählung und Beschreibung der Fälle, in denen die Unifikation zweier Terme erfolgreich verläuft, liefert eine Verarbeitungsvorschrift, d.h. einen *Algorithmus*, für die Durchführung von Unifikationen:

Algorithmus 3.1 *Unifikation*

Eingabe: zwei Terme $Term_1$ und $Term_2$

Ausgabe: die Überdeckung $Term_3$ der beiden Terme, falls diese existiert; sonst die Antwort: `nein`

Methode: Teste, welcher der drei folgenden Fälle für $Term_1$ und $Term_2$ zutrifft, und gib die zugehörige Überdeckung $Term_3$ als Ergebnis zurück:

1. *Ein atomarer Term unifiziert mit demselben atomaren Term:*
   falls $Term_1$ und $Term_2$ dieselben atomaren Terme sind,
   dann ist $Term_3$ gleich $Term_1$
2. *Eine Variable unifiziert mit jedem Term:*
   (a) *falls* $Term_1$ und $Term_2$ beide Variablen sind,
   dann ist $Term_3$ gleich $Term_1$ und $Term_1$ gleich $Term_2$
   (b) *falls* $Term_1$ eine Variable und $Term_2$ ein nichtvariabler Term ist,
   dann sind $Term_3$ und $Term_1$ gleich $Term_2$
   (c) *falls* $Term_2$ eine Variable und $Term_1$ ein nichtvariabler Term ist,
   dann sind $Term_3$ und $Term_2$ gleich $Term_1$
3. *Unifikation komplexer Terme:*
   falls $Term_1$ ein komplexer Term mit Funktor F_1, der Stelligkeit n und den Argumenten $A_{11}, A_{12}, \ldots, A_{1n}$ ist
   und $Term_2$ ein komplexer Term mit Funktor F_2, der Stelligkeit m und den Argumenten $A_{21}, A_{22}, \ldots, A_{2m}$ ist
   und für F_1 und F_2 Fall 1 dieser Unifikationsvorschrift zutrifft
   und die Stelligkeiten gleich sind, d.h. n = m
   und für alle Paare von Argumenten $< A_{11}, A_{21} >, \ldots, < A_{1n}, A_{2m} >$ diese Unifikationsvorschrift „Überdeckungen“ $T_{31}, T_{32}, \ldots, T_{3n}$ liefert, ohne daß die dabei nötigen Variablenersetzungen mehr als einen Wert annehmen müssen
   dann besteht $Term_3$ aus dem Funktor F_1 und den Argumenten $T_{31}, T_{32}, \ldots T_{3n}$.

Ansonsten lassen sich die beiden Terme nicht unifizieren.

Die Unifikation einer Variablen mit einem nicht variablen Term bewirkt die *Instantiierung* dieser Variablen. Instantiierung ist ein neudeutscher Ausdruck, der vom englischen „instantiation“ herstammt. Das Wort „instance“ bedeutet Beispiel. Instantiierung einer Variablen heißt also das Festlegen einer Beispiel-*Belegung* für eine Variable. In einem anderen Programmlauf könnte die Variable ja auch anders belegt werden. Eine Variable, die noch nicht mit einem nicht variablen Term unifiziert worden ist, nennt man *uninstantiiert* .

3.1 Beispiele

Wieder soll uns die Kästchendenkweise zur Veranschaulichung dienen: Die Frage nach der Unifizierbarkeit entspricht in etwa dem folgenden Uberdeckungsproblem:

> Lassen sich zwei Bilder oder Kästchen so übereinander legen, daß sie als eins erscheinen?

Dabei sind, wie im vorherigen Kapitel eingeführt, Variablen Namen von Kästchen, alle anderen Terme mögliche Inhalte von Kästchen. Falls Unifikation durchgeführt werden kann, werden die beiden Objekte identisch. Man kann sich dies ungefähr durch den optischen Effekt veranschaulichen, der entsteht, wenn man zwei auf Transparentfolie gemalte Kästchen übereinanderlegt. Die Kästchen (oder Variablen) haben die Eigenschaft, daß durch Unifikation nur das Hinzufügen von mehr Information bzw. Struktur erlaubt ist, aber daß niemals Information gelöscht werden kann.

Beispiel 3.1 Erfolgreiche Unifikation im atomaren Fall:

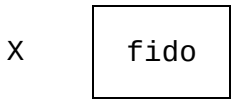
Unifikation („Überdeckung“) ist möglich.

Beispiel 3.2 Scheitern einer Unifikation im atomaren Fall:

Unifikation („Überdeckung“) ist nicht möglich.

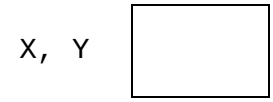
Beispiel 3.3 Unifikation einer uninstantiierten Variablen mit einem Atom:

Das Ergebnis der Unifikation („Überdeckung“) ist:



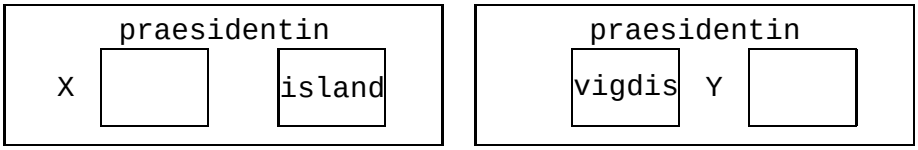
Beispiel 3.4 Unifikation zweier uninstantiierter Variablen:

Das Ergebnis der Unifikation („Überdeckung“) ist:

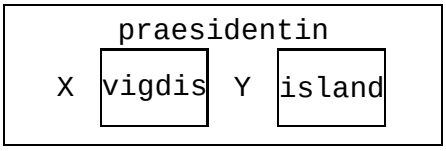


Die Variablen bezeichnen den gleichen Kasten. Wird X später mit irgendeinem anderen Term unifiziert, ist damit gleichzeitig Y auch mit diesem Term unifiziert.

Beispiel 3.5 Unifikation zweier komplexer Terme:



Das Ergebnis der Unifikation („Überdeckung“) ist:



3.2 Übungen

Übung 3.1 Unifizierbarkeit zweier Terme läßt sich direkt im Dialog mit dem PROLOG-System nachprüfen, mittels des eingebauten Prädikats =. Diese Prädikat nimmt die beiden zu vergleichenden Terme als Argumente.

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

Die obige Anfrage läßt sich auch bequemer schreiben als:

```
?- atom = atom.
```

Bei Anfragen mit Variablen, werden die Instantiierungen der Variablen ausgegeben:

```
?- praesident(Wer,Land) = praesident(Person,island).
Wer = Person
Land = island
```

Wenden Sie die Unifikationsvorschrift auf folgende Paare von Termen an, d.h. entscheiden Sie, welche Paare von Termen unifizierbar sind. Geben Sie bei unifizierbaren Termen den betreffenden Unifikator an, bei nicht unifizierbaren die Begründung für das Scheitern der Unifikation.Überprüfen Sie Ihre Vermutungen an Ihrem PROLOG-System. Schlagen Sie zuvor im Handbuch Ihres PROLOG-Systems oder Ihres Rechners nach, wie Programmläufe „gewaltsam“ abgebrochen werden können.

1. `?- _ = X.`
2. `?- X = Y.`
3. `?- a(X,Y,Z) = a(Q,U).`

4. `?- a(X,Y,Z) = a(W,P,t).`
5. `?- a(b(C)) = b(a(C)).`
6. `?- f1(f2(f3,f4),f5(f6,X,Y),g) =`
   `f1(A,f5(B,f7,f8),H).`
7. `?- X = X.`
8. `?- f(X) = X.`
9. `?- f1(f2(f3,f4),f5(f6,X,Y),g) =`
   `f1(A,f5(B,f7),H).`
10. `?- a = a(X,Y).`
11. `?- a = a.`
12. `?- qwertz = qwerty.`

Kapitel 4

Beweisverfahren

Wie in der Einleitung erwähnt, werden Anfragen an eine Datenbasis aufgrund eines *Beweisverfahrens* beantwortet. Wir wollen hier keine formal korrekte Definition von Beweisverfahren für PROLOG liefern, sondern vielmehr etwas Gefühl dafür vermitteln, wie dieses Beweisverfahren in der Praxis abläuft.

Bereits die alten Griechen beschäftigten sich mit der Entwicklung von Schemata, die das systematisieren, was wir *logisches Schließen* nennen. Schlußfolgerungen sollen nicht willkürlich geschehen, sondern durch Befolgen bestimmter Vorschriften nachvollziehbar sein. Ein schon seit den alten Griechen bekanntes Beispiel für eine *Schlußregel* ist der *Modus Ponens*, der in einer verallgemeinerten Form im PROLOG-Beweisverfahren verwendet wird. Schlußregeln beschreiben die Beziehung zwischen einer Menge von Voraussetzungen und der möglichen Folgerung aus diesen Voraussetzungen.

Beispiel 4.1 *Modus Ponens*

Voraussetzungen:	P(X) *falls*	Q(X)
		Q(X)
Folgerung:	P(X)	

Beispiel 4.2 Anwendung des Modus Ponens:

X ist sterblich,	*falls*	X ein Mensch ist.
		Sokrates ist ein Mensch.
Sokrates ist sterblich.		

Die Voraussetzungen des Beispiels stellen, nach PROLOG übersetzt, eine kleine Datenbasis dar:

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

An diese Datenbasis können nun Anfragen gestellt werden, der Form:

„Beweise mir:“
bzw. „Läßt sich aus der Datenbasis folgern:“

```
?- sterblich(sokrates).
```

Da wir beweisen lassen wollen, daß diese Behauptung aus den Klauseln der Datenbasis als den Voraussetzungen folgt, wird die Schlußregel des Modus Ponens „rückwärts“ angewandt werden: Wir kennen das Aussehen der Folgerung und stellen anhand der Schlußregel fest, welche Aussagen bewiesen sein müssen, damit diese Folgerung bewiesen ist. Die Anfrage `sterblich(sokrates)` läßt sich mit dem Kopf der Klausel `sterblich(X):-mensch(X)` unifizieren und führt zu der Belegung `X=sokrates`. Dadurch wird klar, welche weitere Voraussetzung bewiesen werden muß:

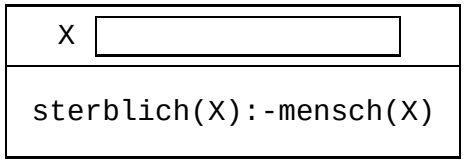
```
mensch(sokrates).
```

Dieser Unterbeweis läßt sich in trivialer Weise durchführen, da dies als Fakt in der Datenbasis steht.

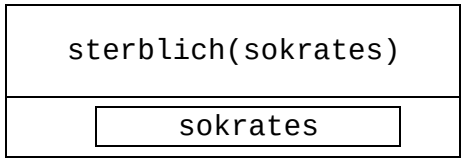
4.1 Beweisverfahren und Unifikation

Bei Betrachtung des Beweisverfahrens wird auch ersichtlich, wo in PROLOG die Unifikation zum Zug kommt: Weitere, gemäß Modus Ponens zu beweisende Voraussetzungen werden mit der Unifikation als Testkriterium in der Datenbasis aufgefunden. Die zu beweisende Behauptung muß mit dem *Kopf* einer Klausel in der Datenbasis unifizierbar sein. Diese Unifikation liefert Variablen-Instantiierungen für weiterhin zu beweisende Aussagen im *Rumpf* dieser Klausel.
Für unser einfaches Beispiel kann uns die Kästchen-Denkweise nocheinmal zur Veranschaulichung des Geschehens dienen. Bei komplizierteren Fällen stößt das Kästchen-Paradigma jedoch auf seine Grenzen. Für den einfachen Fall nun besitze jede Klausel für jede Argumentstelle ihres Kopfes ein Kästchen, das für den auf dieser Argumentstelle sitzenden Term stehen soll. Falls also auf einer Argumentstelle eine Variable steht, wird ein mit dem Namen der betreffenden Variablen gekennzeichnetes Kästchen eingeführt. Beachte, daß die Kästchen und deren Namen lokal zur Klausel sind, in der sie vorkommen. Variablennamen gelten also nicht klauselübergreifend. Werden in verschiedenen Klauseln die gleichen Namen benutzt, sind diese als verschiedene Variablen, d.h. verschiedene Kästchen anzusehen. DerÜbersichtlichkeit wegen zeichnen wir bei Klauseln der Datenbasis die Kästen oberhalb des eigentlichen Texts der Klausel, bei Anfragen an die Datenbasis unterhalb des zu beweisenden Fakts.

Beispiel 4.3 Klausel mit den zugehörigen Kästchen:



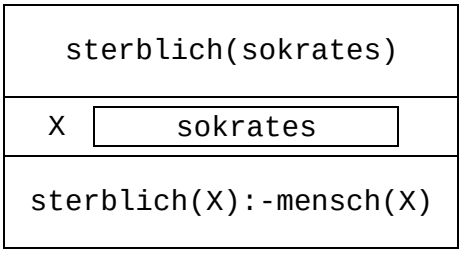
In Kästchen-Notation lautet die Anfrage (*Zielklausel*), die wir betrachten wollen:



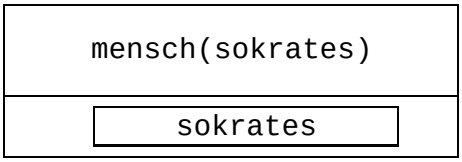
Verfolgen wir noch einmal den Beweisverlauf für diese Zielklausel, um zu sehen, wie die Variablen verschiedener Klauseln durch Unifikation zueinander in Bezug gesetzt werden. Passend zur Anfrage finden wir in der Datenbasis die Klausel

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

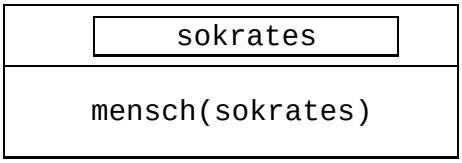
Die Unifikation der einzigen Argumentstelle der Anfrage und der des Kopfes der Klausel in der Datenbasis



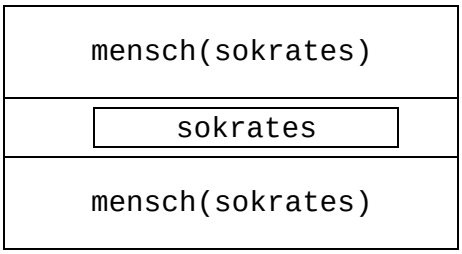
liefert die Variablen-Instantiierung („Füllung des Kästchens“) für die nach dem Schema des Modus Ponens noch zu beweisende Aussage:



Um diese *Unteranfrage*, d.h. vom System erzeugte Anfrage zu beweisen, greifen wir auf den Fakt zurück:



Dieser Fakt läßt sich spielend mit der neuerzeugten zu beweisenden Anfrage unifizieren:



Damit ist der Beweis dafür, daß Sokrates sterblich ist, erfolgreich abgeschlossen.

Wir fassen zusammen: Durch Unifikation wird Information über Klausel-Grenzen hinweg transportiert und „in miteinander unifizierten Kästen“ angesammelt. „Ansammeln“ beinhaltet auch, daß die „Transportrichtung“ der Information nicht festgelegt ist. Einerseits kann Information von einer Anfrage in eine Klausel eingebracht werden, als Rahmenbedingungen, unter denen der Beweis weitergeführt werden muß. Andererseits kann auch Information aus einer Klausel der Datenbasis auf die Anfrage übertragen werden und somit eine „Antwort“ in Form von Instantiierungen der Variablen der Anfrage erzeugt werden.

4.2 Backtracking

In den meisten Fällen laufen jedoch Beweise nicht so geradlinig und problemlos ab. Nehmen wir folgende Datenbasis an:

```
schwabe(X) :-
   ort(Y),
   wohnt(X,Y).
ort(aalen).
ort(lorch).
wohnt(lothar,aalen).
wohnt(gustav,lorch).
wohnt(adele,lorch).
```

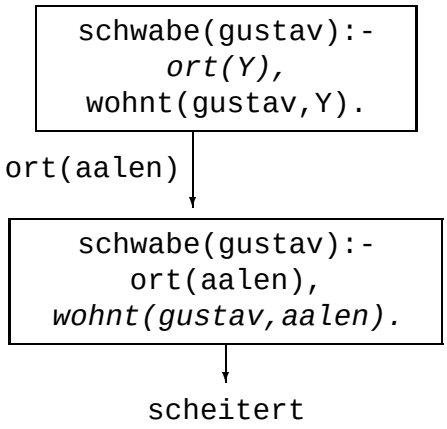
Die Klauseln und Fakten sind in einer bestimmten Reihenfolge angeordnet, die Suche nach einem Beweis basiert auf dieser Reihenfolge. Wollen wir zum Beispiel nachprüfen lassen, ob Gustav gemäß obiger Datenbasis ein Schwabe ist, passiert folgendes: `schwabe(gustav)` paßt auf den Kopf der Klausel:

```
schwabe(X) :-
   ort(Y),
   wohnt(X,Y).
```

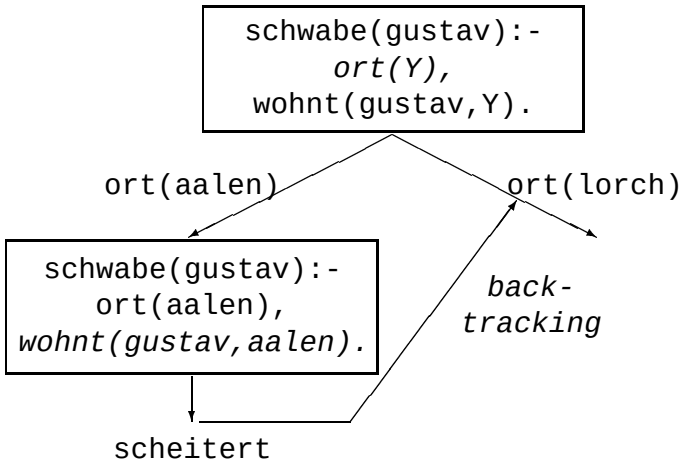
Um `schwabe(gustav)` zu beweisen, muß also zuerst `ort(Y)` gezeigt werden, d.h. es muß ein möglicher Wohnort geraten werden werden. Als erster Fakt über Wohnorte ist eingetragen:

```
ort(aalen).
```

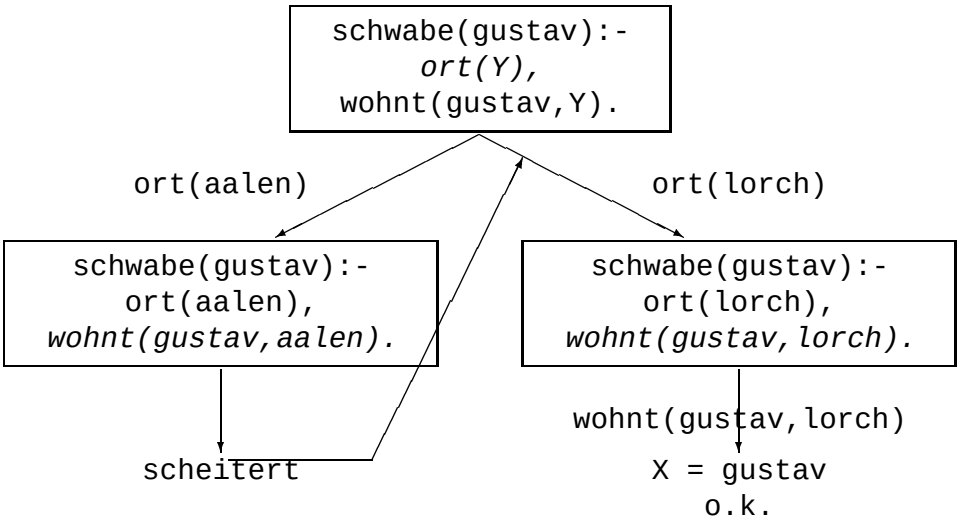
Es bleibt zu zeigen, daß gilt `wohnt(gustav,aalen)`. Dies ist jedoch nicht möglich. Der Teilbeweis `wohnt(gustav,aalen)` scheitert. Die Beweissuche ist in eine Sackgasse geraten.



Man soll nicht gleich aufgeben, wenn man in eine Sackgasse gerät. Häufig hilft es, zum letzten Verzweigungspunkt zurückzugehen und eine der anderen Möglichkeiten auszuprobieren. Die Beweissuche muß also zurück an die Stelle, an der die Entscheidung zwischen `ort(aalen)` und `ort(lorch)` getroffen wurde, und ihr Glück mit der zweiten Alternative versuchen. Dieses *Zurücksetzen* zum letzten Verzweigungspunkt heißt auf Neudeutsch auch *backtracking* . Beim Zurücksetzen wird die alte Instantiierung der Variablen X in der Schwabenklausel „vergessen“ und eine neue Instantiierung `X=lorch` angenommen.



Die zweite Alternative ist angemessener und erlaubt einen erfolgreichen Abschluß des Beweises, daß Gustav ein Schwabe ist:

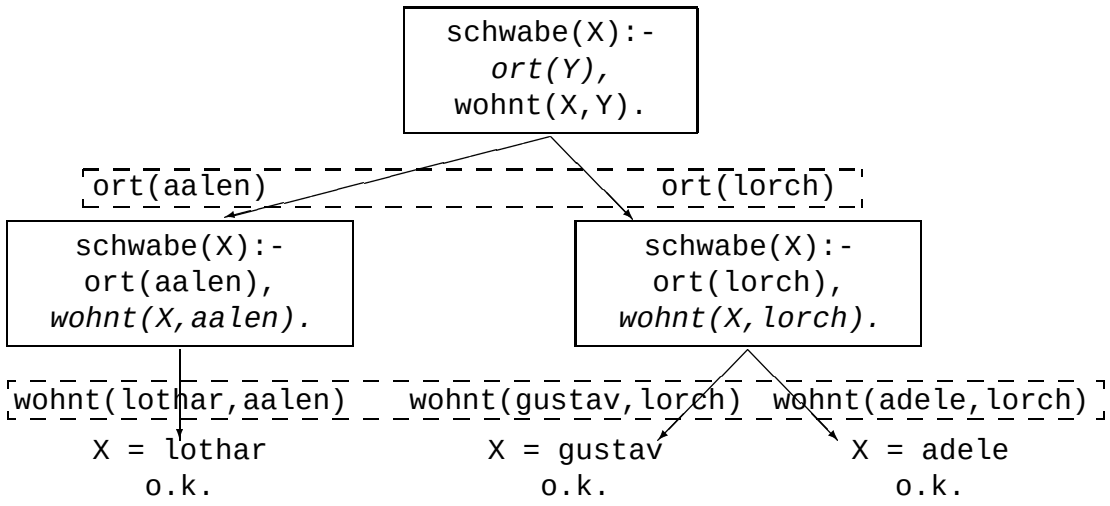


Im eben gezeigten Fall trat ein *internes Scheitern* eines Teilbeweises auf. Wir können aber auch„von außen“ die PROLOG-Maschine zum Backtracking zwingen.

Beispiel 4.4 Stellen wir an obige Datenbasis folgende Frage:

```
?- schwabe( Wer ).
Wer = lothar;
Wer = gustav;
Wer = adele;
no
```

Der Strichpunkt `;` ist jeweils Benutzereingabe und bewirkt, daß nach einer weiteren Lösung gesucht wird. Falls keine weiteren Lösungen zu finden sind, wird das endgültige Scheitern der Suche mit `no` angezeigt. Die verschiedenen Lösungswege können als Baum von Alternativen dargestellt werden. Statt wie in der Datenbasis vertikal sind hier die alternativen Fakten eines Prädikats horizontal angeordnet.



Der Begriff *backtracking* rührt daher, daß sich der Beweiser an den Verzweigungspunkten im Beweis, also an Stellen, in denen mehrere verschiedene Fortsetzungen des Beweises möglich sind, Markierungen setzt, welcher Pfad eingeschlagen wurde, und welche Pfade für die weitere Beweissuche noch zur Verfügung stehen. Scheitert ein Lösungsweg, „setzt“ PROLOG zurück und versucht die nächste mögliche Alternative. Backtracking wird also durch Scheitern eines Teilbeweises ausgelöst. Backtracking läßt alle Variablenbelegungen, die am letzten Verzweigungspunkt gemacht wurden, vergessen und erlaubt die erneute Instantiierung der Variablen mit den anderen möglichen Werten. Dabei ist zu beachten, daß die Datenbasis strikt von oben nach unten durchsucht wird, und daß Unterziele in der Reihenfolge ihrer Erwähnung in der entsprechenden Klausel angestoßen werden.

4.3 Ablaufprotokoll

Die meisten PROLOG-Systeme bieten Möglichkeiten an, die einzelnen Beweisschritte mitzuverfolgen (*trace*). Man gibt ein:

```
trace.
```

und dann zum Beispiel

```
spy schwabe/1.
```

also den Prädikatsnamen und die Stelligkeit des zu untersuchenden Prädikats. Die Beweisschritte werden in vier Klassen eingeteilt: `call, redo, exit, fail.` Der Beginn eines neuen Teilbeweises wird mit `call` gekennzeichnet. Die Markierung `redo` deutet den Versuch der Wiederholung eines Teilbeweises mit alternativer Variablenbelegung an. Mit `exit` wird das erfolgreiche Beenden des gerade bearbeiteten Teilbeweises, mit `fail` das Scheitern dieses Teilbeweises mit der gegenwärtigen Variablen-Instantiierung angezeigt. Das Ablaufprotokoll für das eben besprochene Beispiel sieht auf dem Bildschirm in etwa so aus:

```
call(schwabe(gustav))
    call(ort(Y))
    exit(ort(aalen))
    call(wohnt(gustav,aalen))
```

```
        fail(wohnt(gustav,aalen))
        redo(ort(Y))
        exit(ort(lorch))
        call(wohnt(gustav,lorch))
        exit(wohnt(gustav,lorch))
    exit(schwabe(gustav))
```

4.4 Zusammenfassung

Beweise in PROLOG folgen im Prinzip dem Schema des Modus Ponens. Die *Unifikation* dient hierbei zum Auffinden passender Klauseln in der Datenbasis und somit zum Erzeugen von Unterbeweisen. Da das Beweisverfahren auf Suche beruht, kann es auch in Sackgassen geraten, die dann ein Zurücksetzen (von Variablenbelegungen), d.h. *backtracking* , erfordern. Noch nicht erwähnt wurde, daß das Suchverfahren handelsüblicher PROLOG-Systeme *unvollständig* ist. Das heißt, in einigen Fällen findet der Beweiser keinen Beweis (obwohl nach den Gesetzen der Logik ein Beweis existieren müßte), sondern gerät in eine „unendliche Schleife“, weil er zuerst in die Tiefe des Beweisbaumes sucht und dabei eventuell auf einen Zweig gerät, der sich beliebig verlängern läßt, ohne jemals zu scheitern oder sich erfolgreich abschließen zu lassen (Genaueres hierzu z.B. in [Schöning 1987]).

4.5 Übungen

Übung 4.1 Schlagen Sie in Ihrem PROLOG-Handbuch nach, wie der Trace, das Ablaufprotokoll für Beweise, ein- und ausgeschaltet wird. Normalerweise geschieht dies mittels der Prädikate `trace` und `spy`, `notrace` und `nospy`. Durch `trace` wird ein Ablaufprotokoll ermöglicht, durch `notrace` wieder der Normalzustand hergestellt. `spy schwabe/1` bewirkt, daß mit der nächsten Anfrage zum einstelligen Prädikat `schwabe` das Ablaufprotokoll sichtbar wird, `nospy schwabe/1` entfernt diesen *Spion* wieder vom Prädikat.

1. Definieren Sie sich Datenbasen (d.h. Dateien) mit einigen der bis jetzt erwähnten Programmbeispiele und schauen Sie sich Ablaufprotokolle von Anfragen an diese Datenbasen an.

2. Was geschieht bei der Anfrage

   ```
   ?- raetsel.
   ```

   an folgende Datenbasis:

   ```
   raetsel:-
      was_denn,
      wie_auch_immer.
   was_denn:-
      abxy.
   was_denn:-
      cufg.
   cufg.
   wie_auch_immer.
   ```

3. Wie arbeitet das Prädikat `raetsel(X)`?

```
raetsel(X):-
   was_denn(X).
was_denn(huhn(X)):-
   abrakadabra(X).
was_denn(hahn(X)):-
simsalabim(X).
abrakadabra(ute).
simsalabim(otto).
```

Übung 4.2 Entwerfen Sie eine Datenbasis, die folgende Information enthält:

- Apfelbäume, Stachelbeer- und Himbeersträucher, Kakteen und Hundsrosen sind Pflanzen
- Äpfel sind Früchte von Apfelbäumen, Stachelbeeren von Stachelbeersträuchern, Himbeeren von Himbeersträuchern, Feigen von Kakteen, Hagebutten von Hundsrosen.
- Äpfel, Stachelbeeren, Himbeeren und Feigen sind eßbar
- Etwas kann mit der Bezeichnung „Obst“ versehen werden, wenn es eine Pflanze gibt, dessen Frucht es ist, und wenn es eßbar ist.

Testen Sie die Korrektheit der Datenbasis zum Beispiel mit den Anfragen (mit Backtracking!):

```
?- obst(Was).
?- obst(hagebutte).
```

Übung 4.3 Schreiben Sie mindestens zwei Klauseln, die die Problematik ausdrücken:

> Was war zuerst da - das Huhn oder das Ei?

Verbale Lösung in etwa:

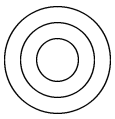
> Ein Huhn ist da, falls ein Ei da ist (war).
> Ein Ei ist da, falls ein Huhn da ist.

Untersuchen Sie das Ablaufverhalten dieser beiden Prädikate.

Kapitel 5

Rekursion

Das zweite wichtige Konzept in PROLOG ist neben der Unifikation die *Rekursion* . Rekursion wird als Prinzip im wesentlichen in der Mathematik eingesetzt, es gibt jedoch auch einige wenige Beispiele aus dem täglichen Leben, die die Idee der Rekursion verdeutlichen.
Eine wunderschöne Realisierung von Rekursivität sind die „Matruschkas“, die russischen Puppen. Man öffnet die äußerste Puppe und erblickt die gleiche Puppe wieder, nur eine Nummer kleiner. Auch diese Puppe kann man öffnen usw., bis man bei der kleinsten, der unzerlegbaren („atomaren“) Puppe angelangt ist. Die Draufsicht auf eine Matruschka wäre ungefähr so:



In PROLOG (und auch in anderen Programmiersprachen) lassen sich vergleichbare Strukturen aufbauen. Bevor wir jedoch unser erstes rekursives Prädikat formulieren, sollten wir dessen Vorgehensweise verbal beschreiben:

- Innerste, „unzerlegbare“ Puppen entsprechen in PROLOG *atomaren Strukturen* . Wählen wir als Beispiel das Atom `emma`.
- Jede größere Puppe besteht aus ihrer nach außen sichtbaren Hülle und der nächstkleineren Puppe in ihr. Die Einbettung von kleineren in größere Strukturen findet sich in *komplexen Termen* wieder. Eine nicht-atomare Puppe soll zum Beispiel die Struktur `emma(Puppe)` besitzen, wobei der Funktor `emma` der äußeren Puppe entspricht, die Variable `Puppe` für die darin enthaltene Puppe steht, über deren genauere Struktur wir ja nicht unbedingt Bescheid wissen.

Eine Puppe der Verschachtelungstiefe 2 werde dann z.B. so dargestellt: `emma(emma(emma))`. Das Prädikat, das obige verbale Beschreibung realisiert, lautet:

```
puppe(emma).

puppe(emma(Puppe)):-
   puppe(Puppe).
```

„Sinnvolle“ rekursive Prädikatsdefinitionen zeichnen sich dadurch aus, daß sie mindestens zwei Klauseln haben, d.h. mindestens zwei Fallunterscheidungen berücksichtigen. Zuerst erwähnt wird meistens die *Abbruchbedingung* . Die Abbruchbedingung heißt deswegen so, weil das zu definierende Prädikat nicht mehr im Rumpf dieser Klausel erwähnt wird, also kein *rekursiver Aufruf* mehr stattfindet. Die andere Klausel beinhaltet die Rekursion: das Prädikat ruft sich selbst wieder auf.

5.1 Die Abbruchbedingung

Was geschieht bei einem rekursiv definierten Prädikat, das keine explizite Abbruchbedingung enthält? Nehmen wir einmal an, das Prädikat `puppe` sei nur durch folgende Klausel definiert:

```
puppe(emma(Puppe)):-
   puppe(Puppe).
```

Die Anfrage: `puppe(emma(emma(emma))).`
wird beantwortet mit: `no`
Die Definition ist also ohne eine *explizite Abbruchbedingung* unvollständig. Aufgrund der *impliziten Abbruchbedingung* , die dadurch entsteht, daß der Fall einer atomaren Puppe nicht vorgesehen ist, folgt zumindest die Antwort `no`. Schwerwiegender wird das Fehlen der Abbruchbedingung, wenn die Anfrage mit einer Variablen statt mit einer vollständig spezifizierten Struktur versehen ist:

```
?- puppe(Puppe).
```

Hier kommt überhaupt keine Antwort vom Beweiser zurück, weil die Beweisprozedur in eine *Schleife* gerät. Es wird versucht, die „unendlich“ große Matruschka aufzubauen.

5.2 Der rekursive Aufruf

So wie das Fehlen von Abbruchbedingungen für rekursive Prädikatsdefinitionen für die Beweissuche „tödlich“ sein kann, ist auch nicht jede Art von rekursivem Aufruf sinnvoll. Verändern wir die Argumente in der rekursiven Klausel wie folgt:

```
puppe(emma).
puppe(Puppe):-
   puppe(emma(Puppe)).
```

Stellen wir an diese Datenbasis die Anfrage

```
?- puppe(emma(emma(emma))).
```

gelangt die Beweissuche wiederum in eine „unendliche“ Schleife, weil mit jedem rekursiven Aufruf des Prädikats `puppe` die Struktur dessen Arguments vergrößert statt verkleinert wird. Um zu beweisen, daß die Struktur `emma(emma(emma))` der Definition von `puppe` genügt, wird versucht, dies für alle beliebig großen Puppen zu beweisen.
Wir folgern daraus, daß rekursive Prädikatsdefinitionen nur dann Sinn machen, wenn die Argumente des rekursiven Aufrufs mit „kleineren“ Strukturen instantiiert sind als die Argumente im Prädikatskopf. Rekursive Aufrufe mit der gleichen oder gar einer größeren Struktur führen im allgemeinen in eine Schleife.

5.3 Beispiel: Wegsuche

Nehmen wir an, wir wollten anhand des folgenden Ausschnitts aus einem Stadtplan den Weg vom Bahnhof zur Universität finden. Wenn wir einmal die Richtung festgestellt haben, in die wir gehen müssen, wird unsere Aufgabe darin bestehen, den noch verbliebenen Weg zu unserem gegenwärtigen Standort schrittweise zu verkleinern. Oder „rekursiv formuliert“:

- Die Wegsuche endet, wenn unser Standort mit dem Zielpunkt identisch ist.
- Wegsuche besteht darin, die erste Teilstrecke des Weges zu finden, und dann vom neuen Standort aus die Wegsuche erneut zu starten.

Beispiel 5.1 *Stadtplan*

Da PROLOG keine solchen übersichtlichen graphischen Darstellungen zuläßt, müssen wir zuerst daran gehen, die Information aus der Karte in eine Liste von Fakten zu übersetzen. Ein Fakt der Art `weg(Punkt1,Punkt2)` beschreibt die Verbindung zwischen zwei Punkten wie zum Beispiel einer Kreuzung oder einem markanten Gebäude. Die Kreuzungen selbst werden dargestellt durch Terme `kreuzt(Strasse1,Strasse2)`.

```
weg(bahnhof,kreuzt(kronenstr,lautenschlagerstr)).
weg(kreuzt(kronenstr,lautenschlagerstr),
    kreuzt(friedrichstr,kronenstr)).
weg(bahnhof,kreuzt(friedrichstr,kriegsbergstr)).
weg(kreuzt(friedrichstr,kriegsbergstr),
    kreuzt(friedrichstr,kronenstr)).
weg(kreuzt(friedrichstr,kronenstr),uni).
weg(kreuzt(friedrichstr,kronenstr),
    kreuzt(kriegsbergstr,kronenstr)).
weg(kreuzt(kriegsbergstr,kronenstr),
    kreuzt(keplerstr,kriegsbergstr)).
weg(kreuzt(keplerstr,kriegsbergstr),uni).
```

Wir erweitern diese Datenbasis noch um eine rekursive Prädikatsdefinition, die uns nach oben genannten Gesichtspunkten die Suche nach einem Weg beschreibt. Das Prädikat `wegsuche` ist dreistellig. Sein erstes Argument beinhaltet den gegenwärtigen Standort, das zweite Argument dient zum Mitführen der Information über den Zielpunkt, das dritte Argument hat die Funktion, die Information über den Wegverlauf „einzusammeln“.

```
wegsuche(Punkt,Punkt,Punkt).
wegsuche(Anfang,Ende,weg(Anfang,NaechsterWeg)) :-
   weg(Anfang,NaechsterPunkt),
   wegsuche(NaechsterPunkt,Ende,NaechsterWeg).
```

Rekursion wird hier „vernünftig“ verwendet: Die rekursive Klausel „verkürzt“ jeweils die noch zu suchende Wegstrecke um eine Stufe. Die Gefahr, daß die Beweissuche in eine Schleife gerät, ist, zumindest von dieser Seite her, gebannt. Wie für „vernünftig“ angewandte Rekursion gefordert, wird die Aufgabe der Bearbeitung einer Struktur auf die Aufgabe zur Bearbeitung einer Teilstruktur davon zurückgeführt.

5.4 Übungen

Übung 5.1 Lassen Sie sich von `wegsuche` den Weg vom Bahnhof zur Universität suchen (mit Ablaufprotokoll):

```
?- wegsuche(bahnhof,uni,Weg).
Weg = ...
```

Übung 5.2 Beschreiben Sie rekursiv die Hierarchiestufen in einer Firma mit einer sehr rigiden Organisationsform:

- Eine Managerin hat immer zwei untergebene Managerinnen.
- Auf unterster Führungsebene unterstehen einer Managerin jeweils eine bestimmte Anzahl von Mitarbeiterinnen.

Aus einer Grundmenge von Mitarbeiterinnen und Chefinnen sollen sich durch Backtracking verschiedene (im Prinzip unendliche viele) personelle Konfigurationen herstellen lassen.

Übung 5.3 Erweitern Sie die Prädikate zum Hühner-Ei-Problem des vorigen Kapitels um (willkürliche) Abbruchbedingungen, zum Beispiel um Angaben über ein Urhuhn, das vom Himmel fiel. (Wo bleibt der Hahn?)

Übung 5.4 Erweitern Sie für das Problem `wegsuche` die Menge der Fakten über mögliche Verbindungen. Lassen Sie zu, daß einzelne Verbindungen auch in der Gegenrichtung begangen werden können. Untersuchen Sie jetzt das Ablaufverhalten von

```
wegsuche(Start,Ende,Weg).
```

Welches Problem ist entstanden? Machen Sie sich Gedanken, wie dem abgeholfen werden könnte.

Übung 5.5 Schreiben Sie ein Prädikat

```
simple_term(SimpleTerm)
```

das nachprüft, ob ein Term zur speziellen Menge der SIMPLE-Terme gehört bzw. das SIMPLE-Terme erzeugt. Das Atom `argument` ist ein SIMPLE-Term. Komplexe SIMPLE-Terme haben als Funktor `functor` und als einziges Argument einen SIMPLE-Term. Tests:

```
?- simple_term(Term).
Term = argument;
Term = functor(argument);
Term = functor(functor(argument))
yes
?- simple_term(hallo).
no
?- simple_term(functor(argument,functor)).
no
?- simple_term(a23489(hallo)).
no
```

Kapitel 6

Listen

Im letzten Kapitel wurden Wegverläufe etwas umständlich als Terme kodiert. Wegverläufe als Aufzählung von Kreuzungspunkten sind Beispiele für *Listen*. Als Listenstrukturen darstellen lassen sich alle Arten von Wortlisten, Teilnehmerlisten, Verzeichnissen, Zugverläufen usw. Da Listen oder Aufzählungen von Dingen so häufig auftreten, gibt es für sie, zusätzlich zu der seither benutzten Notation für Termstrukturen eine vereinfachte Schreibweise, aus der der Aufzählungsaspekt leichter ersichtlich ist.

6.1 Notation

Listenstrukturen werden in PROLOG durch die eckigen Klammern `[`, `]` gekennzeichnet. Zwischen den Klammern werden die Elemente der Liste aufgezählt, durch Kommas voneinander getrennt. Listen lassen sich nach bewährtem Schema rekursiv beschreiben:

- Der Grenzfall, die atomare Liste ist die *leere Liste*. Die leere Liste enthält sinnigerweise keine Elemente und wird folgendermaßen dargestellt: `[ ]`
- *Nichtleere Listen* bestehen aus zwei „Teilen“:
  - dem *ersten Listenelement*, das ein beliebiger Term sein kann
  - und der *Restliste*, die wiederum dieser Listendefinition Genüge tun muß

Beispiel 6.1 *Beispiele für Listen*

```
[gitte,fido,venus]
[X]
[gitte,Planet]
[sonne,[erde,[mond],venus,saturn]]
[[ ],impliziert(mensch(X),sterblich(X))]
[menschen([gitte,lore,gert]),hunde([fido,bello])]
```

Listen, als Spezialfall von Termen, können beliebig ineinander und in andere Termen „verschachtelt“ sein.

6.2 Der Listenkonstruktor

Bei unserer selbsterstellten „Listennotation“ im Beispiel „Wegsuche“ war klar, wie man aus der Darstellung den Anfang und den Rest eines Weges erhält. Wegbeschreibungen waren als zweistellige Relation `weg(Punkt,RestWeg)` zwischen einem „Anfangspunkt“ und dem Restweg kodiert. Der Zugriff auf die Argumente erfolgte schlichtweg über Unifikation. In der abkürzenden Schreibweise für Listen ist uns die dahinterstehende Termstruktur verborgen. Sie entspricht jedoch ebenfalls einer zweistelligen Relation zwischen einem ersten Listenelement und der Restliste. Um diese Beziehung handlich darzustellen, gibt es den *Listenkonstruktor* , dargestellt durch das Zeichen `|` . Er trennt aber nicht nur das erste Element einer Liste von deren Rest, sondern kann allgemein zwischen einer festen Anzahl von Anfangselementen einer Liste und der entsprechenden Restliste stehen.

Beispiel 6.2 *Verwendung des Listenkonstruktors*

```
[Erstes|Rest]
[a,b,c|[d,e]]
```

Der Listenkonstruktor wird vielfältig eingesetzt, um Listenstrukturen aufzubauen und, analog dazu, auseinanderzunehmen. Zuerst wollen wir uns den Zugriff auf Elemente einer Liste veranschaulichen. Man kann sich eine Listenstruktur als einen Zug vorstellen, der vor einem Prellbock steht. Das Wichtige an dieser Metapher ist, daß nur der Zugriff auf den vordersten Wagen des Zuges möglich ist. Um zum Beispiel den letzten Wagen des Zuges zu entfernen, müssen alle davor stehenden Wagen„angefaßt“ und vorübergehend entfernt werden. Man kann also nur von einer Seite her und dann nur auf eine feste Anzahl von Wagen zugreifen. Der Verwendung des Listenkonstruktors entspricht also das„Wagenvoranstellen“ bzw. „-abhängen“. Hierzu folgen nun einige Beispiele:

Beispiel 6.3 *Notationelle Varianten von Listen*

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

Beispiel 6.4 *Zugriff auf erstes Element und auf Restliste*

```
?- [a,b,c] = [Erstes|Restliste].
Erstes = a
Restliste = [b,c]
?- [a] = [X|Y].
X = a
Y = []
?- [[mond],venus,saturn] = [A|B].
A = [mond]
B = [venus,saturn]
?- [mond,[venus,saturn]] = [A|B].
A = mond
B = [[venus,saturn]]
```

Beispiel 6.5 *Zugriff auf die beiden ersten Elemente und die Restliste*

```
?- [a,b,c] = [Erstes,Zweites|Rest].
Erstes = a
Zweites = b
Rest = [c]
```

Beispiel 6.6 *Test auf Mindestlänge 3*

```
?- [a,b,c,d] = [_,_,_|_].
yes
?- [a,b] = [_,_,_|_].
no
?- [ ] = [_,_,_|_].
no
```

Tests und Zugriffe nach obigem Muster lassen sich auch als Prädikate festschreiben, zum Beispiel:

```
%   zweites(Liste,ZweitesElement)
%   setzt eine Liste und deren zweites Element
%   zueinander in Beziehung.

zweites([_,Zweites|_],Zweites).
```

6.3 Rekursive Listenverarbeitung

Durch den Einsatz von Rekursion wird die Verarbeitung von Listenstrukturen wesentlich flexibler. Das Schema für die Rekursion über Listenstrukturen entspricht genau deren Aufbau. Rekursive Prädikate zur Manipulation sollten also aus mindestens zwei Klauseln bestehen:

- Einer Klausel, die die *Abbruchbedingung* der Rekursion angibt. Bei Listen sind dies meistens Klauseln, die aussagen, was mit einer leeren Liste oder einer Liste von einer vorgegebenen Länge (zum Beispiel einer einelementigen Liste) geschehen soll.
- Einer Klausel, die den *Rekursionschritt* beschreibt. Meist besteht dieser Schritt aus einer Manipulation des ersten Listenelements und einem rekursiven Aufruf mit der Restliste als Argument.

Als erstes Beispiel für ein rekursives Prädikat zur Beschreibung und Verarbeitung von Listen wollen wir die Listeneigenschaft, wie eingangs beschrieben, kodieren.

```
%   ist_liste(Liste)
%
%   testet, ob ein Term eine Liste ist:
%   - leere Listen,
%   - mindestens einelementige Listen,
%     deren Rest die Listeneigenschaft besitzt.

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

Ein weiteres, elementares Prädikat ist die Beschreibung der Elementbeziehung zwischen einem Term und einer Liste. Die Idee, die hinter diesem Prädikat steckt, ist, jeweils das erste Element der Liste mit dem potentiellen Element zu vergleichen. Falls dieser Vergleich zu einem negativen Ergebnis führt, wird (rekursiv) versucht, die Elementbeziehung für die noch verbliebene Restliste nachzuweisen. Diese Definition der Elementbeziehung leistet mehr als vermutet: Ruft man das Prädikat `element` mit einer Variablen für das Element auf, lassen sich über Backtracking alle Elemente einer Liste aufzählen.Übrigens wird dieses Prädikat in der PROLOG-Literatur meistens unter seinem englischen Namen `member` geführt.

```
%   element(Element,Liste)
%
%   Dieses Prädikat testet, ob 'Element'
%   in 'Liste' ist:
%   - Entweder ist 'Element' mit dem
%     ersten Element der Liste unifizierbar,
%   - oder 'Element' ist in der Restliste der Liste

element(Element,[Element|_]).
element(Element,[_|Restliste]) :-
    element(Element,Restliste).
```

Eine sehr häufige Operation auf Listen ist das „Zusammenhängen“ zweier Listen in eine Liste. Dieses Prädikat soll `append` heißen, weil das englische Wort schön kurz ist und das Prädikat unter diesem Namen schon in die PROLOG-Geschichte eingegangen ist. `append` beschreibt die Beziehung zwischen drei Listen, zwischen

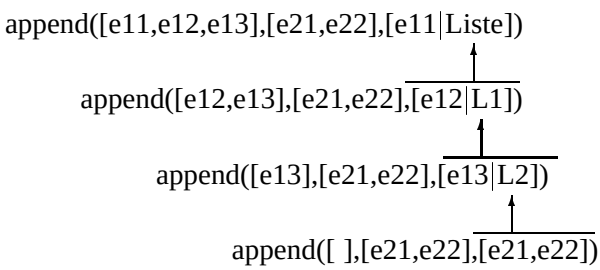
1. der Liste, deren Elemente „vorne“ stehen sollen
2. der Liste, deren Elemente „hinten“ stehen sollen
3. und der Gesamtliste

Anhand des Schemas für die Listenrekursion soll dieser Zusammenhang zuerst verbal beschreiben werden. Über welche der drei Listen soll die Rekursion „laufen“? Wir wählen die „vordere Liste“, da wie wir wissen, daß der Zugriff auf Listen nur von vorne her möglich ist.

- Die *Abbruchbedingung* ist schlichtweg der „triviale“ Fall, der Fall, in dem die vordere Liste leer ist. Dann stellt die hintere Liste die Gesamtliste dar.
- *Rekursion* : Falls die vordere Liste nicht leer ist, läßt sich folgender Zusammenhang formulieren:
  1. Das erste Element der vorderen Liste ist das erste Element der Gesamtliste.
  2. Der Rest der Gesamtliste ergibt sich durch Zusammenhängen des Restes der vorderen Liste mit der hinteren Liste.

```
%   append(VordereListe,HintereListe,GesamtListe)
%
%   Die GesamtListe ist eine Konkatenation von
%   VordereListe und HintereListe.

append([ ],Liste,Liste).
append([Element|Rest],Liste,[Element|Liste1]) :-
    append(Rest,Liste,Liste1).
```

Die Arbeitsweise von `append` soll durch eine schematische Darstellung des Ablaufprotokolls verdeutlicht werden:



`Liste = [e12|[e13|[e21,e22]]]`

PROLOG vereinfacht dies zu: `[e12,e13,e21,e22]`

Es gibt verschiedene Sichtweisen auf das Prädikat `append`. Beim Entwurf des Prädikats gingen wir ursprünglich von der Anwendung aus, daß zwei gegebene Listen zusammengehängt werden sollen. Das Ergebnis soll dann, wie gerade gezeigt, mit der beim Aufruf uninstantiierten Variablen auf der dritten Argumentstellte unifiziert werden. Doch `append` kann noch mehr. Es kann verwendet werden, um zu prüfen, ob die Konkatenationsbeziehung zwischen drei gegebenen Listen besteht:

```
?- append([a,b,c],[d,e,f],[a,b,c,d,e,f]).
yes
?- append([a,b,c],[d,e],[a,b,c,d,e,f]).
no
```

Das Prädikat `append` kann auch auf die Frage antworten, welche Liste noch an eine gegebene Liste angehängt werden muß, um eine bestimmte Gesamtliste zu ergeben.

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

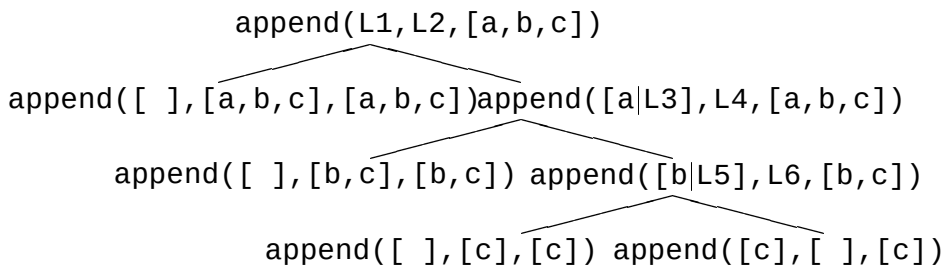
Das gleiche ist für eine noch unbekannte vordere Liste möglich:

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

Gleich mehrere Antworten (durch „von außen“ erzwungenes Backtracking) liefert die Frage nach den möglichen Zerlegungen einer Liste in eine vordere und eine hintere Liste:

```
?- append(VordereListe,HintereListe,[a,b,c]).
VordereListe = [ ]
HintereListe = [a,b,c];
VordereListe = [a]
HintereListe = [b,c];
VordereListe = [a,b]
HintereListe = [c];
VordereListe = [a,b,c]
HintereListe = [ ];
no
```

Die vier verschiedenen Lösungen kommen dadurch zustande, daß die vordere Liste leer sein kann oder daß die vordere Liste das erste Element der Gesamtliste enthält; die Restliste der vorderen Liste kann dann entweder leer sein oder das erste Element der Restliste der Gesamtliste enthalten usw. Nimmt man die Instantiierungen der Abbruchbedingung von `append` auf verschiedenen Rekursionsstufen, erhält man die Blätter eines „Suchbaumes“ für diesen Prädikatsaufruf. Die rekursiven Aufrufe stellen die inneren Knoten des Baumes dar:



6.4 Übungen

Übung 6.1 Beschreiben Sie durch ein nicht-rekursives Listenprädikat, wie das Auswechseln des vierten Elements einer mindestens vierelementigen Liste geschieht.

Übung 6.2 Schreiben Sie ein rekursives Prädikat, das aus einer Liste ein Element löscht. Das Prädikat sollte folgendermaßen aufgerufen werden können:

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

Experimentieren Sie mit `delete`, indem Sie versuchen, beim Aufruf andere Argumentpositionen mit Variablen zu belegen.

Übung 6.3 Schreiben Sie ein rekursives Prädikat, das es erlaubt, in einer Liste ein bestimmtes Element durch ein anderes zu ersetzen. Der Aufruf des Prädikats müßte so aussehen können:

```
replace(ZuErsetzendesElement,
        ErsatzElement,
        AlteListe,
        NeueListe).
```

Testen Sie das Prädikat zum Beispiel mit:

```
?- replace(hajo,katja,[otto,emil,hajo,mario,egon],X).
X = [otto,emil,katja,mario,egon]
```

Übung 6.4 Gegeben sei eine Datenbasis mit folgenden Personenbeschreibungen[1]:

```
person([1956,weiblich,verheiratet,stuttgart1]).
person([1964,weiblich,ledig,stuttgart70]).
person([1905,maennlich,verwitwet,stuttgart30]).
person([1980,weiblich,ledig,stuttgart40]).
person([1940,maennlich,geschieden,stuttgart1]).
person([1932,weiblich,verheiratet,waiblingen]).
person([1949,weiblich,geschieden,stetten]).
person([1910,maennlich,ledig,fellbach]).
person([1968,weiblich,ledig,stuttgart80]).
person([1956,maennlich,verheiratet,sindelfingen]).
```

Entwerfen Sie ein Prädikat `fahndung`, das zu bestimmten Merkmalen die zugehörigen Personenbeschreibungen nacheinander mittels Backtracking mit `;` ausgibt, zum Beispiel:

```
?- fahndung(weiblich,Beschreibung).
Beschreibung = [1956,weiblich,verheiratet,stuttgart1]
```

Verwenden Sie als „Unterprädikat“ das Prädikat `element`.

[1] Wir haben die heile Welt inzwischen verlassen.

Übung 6.5 Es soll ein Prädikat entworfen werden, das die Reihenfolge der Elemente einer Liste umdreht. Es sollte sich folgendermaßen verhalten:

```
?- reverse(Liste,[]).
Liste = []
?- reverse([a],[a]).
yes
?- reverse([a,b,c,d,e],Liste).
Liste = [e,d,c,b,a]
```

Übung 6.6 Für die Analyse eines Korpus, der aus einer Sammlung von Grabinschriften[2] besteht, wird eine Routine benötigt, die die Vorkommen von definiten und indefiniten Artikeln auflistet. Die Repräsentation für die Inschriften sei durch folgende Beispiele vorgegeben:

```
inschrift([[in,den,herzen],
           [der,kinder],
           [bleibt,fuer,ewig],
           [bewahrt,was,in],
           [treuer,liebe],
           [gewirkt]]).
inschrift([[die,trauernden,kinder],
           [beweinen,in,der,treuen,mutter],
           [ein,herrliches,gemueth],
           [einen,kraeftigen,geist],
           [eine,seltene,freimuetigkeit],
           [eine,treffliche,beraterin]]).
```

[2]Besonderer Dank gelte Herrn M.A. Dipl.-Bibl. U. Dickenberger dafür, daß wir in seine aufschlußreiche Magisterschrift Einsicht nehmen durften

Kapitel 7

Programmiertechnik

In den vorigen Kapiteln wurden schon häufiger verbale Beschreibungen und Programmkode zueinander in Bezug gesetzt. In diesem Kapitel sollen Antworten gegeben werden auf die Frage: Wie entsteht aus einer Idee ein Programm? Dies wird zuerst an einem Beispiel illustriert, darauf folgt eine Liste von Tips zur Vorgehensweise bei der Programmerstellung und zur Vermeidung gängiger Fehler.
Programmierer reden häufig davon, daß sie Programme, oder besser Programm-Systeme, „bauen“. Sind Programmierer eher mit Architekten oder mit Maschinenbauern zu vergleichen? Auf diese Frage möchten wir keine Antwort geben, sondern nur darauf hinweisen, daß die Funktionalität von Programmen eine größere Rolle spielt als die Ästhetik des Programmkodes. Obwohl es Leute gibt, die behaupten, daß gerade PROLOG-Programme auch ästhetisch, weil nämlich schön logisch, sein können.

7.1 Ein Beispiel

Die allgemeine Vorgehensweise beim Programmentwurf soll an einem Beispiel verdeutlicht werden. Nehmen wir an, wir hätten von einer Billig-Möbelfirma einen Bausatz für ein Bücherregal erstanden. Vor uns liege ein Karton mit den Einzelteilen. Da keine Aufbauanleitung mitgeliefert wurde und wir zum ersten Mal ein Regal zusammenbauen, wird es nötig sein, daß wir uns einige Gedanken machen.

7.1.1 Konzeptioneller Entwurf

Verfeinerungstufe 0

Beginnen wir mit einer Inventur der vor uns liegenden Einzelteile. Folgende Teile sollten sinnigerweise alle im fertigen Regal verwendet worden sein:

- 2 Seitenteile
- 12 Metallstifte
- 3 Regalbretter
- 1 Stützkreuz
- 4 Schrauben

Mathematisch gesprochen, läßt sich sagen, daß der Regalbau offenbar eine *Abbildung* von den Einzelteilen auf ein Regal ist:

$$\left.\begin{array}{l}\text{2 Seitenteile}\\ \text{12 Metallstifte}\\ \text{3 Regalbretter}\\ \text{1 Stützkreuz}\\ \text{4 Schrauben}\end{array}\right\} \overset{Regalbau}{\Rightarrow} \text{Regal}$$

Verfeinerungstufe 1

Die Aussage, daß ein Regal aus seinen Einzelteilen zusammengebaut wird, besagt noch nichts über die einzelnen Tätigkeiten, die alle zusammen den Regalbau konstituieren. Wie läßt sich feststellen, welche Tätigkeiten erforderlich sind? Da uns nichts anderes zur Verfügung steht, als die Sammlung der Einzelteile, bleibt nur der Versuch, aus der *Struktur* der Einzelteile Hinweise auf deren Verwendung zu erhalten. Wiederholen wir also nocheinmal die Inventarliste, aber diesmal um die nötigen Details erweitert:

- 2 Seitenteile. Jedes Seitenteil besitzt
  - je 3 große Löcher an den Innenseiten seiner beiden Leisten
  - 2 kleine Löcher an der Rückseite der hinteren Leiste.
- 12 Stifte
- 4 Regalbretter. An jedem der 4 Ecken jedes Brettes ist jeweils eine kurze Rille eingefräßt
- 1 Stützkreuz mit 4 Löchern.
- 4 Schrauben.

Der Regalbau wird im wesentlich darin bestehen, *Verbindungen* zwischen den Einzelteilen zu schaffen. Wo sind Verbindungen möglich?

- Die *Stifte* passen größen- und anzahlmäßig in die großen Löcher der Seitenteile und in die Rillen der Bretter
- Die *Schrauben* passen in die kleinen Löcher der Seitenteile und in die Löcher des Stützkreuzes

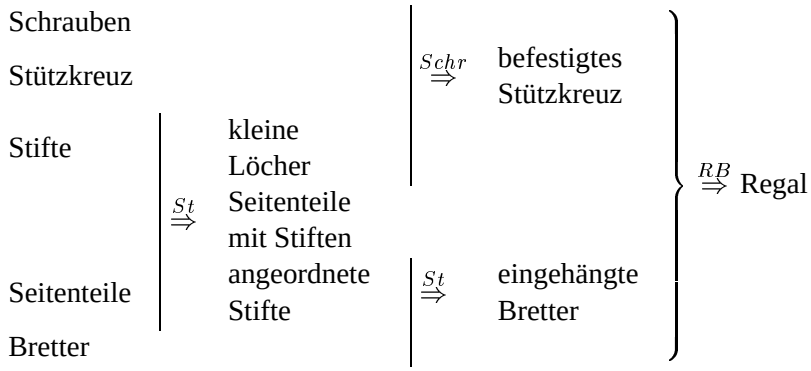
Wir erahnen den Ablauf der einzelnen Tätigkeiten für den Regalbau. Als erstes werden die Stifte in die großen Löcher der Seitenteile gesteckt, dann die Bretter auf die Stifte aufgelegt. Als letztes wird zur Stabilisierung des Regals das Stützkreuz mittels der Schrauben an die kleinen Löcher der Seitenteile befestigt. Damit kann auch die Abbildung *Regalbau* detaillierter beschrieben werden:

$$\left.\begin{array}{l}\text{Stifte}\\ \text{Seitenteile}\end{array}\right\} \overset{ES}{\Rightarrow} \text{Seitenteile mit Stiften} \left.\begin{array}{l}\\ \end{array}\right\} \overset{EL}{\Rightarrow} \text{Seitenteile mit Stiften und Brettern} \left.\begin{array}{l}\\ \\ \\ \\ \\ \\ \end{array}\right\} \overset{FS}{\Rightarrow} \text{Regal}$$

$$\text{Bretter}$$
$$\text{Stützkreuz}$$
$$\text{Schrauben}$$

Die Tätigkeit des Regalbauens besteht also aus den drei „Untertätigkeiten“: Einstecken ES, Einlegen EL und Festschrauben FS. Die mathematische Darstellung der Abbildungen gibt zwar den tatsächlichen Ablauf des Entstehens eines Regals wieder, sie deckt sich aber nicht ganz mit der vorhergehenden verbalen Beschreibung: Für das Auflegen der Bretter interessieren nicht die gesamten Seitenteile, sondern nur die *angeordneten Stifte* . Für das Befestigen des Stützkreuzes sind die Bretter und die Stifte als solche unwichtig, es ist nur nötig, auf die kleinen Löcher der Seitenteile „Zugriff“ zu haben. Es reicht

also, zumindest rein logisch gesehen, der Tätigkeit des Bretter-Einlegens nur den Teilaspekt„Angeordnete Stifte“ des in der Entstehung begriffenen Regals zur Verfügung zustellen, und der Tätigkeit des Stützkreuz-Festschraubens nur eine Liste der kleinen Löcher der Seitenteile. Oder noch einmal anders formuliert: Die Bezugspunkte, d.h. die *Schnittstellen* , zwischen den mit Stiften versehenen Seitenteilen und den Brettern sind die angeordneten Stifte, zwischen Seitenteilen und Stützkreuz sind es die kleinen Löcher der Seitenteile. Aufgrund dieser Überlegungen lassen sich die Abbildungen nocheinmal umformulieren.



Die Abbildung von den Einzelteilen auf ein Regal wird diesmal zerlegt in die drei Tätigkeiten: Das Stecken St von Stiften in Löchern oder von Stiften in Rillen und das Schrauben $Schr$. Das zusammengebaute Regal ist dann nur noch die „Aufzählung“ von befestigtem Stützkreuz, Seitenteilen mit Stiften und eingehängten Brettern, genauer gesagt: das an den Seitenteilen befestigte Stützkreuz, die Seitenteile mit Stiften und die in die Stifte eingehängten Bretter - damit sind auch die Verbindungen beschrieben. Diese Abbildung von Einzelteilen auf ein Regal über mehrere Stufen enthält etwas *Redundanz* . Die aus Seitenteilen und Stiften „zusammengebauten“ Seitenteile mit Stiften „enthalten“ schon die kleinen Löcher und die angeordneten Stifte, trotzdem werden diese gesondert erwähnt. Diese Redundanz nach der ersten Abbildung St erspart jedoch später, bei den Abbildungen $Schr$ und St das„Hantieren“ mit den „gesamten“ Seitenteilen.
Diese Betrachtungen haben hoffentlich das Thema Regalbau präzise genug abgehandelt, um als Grundlage für das nächste Teilkapitel dienen zu können.

7.1.2 Programmierung

Da wir uns über die Stupidität des Regalbauens ärgern, wollen wir unsern Beitrag zur Automatisierung des täglichen Lebens liefern und das Steuerprogramm für einen Roboter entwerfen, der Regale zusammenbaut. Dazu wird es nötig sein, die mathematischen Darstellung der Abbildungen vom vorherigen Abschnitt in ein PROLOG-Programm zu überführen.

Die Verbindungen

Wie lassen sich die Verbindungen innerhalb des Regals darstellen? Lassen wir unserer Intuition ein wenig Auslauf: Verbindungen zwischen zwei Teilobjekten des Regals konstituieren sich aus Löchern oder Rillen und Stiften oder Schrauben, die man in diese Aussparungen steckt bzw. schraubt. Als Analogie dazu gibt es auf der PROLOG-Ebene einerseits die Variablen, die wir als Namen von „Kästen“ kennengelernt haben, und andererseits nichtvariable Strukturen, die ebensolche„Kästen“ füllen. Für Löcher und Rillen bietet sich also eine Darstellung als Variablen an, für Stifte und Schrauben eine Repräsentation in Form nichtvariabler Strukturen. Verbindungen schaffen heißt dann, Variablen zu instantiieren. Die Tatsache, daß zwei Objekte miteinander verbunden sind, wird dadurch repräsentiert, daß sich die

PROLOG-Darstellungen dieser Objekte Variableninstantiierungen teilen. Zum Beispiel muß die Variable, die für die Eckrille eines Brettes steht, mit derselben Struktur instantiiert sein wie die Variable, die das entsprechende Loch am Seitenteil repräsentiert, auf das dieses Eck zeigt.
Das zweistellige Prädikat `stecken`, das einen Stift und ein Loch zueinander in Bezug setzt und das dreistellige Prädikat `verschrauben`, das eine Schraube, ein Loch des Seitenteils und ein Loch des Stützkreuzes in Relation setzt, lauten dann so:

```
verschrauben(X,X,X).
stecken(X,X).
```

Die Objekte

Der erste Schritt zur Darstellung der verschiedenen Objekte in PROLOG ist deren Unterteilung in atomare und komplexe Objekte, entsprechend der Unterscheidung zwischen atomaren und komplexen Strukturen. Als atomar wollen wir diejenigen Objekte ansehen, die keine „Unterobjekte“ besitzen, also nicht weiter strukturiert sind. Alles andere sind komplexe Objekte, deren Struktur noch genauer beschrieben werden muß.

- *atomare Objekte* : Stifte und Schrauben
- *komplexe Objekte* : Stützkreuz, Bretter, Seitenteile

Die Repräsentation für die atomaren Objekte ist einfach: wir tragen zwei Fakten `stifte` und `schrauben` in die Wissensbasis ein. Jeder der zwölf Stifte und jede der vier Schrauben erhält zur Kennzeichnung ein PROLOG-Atom als Namen.

```
stifte([st1,st2,st3,st4,st5,st6,
        st7,st8,st9,st10,st11,st12]).

schrauben([s1,s2,s3,s4]).
```

Wenn die atomaren Objekte als Fakten in der Datenbasis untergebracht werden, müssen konsequenterweise die komplexen Objekte durch Regeln dargestellt werden. Nehmen wir an, jedes komplexe Objekt „weiß“, welches seine „direkten“ Teilobjekte sind. Regale bestehen gemäß der zuletzt erwähnten mathematischen Abbildung aus zwei Seitenteilen, Brettern und einem Stützkreuz. Damit läßt sich das Grundgerüst für die PROLOG-Darstellung festlegen:

```
regal(Regal):-
   seitenteile(Seitenteile),
   bretter(Bretter),
   stuetzkreuz(Stuetzkreuz),
   Regal = r(Seitenteile,Bretter,Stuetzkreuz).
```

Oben wurde jedoch erwähnt, daß sich Seitenteile und Bretter die „angeordneten Stifte“ und daß sich Seitenteile und Stützkreuz die kleinen Löcher des Stützkreuzes teilen. Der Rumpf des Prädikats `regal` muß umgeschrieben werden, damit dieser Informationstransport über Variablen ermöglicht wird:

```
regal(Regal):-
   seitenteile(KleineLoecher,
               AngeordneteStifte,
               Seitenteile),
   bretter(AngeordneteStifte,Bretter),
   stuetzkreuz(KleineLoecher,Stuetzkreuz),
   Regal = r(Seitenteile,Bretter,Stuetzkreuz).
```

Das Prädikat `regal` enthält keine Hinweise, wie mit Stiften versehene Seitenteile oder Seitenteile überhaupt aussehen. Dies obliegt dem Prädikat `seitenteile` selbst: Man nehme die Stifte, stecke davon so viele wie nötig in das nächstliegende Seitenteil und den Rest in das übrige Seitenteil. Wenn man die richtige Zahl von Stiften der richtigen Zahl von großen Löchern zugeordnet hat, kann man mittels `stecken` die Verbindung einer Liste von Stiften mit einer Liste von Löchern auf ein Mal erzielen, dann lassen sich auch komplexere Strukturen miteinander unifizieren. Die Anordnung der Stifte wird festgehalten in vier Listen, d.h. für jedes Seitenteil zwei Verzeichnisse, welche Stifte entlang der vorderen und welche entlang der hinteren Lochreihe gesteckt wurden. Außerdem werden wir beim Betasten der Seitenteile gleich noch die kleinen Löcher jedes Seitenteils lokalisieren. Ein Seitenteil definiert sich dann zum Beispiel als ein Term `stl(AngeordneteStifte,KleineLoecher)`.

```
stifte_stecken([Stift1,Stift2,Stift3|RestStifte],
               RestStifte,
               [Stift1,Stift2,Stift3],
               [Loch11,Loch12,Loch13|RestLoecher],
               RestLoecher):-
   stecken([Stift1,Stift2,Stift3],
           [Loch1,Loch2,Loch3]).

seitenteile(Loecher,AngeordneteStifte,Seitenteile):-
   stifte(Stifte),
   seitenteil(Stifte,
              Reststifte,
              KleineLoecher1,
              AngeordneteStifte1,
              Seitenteil1),
   seitenteil(Reststifte,
              [],
              KleineLoecher2,
              AngeordneteStifte2,
              Seitenteil2),
   Loecher =  [KleineLoecher1,KleineLoecher2],
   AngeordneteStifte =
              [AngeordneteStifte1,AngeordneteStifte2],
   Seitenteile = [Seitenteil1,Seitenteil2].

seitenteil(Stifte,
           Reststifte,
           KleineLoecher,
           AngeordneteStifte,
           Seitenteil):-
   stifte_stecken(Stifte,
                  Reststifte1,
                  StifteVorne,
                  GrosseLoecher,
                  RestLoecher1),
   stifte_stecken(Reststifte1,
                  Reststifte,
                  StifteHinten,
```

```
                RestLoecher1,
                []),
   AngeordneteStifte = [StifteVorne,StifteHinten],
   Seitenteil =
      stl(AngeordneteStifte,KleineLoecher).
```

Ein Brett wird durch den vierstelligen Term `b(R1,R2,R3,R4)` dargestellt, wobei die vier Argumente die vier Rillen bedeuten. Das Einhängen eines Brettes wird dadurch nachgespielt, daß von jeder der vier Listen räumlich angeordneter Stifte jeweils das erste, d.h. das „oberste“ Element weggenommen wird und diese vier Stifte in die Rillen „gesteckt“ werden. Mit den verbliebenen Stiften werden die restlichen Bretter „eingehängt“. Diese Listenrekursion erlaubt uns, maximal soviele Bretter einzuhängen, wie Stifte vorhanden sind.

```
% Ende: keine Stifte mehr vorhanden
bretter([[[],[]],[[],[]]],[]).


% Rekursion:
% Nimm von allen 4 Listen das erste Element
bretter([[StifteHintenLinks,
          StifteVorneLinks],
         [StifteHintenRechts,
          StifteVorneRechts]],
        [Brett|Bretter]):-
   brett(StifteHintenLinks,
         ReststifteHL,
         StifteVorneLinks,
         ReststifteVL,
         StifteHintenRechts,
         ReststifteHR,
         StifteVorneRechts,
         ReststifteVR,
         Brett),
   bretter([[ReststifteHL,
             ReststifteVL],
            [ReststifteHR,
             ReststifteVR]],
           Bretter).
brett([Stift1|StifteHL],
      StifteHL,
      [Stift2|StifteVL],
      StifteVL,
      [Stift3|StifteHR],
      StifteHR,
      [Stift4|StifteVR],
      StifteVR,
      b(Rille1,Rille2,Rille3,Rille4)):-
   stecken([Stift1,Stift2,Stift3,Stift4],
           [Rille1,Rille2,Rille3,Rille4]).
```

Zum Schluß verbindet das Stützkreuz mittels der Schrauben die kleinen Löcher der Seitenteile mit seinen eigenen kleinen Löchern.

```
stuetzkreuz(SeitenteilLoecher,
            sk(StuetzkreuzLoecher)):-
    schrauben(Schrauben)
    SeitenteilLoecher = [[Loch1,Loch2],[Loch3,Loch4]],
    FlacheListe = [Loch1|[Loch2|[Loch3,Loch4]]],
    verschrauben(Schrauben,
                 FlacheListe,
                 StuetzkreuzLoecher).
```

Die Anfrage an diese Wissensbasis, wie denn nun ein Regal aussehe, führt zu folgender Antwort (hier wegen der Leserlichkeit schön formatiert):

```
    regal(Regal).
    Regal = r([stl([[st1,st2,st3],
                    [st4,st5,st6]],
                   [s1,s2]),
               stl([[st7,st8,st9],
                    [st10,st11,st12]],
                   [s3,s4])],
              [b(st1,st4,st7,st10),
               b(st2,st5,st8,st11),
               b(st3,st6,st9,st12)],
              sk([s1,s2,s3,s4]))
```

7.2 Programmentwurf-Richtlinien

Es folgt nun eine Art Checkliste, die eine Hilfestellung beim Programmentwurf geben soll. Die dabei anfallenden Arbeiten lassen sich in folgende Aufgabengebiete gruppieren, die nicht unbedingt nacheinander sondern untereinander abwechselnd erfüllt werden müßen:

- Konzeptioneller Entwurf
- Realisierung/Programmierung
- Testen
- Fehlersuche und -behebung
- Dokumentation

7.2.1 Konzeptioneller Entwurf

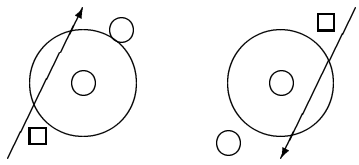
Der eigentlichen Kodier- und Testarbeit an der Maschine sollte unbedingt ein *konzeptioneller Entwurf* vorausgehen. Das Ziel des konzeptionellen Entwurfs ist es, sich über das zu behandelnde Problem und dessen Realisierungsmöglichkeiten klar zu werden. Hilfreich sind hierfür sowohl rein verbale als auch graphische und formale Beschreibungen der Objekte und Operationen auf den Objekten, die erfolgen sollen. Über verschiedene *Verfeinerungsstufen* der Betrachtungsweise hinweg sollten zumindest auf folgende Fragen Antworten gefunden werden:

1. Wie sieht der Ausgangszustand aus?
   (a) Welche Objekte spielen eine Rolle?
   (b) Wie stehen diese Objekte zueinander in Bezug?
2. Wie sieht der Zielzustand aus?
   (a) Welche Objekte sollen erzeugt werden?
   (b) Wie stehen diese Objekte zueinander in Bezug?
   (c) Welche Arbeitsvorgänge werden für diese Inbezugsetzung benötigt?
   (d) Welche „Werkzeuge“ braucht man dazu?
   (e) Welche Bearbeitungsfolge bietet sich an? (Diese Frage spielt in PROLOG im allgemeinen keine sehr herausragende Rolle.)

Bei komplizierten Aufgabenstellungen sind Verfeinerungstufen in der Beschreibung nötig, um nicht„vor lauter Bäumen den Blick auf den Wald“ zu verlieren. Um weiter im Bild zu sprechen: Auf einer Betrachtungsebene mag es ausreichen, den Wald als Ganzes zu betrachten und die Bäume zu vernachläßigen, auf einer anderen wird es nötig sein, auf seine Bestandteile, die Bäume näher einzugehen. EinÜbergang von einer Verfeinerungstufe zur nächsten zeichnet sich zum Beispiel dadurch aus,

- daß Objekte detaillierter beschrieben werden
- daß nur gewisse Teilobjekte betrachtet werden
- daß nur gewisse Teiloperationen untersucht werden

Oft zeigt sich bei der detaillierten Beschreibung einer Problemstellung, daß das Problem einem früher geschriebenen und eventuell schon gelösten Problem strukturell ähnlich ist.



7.2.2 Realisierung

Die Umsetzung des konzeptionellen Entwurfs in ein Programm wird auch Realisierung genannt. Es ist hilfreich, die in jeder Verfeinerungstufe erzielten Beschreibungen einzeln anhand der beiden folgenden Fragen nach PROLOG zu übersetzen.

1. Wie sollen die Objekte repräsentiert werden?
2. Wie sollen die Beziehungen zwischen Objekten dargestellt werden?

Die Arbeit läßt sich noch weiter erleichtern, wenn man folgende Hinweise beachtet:

- Die Fallunterscheidungen für die einzelnen Klauseln von Prädikaten ergeben sich meist aus der Struktur der Objekte (z.B. atomare Objekte versus strukturierte Objekte).
- Die Verarbeitung strukturierter Objekte verläuft oft rekursiv über deren Struktur.
- Die Hierarchie der Abstraktionsebenen sollte sich in der Hierarchie der Prädikatsaufrufe widerspiegeln.
- Soweit sinnvoll und möglich, sollte „reines“ PROLOG verwendet werden. (Über den vollen Sprachumfang von PROLOG, der über „reines“ PROLOG hinausgeht, wird später geredet.)

7.2.3 Programmtest

Um nachzuweisen, daß das entwickelte Programm seine Aufgaben erfüllt, sollte auf verschiedene Arten getestet werden:

- Tests für den allgemeinen (typischen) Fall
- Tests für Randfälle, Ausnahmen und eventuell für falsche Eingaben durch den Benutzer

Tests können Fehler aufdecken, deren Behebung Änderungen im Programmkode erfordern, wenn nicht sogar Änderungen im konzeptionellen Entwurf (wenn beispielsweise Randfälle vergessen wurden). Erfolgreich verlaufene Tests sind jedoch noch keine Garantie, daß das Programm immer korrekt arbeitet.

7.2.4 Fehlerbehebung

Die beste Methode, Fehler zu „behandeln“, ist, durch achtsames Vorgehen die Zahl der Fehler einzuschränken. Die Beachtung folgender Regeln dient der *Fehlervorbeugung* :

- Erstellung eines ordentlichen konzeptionellen Entwurfs
- übersichtliche Darstellung der Prädikate:
  - Kommentare!
  - Faustregel:
    Nicht mehr als 5 bis 7 Klauseln pro Prädikatsdefinition, sonst ein *Unterprädikat* einführen.
  - Da PROLOG die Datenbasis von oben nach unten durchsucht, empfiehlt sich folgende Anordnung der Klauseln in einer Prädikatsdefinition:
    1. Abbruchbedingung(en)
    2. Sonstige Klauseln
    3. Allgemeinster Fall

Wenn sich trotz allem Fehler eingeschlichen haben, sollte man für die *Fehlersuche* folgende Punkte im Auge haben:

- Wird das Programm vom PROLOG-Interpreter nicht kommentarlos eingelesen, spricht man von *Syntaxfehlern* . Die häufigsten Syntaxfehler sind:
  - Der Punkt und der Zeilenvorschub fehlen am Ende einer Klausel.
  - Die Zahl der schließenden und die Zahl der öffnenden Klammern entsprechen sich nicht. Dies kann besonders bei Kommentarklammern $/* \ldots */$ heimtückische Auswirkungen haben.
  - Ein Leerzeichen zwischen Prädikatnamen und erster öffnender Klammer mißfällt vielen PROLOG-Interpretern.
- Wenn das Programm nicht läuft (d.h. nur stur mit `no` antwortet) oder nicht das tut, was es tun sollte, sollten folgende Punkte überprüft werden:
  - Sind alle Prädikats- und Variablennamen, die für das Gleiche stehen auch gleich geschrieben?
  - Ist die Argumentzahl und die Reihenfolge der Argumente eines bestimmten Prädikats bei allen seinen Erwähnungen eingehalten?
  - Passen Listen und Terme, die miteinander unifiziert werden sollen, auch wirklich aufeinander?

  - Stimmt die Zuordung von öffnenden und schließenden Klammern?

Es gibt eine Reihe *eingebauter Prädikate* , die die Fehlersuche unterstützen, wie zum Beispiel:

- `trace`
- `spy`

Die Arbeitsweise dieser beiden und weiterer *debugging*-Prädikate wurde bereits in Kapitel 4 angedeutet und ist in den Handbüchern und der einschlägigen Literatur beschrieben.

7.2.5 Dokumentation

Alle verschiedenen Stufen der Programmentwicklung sollten ständig dokumentiert werden. Der konzeptionelle Entwurf zählt dabei schon als der erste Teil der Dokumentation. Die Vorteile ausführlicher Dokumentation liegen auf der Hand:

- ein guter *konzeptioneller Entwurf* erspart Irrwege in der Programmierung
- eine Art *Tagebuch* über die Entwicklung bewahrt davor, denselben Irrweg zweimal zu gehen
- bei einem gut dokumentiertem Programm besteht die Chance, daß andere Leute (und man selbst), das Programm jederzeit verstehen und erweitern können

Bereits die Verwendung klingender Namen für Prädikate und deren Argumente trägt zur Dokumentation bei. Außerdem empfiehlt es sich, vor jeder (wichtigen) Prädikatsdefinition einen Kommentarkasten folgender Art einzufügen und entsprechend auszufüllen:

```
%   Prädikat
%
%   Argumente des Prädikats und deren Struktur
%
%   Beschreibung der Arbeitsweise des Prädikats
%
%   Beispiele für Aufrufe und zugehörige Antworten
%
%   Besondere Hinweise, Warnungen, Ausnahmen
```

7.3 Zusammenfassung

Der „Rohstoff“, mit dem Programmierer arbeiten, ist Information. Das Wesen von Information ist ihre Struktur. Genauso wie ein Werkstück von einem Bearbeitungszustand in den nächsten überführt werden kann, kann Information von einer Art von Strukturiertheit in eine andere überführt werden.
Programmierung in PROLOG hat viel mit *Wissensrepräsentation* , der Logik-Darstellung von Aspekten alltäglichen, meist wenig formalisierten Wissens, zu tun. Zum Beispiel ist es im Falle des Regalbaus keineswegs klar, warum gerade diese Repräsentation für Regale und ihre Bestandteile gewählt wurde und nicht eine andere. Der Zweck heiligt auch hier die Mittel. Unsere PROLOG-Darstellung für ein Regal beleuchtet nur gewisse Aspekte eines Regals, die räumlichen Eigenschaften sind zum Beispiel etwas zu kurz gekommen.
Insgesamt wurde eine Leitlinie für die Programmentwicklung vorgestellt. Die Kreativität, die Ideen hervorbringt, funktioniert wohl etwas sprunghafter und unsystematischer.

7.4 Übungen

Übung 7.1 Folgendes Prädikat enthält einige Fehler. Beheben Sie sie!

```
Permute(Liste,[Erstes|Element|Rest]):-
    append(Liste1,[ErstesElement|Liste2],Liste),
    append(Liste1,Liste2,Lsite3),
permute(Liste3,Rest,Rest1).
permute([],[]). /* Abbruchbedingung

append([],Liste,Liste)
append([ErstesElement|Rest],
       Liste,
       [ErstesElement|Liste1]):-
    append(Rest,Liste).
```

Übung 7.2 Fertigen Sie einen konzeptionellen Entwurf für folgende Idee und realisieren Sie ihn als Programm. Dokumentation und Tests nicht vergessen!

Idee: „Der große Zauberer und seine Gehilfen“

Der große Zauberer verwandelt Dinge:

- einen gelben Zauberstab in drei rote Tücher
- schwarze Hasen mit weißen Ohren und blauen Augen in weiße Hasen mit weißen Ohren und roten Augen
- braune Hasen werden zuerst in blaue Hüte, und dann anschließend in grüne Zauberstäbe mit schwarzem Griff und weißer Spitze verwandelt
- gelbe Mützen werden manchmal in schwarze Hüte, manchmal in Geldstücke verwandelt.

Außerdem ist zu bemerken, daß der große Zauberer eigentlich nichts selbst macht, sondern die Arbeit an seine, auf bestimmte Dinge spezialisierten Gehilfen abgibt.

Übung 7.3 Entwerfen Sie ein Programm, das folgenden Wortsalat sinnvoll zu einem Satz anordnen kann:

```
katze, maus, die, einer, nach, sucht
```

Teil II

Linguistische Anwendungen

Kapitel 8

Syntaxanalyse

In diesem Kapitel wollen wir ein PROLOG-Programm entwickeln, das für einfache englische Sätze überprüft, ob diese grammatikalisch korrekt sind. Dazu müssen wir zunächst klären, was unter *grammatikalisch korrekt* zu verstehen ist.

8.1 Kontextfreie Grammatiken

Um uns die Arbeit für den Anfang so leicht wie möglich zu machen, nehmen wir an, daß sich das zu behandelnde Fragment der englischen Sprache durch eine kontextfreie Grammatik beschreiben läßt.
Eine Grammatik beschreibt formal exakt, wie sich jede Konstituente der Sprache aus anderen Konstituenten zusammensetzt. Zum Beispiel besteht der einfache Hauptsatz

John eats the apple

aus der Nominalphrase John und der Verbalphrase eats the apple. Dies kann durch die Regel

S $\rightarrow$NP VP

ausgedrückt werden. Eine solche Regel heißt auch *Produktion*. Solche Produktionen sind ein Bestandteil einer *kontextfreien Grammatik*.
Insgesamt besteht ein kontextfreie Grammatik aus vier Komponenten:

1. Einer Menge von *Terminalsymbolen*
2. Einer Menge von *Nichtterminalsymbolen*
3. Einer Menge von *Produktionen*
4. Einem ausgezeichnetem Nichtterminalsymbol, dem *Startsymbol*

Terminalsymbole sind in unserem Fall die Wörter der englischen Sprache. Nichtterminalsymbole sind die Bezeichnungen für die Konstituenten. Das Startsymbol ist in der Regel S, da wir Sätze analysieren wollen.
Produktionen bestehen aus einem Nichtterminalsymbol, genannt *linke Seite*, einem Pfeil und einer Folge von Terminal- oder Nichtterminalsymbolen, der *rechten Seite*.
Die Bezeichnung „kontextfrei“ für diese Grammatiken deutet an, daß es nicht möglich ist, für die Anwendbarkeit von Produktionen Kontextbedingungen anzugeben. Dies wäre z.B. möglich, wenn man auf

der linken Seite von Produktionen mehr als nur ein einzelnes Nichtterminalsymbol angeben könnte. Grammatiken, die dies erlauben, heißen dann „kontextsensitiv“.
In der Linguistik ist es üblich, in den Produktionen keine Terminalsymbole zu verwenden. Stattdessen werden die Wörter in *lexikalische Kategorien* eingeteilt. Diese werden dann in den Produktionen verwendet. Man bezeichnet sie oft als *Präterminalsymbole*. Die Einteilung geschieht in einem *Lexikon*. Dabei kann ein Wort auch mehreren Kategorien zugeordnet werden.

Beispiel 8.1 Für unser (Mini-) Fragment des Englischen nehmen wir eine *Grammatik* an, die aus folgenden *Produktionen* besteht:

1. S $\rightarrow$NP VP
2. NP $\rightarrow$DET N
3. NP $\rightarrow$N
4. VP $\rightarrow$V NP
5. VP $\rightarrow$V

Das *Lexikon* sei gegeben durch folgende Wörter und ihre syntaktischen Kategorien:

mary	N	the	DET
john	N	loves	V
woman	N	eats	V
man	N	sings	V
apple	N		

8.1.1 Ableitung

Auf welche Weise genau wird nun durch eine Grammatik eine Sprache charakterisiert? Um dies exakt zu beschreiben, führen wir den Begriff der *Ableitung* ein.
Ein Satz ist genau dann in der von einer kontextfreien Grammatik *erzeugten Sprache* enthalten, wenn es eine Folge von Produktionen gibt, deren sukzessive Anwendung ausgehend vom Startsymbol gerade diesen Satz ergibt. Die Folge dieser Produktionsanwendungen heißt *Ableitung*.

Beispiel 8.2 Betrachten wir wieder den Satz John eats the apple. Eine mögliche Ableitung ist:

	(1)	(3)	(4)	(2)
S	$\Rightarrow$NP VP			
		$\Rightarrow$N VP		
			$\Rightarrow$N V NP	
				$\Rightarrow$N V DET N

Die Kette der Präterminalsymbole paßt auf den Satz. Damit ist John eats the apple gemäß unserer Grammatik ein korrekter Satz.

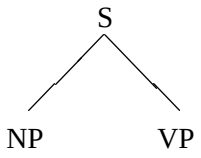
Die Angabe einer Ableitung ist ein Beweis, daß eine gegebene Eingabekette zu der durch die Grammatik erzeugten Sprache gehört.
Zu einem Satz der Sprache gibt es in der Regel mehr als eine Ableitung. Im obigen Beispiel können wir auch die Produktion (4) vor der Produktion (3) anwenden und erhalten die folgende Ableitung für den Satz John eats the apple:

(1) (4) (3) (2)
S ⇒NP VP
⇒NP V NP
⇒N V NP
⇒N V DET N

Und es gibt sogar noch eine weitere Ableitung!
Doch besteht ein relevanter Unterschied zwischen diesen verschiedenen Ableitungen? Eigentlich sind wir nur an der *Konstituentenstruktur* eines Satzes interessiert und nicht an den verschiedenen Ableitungen. Die Konstituentenstruktur eines Satzes gibt uns Aufschluß darüber, welche Teile des Satzes miteinander eine Einheit, eine *Konstituente*, bilden und wie einfache Konstituenten zu größeren zusammengefaßt werden. Lassen sich für einen Satz verschiedene Konstituentenstrukturen angeben, so kann man in der Regel auch verschiedene Bedeutungen in diesen Satz hineinlesen, d.h. er hat verschiedene *Lesarten*. Im obigen Fall handelt es sich aber nur um eine unterschiedliche Reihenfolge der Anwendung der Produktionen und nicht um einen Unterschied in der Konstituentenstruktur.

8.1.2 Syntaxbäume

Wir benötigen also eine geeignete Darstellung, die es uns erlaubt, von der Reihenfolge der Produktionsanwendungen zu abstrahieren und nur die Konstituentenstruktur eines Satzes darzustellen. Genau das ist die Aufgabe eines *Syntaxbaumes*.
Bäume sind Datenstrukturen, die dazu dienen, hierarchische Strukturen darzustellen. Die Konstituentenstruktur eines Satzes ist ein typisches Beispiel für eine hierarchische Struktur. Daher ist ein Baum ein geeignetes Beschreibungsmittel dafür.
Ein *Syntaxbaum* stellt graphisch dar, wie ein Satz ausgehend vom Startsymbol der Grammatik abgeleitet werden kann. Wenn für die Ersetzung des Nichtterminalsymboles S die Produktion S → NP VP angewandt wird, so erhalten wir einen Syntaxbaum mit der *Wurzel* S und den *Blättern* NP und VP:

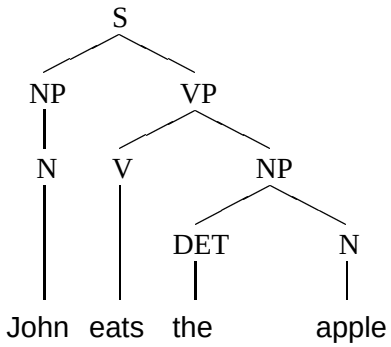


S, NP und VP sind *Markierungen* an *Knoten* des Baumes. Der Knoten mit der Markierung S heißt *innerer Knoten*, da er noch untergeordnete *Teilbäume* hat. Knoten ohne untergeordnete Teilbäume heißen *Blätter*.
Wenn an einem Blatt eines Baumes ein Nichtterminalsymbol steht, so kann hier ein weiterer Teilbaum für eine Regel mit passender linker Seite eingesetzt werden.
Für einen ganzen Satz hat jeder Syntaxbaum folgende Eigenschaften:

1. Die Wurzel des Syntaxbaumes ist mit dem Startsymbol der Grammatik markiert
2. Jedes Blatt des Syntaxbaumes ist mit einem Lexem markiert
3. Jeder innere Knoten des Syntaxbaumes ist mit einem Nichtterminalsymbol markiert
4. Wenn ein innerer Knoten mit dem Nichtterminalsymbol A markiert ist und die unmittelbarer Unterknoten mit den Symbolen B, C, ..., D markiert sind, so ist die zugehörige Produktion A → B C ... D.
5. Wenn ein innerer Knoten mit einem präterminalen Symbol markiert ist, so ist der unmittelbare Unterknoten ein Blatt mit einem Lexem dieser Kategorie

Zu jeder Ableitung gibt es genau einen Syntaxbaum. Alle Ableitungen, die dieselbe Konstituentenstruktur darstellen, ergeben denselben Syntaxbaum.

Beispiel 8.3 Der Syntaxbaum für den Satz John eats the apple ist:



Dieser Syntaxbaum stellt alle möglichen Ableitungen für diesen Satz auf einmal dar.

8.1.3 Ambiguität

Wir müssen vorsichtig sein, wenn wir von *der* Konstituentenstruktur eines Satzes gemäß einer Grammatik reden. Es ist zwar klar, daß es zu jedem Syntaxbaum nur genau einen Satz gibt, dessen Konstituentenstruktur er darstellt, aber es ist durchaus möglich, daß eine Grammatik für einen Satz mehrere Syntaxbäume zuläßt. In diesem Fall sprechen wir von *Ambiguität* oder *Mehrdeutigkeit* der Grammatik und auch des Satzes. In der Regel bedeuten verschiedene Syntaxbäume für einen Satz, daß es verschiedene Lesarten für diesen Satz gibt.

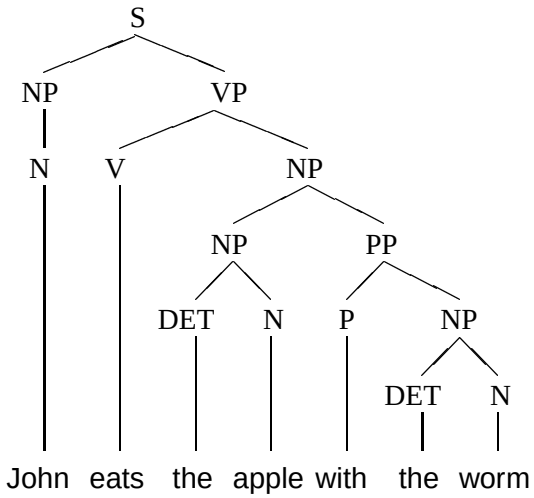
Beispiel 8.4 Wir wollen unser Fragment des Englischen um eine einfache Präpositionalphrasensyntax erweitern. Dazu fügen wir zu der Grammatik aus Beispiel 8.1 noch folgende Regeln hinzu:

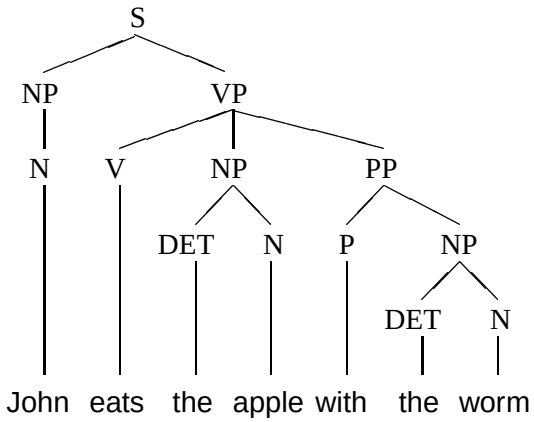
6. PP $\rightarrow$P NP
7. NP $\rightarrow$NP PP
8. VP $\rightarrow$V NP PP

Das Lexikon wird erweitert um folgende Wörter und ihre syntaktischen Kategorien:

worm	N	with	P
knife	N	in	P

Für den Satz John eats the apple with the worm gibt es zwei verschiedene Syntaxbäume. Der erste gibt die Lesart wieder, in der John einen wurmigen Apfel ißt; der zweite dagegen bedeutet, daß John mit einem Wurm einen Apfel ißt. Dies wäre die richtige Lesart für den Satz John eats the apple with a knife.





Das Erkennen der „richtigen“ Lesart für einen Satz ist in der Regel ein sehr schwieriges Problem, zu dessen Lösung Information auf ganz verschiedenen Ebenen relevant ist. Oft gibt es syntaktische Bedingungen, die gewisse Lesarten ausschließen, z.B. die Subkategorisierung von Verben. Manche Ambiguität läßt sich auf der semantischen Ebene beseitigen, z.B. durch Restriktionen für die Auswahl von Füllern für semantische Rollen. In unserem Beispiel ließe sich die zweite Lesart erst durch unser„Weltwissen“, daß nämlich ein Wurm kein Instrument zum Essen eines Apfels sein kann, ausschließen. In vielen Fällen besteht aber auch überhaupt keine Chance, für einen einzelnen Satz eine eindeutige Lesart festzulegen. Mit solchen Ambiguitäten müssen die weiteren Komponenten in einem natürlichsprachlichen System umgehen können.

8.2 Parsing

In diesem Abschnitt wollen wir ein PROLOG-Programm entwickeln, das für die Grammatik aus Beispiel 8.1 überprüft, ob ein gegebener Satz in der von der Grammatik erzeugten Sprache enthalten ist. Ein solches Programm nennt man einen *Parser*. Der Prozess der Analyse eine Eingabesatzes durch den Parser heißt *Parsing*.
Zur Darstellung des zu analysierenden Satzes wählen wir eine Liste, die als Elemente die einzelnen

Wörter des Satzes enthält. Die Wörter werden durch PROLOG-Atome dargestellt. Unser Parser soll also z.B. folgendes Verhalten zeigen:

```
?- s([john,eats,the,apple]).
yes

?- s([john,the,apple,eats]).
no
```

Der Parser soll also *beweisen*, ob ein gegebener Satz in der von der Grammatik erzeugten Sprache enthalten ist. Wie kann ein solcher Beweis aussehen? Nehmen wir dazu die Produktion S $\rightarrow$ NP VP und formulieren ihre Bedeutung:

> Eine Eingabekette stellt eine Konstituente S dar,
>
> *falls* sich diese Kette in zwei Teile aufspalten läßt
>
> *und* der erste Teil eine Konstituente NP
>
> *und* der zweite Teil eine Konstituente VP darstellt.

Analog dazu können wir jede Produktion der Grammatik in eine entsprechende Regel umformulieren. Die obige Formulierung der Regel hat den Vorteil, daß sie schon genau die Form einer Klausel hat. Wir können diese Regel unmittelbar in PROLOG aufschreiben:

```
s(S) :-
   append(NP,VP,S),
   np(NP),
   vp(VP).
```

Wir benutzen hier `append` dazu, die Eingabekette in zwei Teile zu zerlegen.
Wie funktioniert nun dieses Prädikat `s(S)`? Nehmen wir einmal an, die Prädikate `np(NP)` und `vp(VP)` würden bereits wie gewünscht funktionieren. Also ist z.B. `np(NP)` genau dann beweisbar, wenn `NP` eine Nominalphrase gemäß unserer Grammatik darstellt.
Betrachten wir jetzt folgende Anfrage:

```
?- s([john,eats,the,apple]).
```

Der Kopf der Klausel `s(S)` paßt auf diese Anfrage; daher muß PROLOG jetzt versuchen, den Rumpf der Klausel zu beweisen. Dadurch erhalten wir als nächstes Beweisziel:

```
append(NP,VP,[john,eats,the,apple])
```

PROLOG findet hierfür eine erste Lösung mit `NP = []` und `VP = [john,eats,the,apple]`. Daraus erhalten wir als nächstes Beweisziel:

```
np([])
```

Eine Nominalphrase kann aber nach unserer Grammatik nicht leer sein; der Beweisversuch für `np([])` scheitert, da wir ja die Korrektheit der Prädikate `np` und `vp` vorausgesetzt hatten. PROLOG muß hier also „backtracken“ und für `append(NP,VP,[john,eats,the,apple])` eine neue Lösung suchen. Dies führt zu `NP = [john]` und `VP = [eats,the,apple]` (warum?). Damit erhalten wir das folgende neue Beweisziel:

```
np([john])
```

Dieses Mal ist PROLOG erfolgreich und daher ergibt sich als nächstes Beweisziel:

```
vp([eats,the,apple])
```

Auch das klappt wunderbar, sodaß jetzt der gesamte Beweis für `s([john,eats,the,apple]` abgeschlossen ist und PROLOG erleichtert

```
yes
```

ausgeben kann.
Nun kann man natürlich sämtliche Produktionen der Grammatik genau nach dem Muster für S $\rightarrow$ NP VP in Klauseln übersetzen. Dies ergibt folgendes Programm:

```
s(S) :-
   append(NP,VP,S),
   np(NP),
   vp(VP).

np(NP) :-
   append(DET,N,NP),
   det(DET),
   n(N).
np(NP) :-
   N = NP,
   n(N).

vp(VP) :-
   append(V,NP,VP),
   v(V),
   np(NP).
vp(VP) :-
   V = VP,
   v(V).
```

Bei den Klauseln für die Produktionen, deren rechte Seite nur ein Nichtterminalsymbol enthält, kann natürlich die Zerlegung mittels `append` unterbleiben.
Die Darstellung des Lexikons in PROLOG für diesen Parser ist naheliegend. Da z.B. `v(V)` beweisbar sein soll, wenn die Liste `V` genau ein Wort enthält, das ein Verb ist, können wir alle Verben einfach durch Fakten der Form `v([eats])` in die Datenbasis eintragen.
Hier also unser vollständiges Lexikon[1]:

```
n([mary]).
n([john]).
n([woman]).
n([man]).
```

[1] Eigennamen werden hier als normale Nomen behandelt. Uns ist klar, daß dies nicht adäquat ist. In der Übung 8.3 erhalten Sie Gelegenheit, dies besser zu machen.

```
n([apple]).

det([the]).

v([loves]).
v([eats]).
v([sings]).
```

8.3 Übungen

Übung 8.1 Geben Sie alle Ableitungen für den Satz The man loves the woman an, die mit der Grammatik aus Beispiel 8.1 möglich sind.

Übung 8.2 Geben Sie alle Syntaxbäume für den Satz John saw the man in the park with the telescope an, die mit der Grammatik aus Beispiel 8.4 nach entsprechender Erweiterung des Lexikons möglich sind. Beschreiben Sie, welche Bedeutung des Satzes durch jeden Syntaxbaum wiedergegeben wird. Sind alle denkbaren Bedeutungen – sinnvoll oder nicht – dieses Satzes damit erfaßt? Wenn nicht, wie müßte die Grammatik erweitert werden?

Übung 8.3 Erweitern Sie die Grammatik aus Beispiel 8.1 so, daß Sätze der Form John eats apple oder The John eats the apple nicht mehr erzeugt werden. Erweitern Sie auch den Parser aus Abschnitt 8.2 entsprechend und testen Sie damit Ihre Grammatik.

Übung 8.4 Erweitern Sie die Grammatik aus Übung 8.3 um eine einfache Behandlung von Adjektiven. Es sollten z.B. die Sätze John loves the beautiful woman und Mary lives in a big white house möglich sein. Erweitern Sie auch den Parser und testen Sie Ihre Grammatik.

Übung 8.5 Schreiben Sie einen Parser für die Grammatik aus Beispiel 8.4. Was geht beim Parsing schief?

Kapitel 9

Definite Clause Grammars

Die syntaktische Analyse von Eingaben ist eine sehr häufig vorkommende Aufgabe bei der PROLOG-Programmierung. Deshalb bietet PROLOG eine einfache Möglichkeit, Grammatiken und Parser zu schreiben. Um einen Parser für eine kontextfreie Grammatik zu programmieren, reicht es aus, die Grammatikregeln in einer bestimmten Notation aufzuschreiben und diese Datei dann mit `consult` zu laden. In der PROLOG-Literatur heißt diese Notation *Definite Clause Grammar* oder kurz *DCG*.

Beispiel 9.1 Die Grammatik aus Beispiel 8.1 im DCG-Format:

```
s   --> np, vp.
np  --> det, n.
np  --> n.
vp  --> v, np.
vp  --> v.
```

Das Lexikon dazu erhält die Form:

```
n   --> [mary].
n   --> [john].
n   --> [woman].
n   --> [man].
n   --> [apple].
det --> [the].
v   --> [loves].
v   --> [eats].
v   --> [sings].
```

PROLOG benutzt die DCG aber nicht direkt zum Parsing. Stattdessen werden beim Laden der Datei mit `consult` die Produktionen der Grammatik in „normale“ Klauseln übersetzt. Dabei wird für jede DCG-Produktion genau eine Klausel erzeugt.

Nehmen wir an, wir hätten die Datei mit der DCG aus Beispiel 9.1 mittels `consult` erfolgreich geladen. Wie sieht nun die interne Darstellung der Produktionen aus? Dazu schauen wir uns mit `listing` einmal die Klausel für `s` und die Lexikoneinträge für die Verben an:

```
?- listing(s).
s(A,B) :-
   np(A,C),
   vp(C,B).
yes
?- listing(v).
v([loves|A],A).
v([eats|A],A).
v([sings|A],A).
yes
```

Diese Darstellung sieht völlig anders aus als unser Parser im vorigen Kapitel. Welche Idee steckt hinter dieser Kodierung?
Die Bedeutung der Klausel `s` läßt sich folgendermaßen umschreiben:

> `s(A,B)` gilt,
>
> *falls* die Liste `A` mit einer Konstituente vom Typ s beginnt
>
> *und* der Rest der Liste ohne diese Konstituente gerade `B` ist.

Jede Produktion der DCG wird also in eine Klausel übersetzt, die von der Liste an der ersten Argumentstelle am Anfang ein Stück „abknabbert“, das gerade eine mögliche Konstituente des jeweiligen Typs ist und den Rest der Liste auf dem zweiten Argument übrigläßt.
Das Lexikon muß demzufolge so kodiert werden, daß genau das Lexem vom Anfang der Liste entfernt wird:

```
?- n([john,loves,mary],X).
X = [loves,mary]
yes
```

Die Lexikoneinträge erhalten deshalb immer die Form `kategorie([lexem|Rest],Rest]`, so wie wir dies am Beispiel der Verbeinträge schon gesehen haben.
Zum besseren Verständnis wollen wir uns die Abarbeitung des einfachen Satzes John loves Mary durch den DCG-Parser genau ansehen. Dazu betrachten wir das folgende, etwas aufbereitete Ablaufprotokoll von PROLOG:

```
call  ?- s([john,loves,mary],[]).
         s(A               ,B ) :-
call        np([john,loves,mary],C),
            np(A1              ,B1) :-
call           n([john,loves,mary],B1).
               n([john|A2]       ,A2).
exit           n([john|[loves,mary]],[loves,mary])
exit        np([john,loves,mary],[loves,mary])
call        vp([loves,mary],[]).
            vp(A3          ,B3) :-
call           v([loves,mary],B3),
               v([loves|A4]  ,A4).
exit           v([loves|[mary]],[mary])
```

```
call          np([mary],[]).
              np(A5    ,B5) :-
call             n([mary]   ,B5).
                 n([mary|A6],A6).
exit             n([mary|[]],[])
exit          np([mary],[])
exit       vp([loves,mary],[])
exit    s([john,loves,mary],[])
     yes
```

Vergleicht man in diesem Protokoll die Zeile `call` für eine Konstituente mit der zugehörigen Zeile `exit`, so sieht man, wie jede Klausel genau das Anfangsstück von der Eingabeliste entfernt, das die zugehörige Konstituente bildet:

```
call   np([john,loves,mary],A)
exit   np([john,loves,mary],[loves,mary])
```

Der Grund, daß DCGs in diese spezielle Form übersetzt werden und nicht in unsere frühere Version mit `append`, liegt in der Effizienz dieses Mechanismus. In der `append`-Version muß PROLOG die Eingabeliste nacheinander in alle möglichen Teillisten zerlegen und dann testen, ob diese Zerlegung der Subkonstituentenstruktur der Produktion entspricht. Dieses Verfahren bedeutet ein blindes Probieren und anschließendes Testen.
Der DCG-Parser dagegen geht sehr zielgerichtet vor. Er nutzt die Tatsache aus, daß jede Konstituente „weiß“, wie sie intern aufgebaut ist. Es wird jeder Konstituente überlassen, sich am Anfang der Eingabeliste den für sie nötigen Teil abzuschneiden. Den Rest, mit dem sie nichts anzufangen weiß, wird einfach zur Bearbeitung an nachfolgende Konstituenten weitergegeben. Dadurch wird vermieden, daß die Eingabekette zunächst einmal blind zerlegt werden muß; stattdessen wird sie einfach von vorne her „aufgebraucht“, bis zum Schluß hoffentlich die leere Liste übrigbleibt.
Der Nachteil dieser Kodierung ist, daß zusätzliche Variablen in den Klauseln mitgeschleppt werden müssen. Dadurch wird das Programm schlechter lesbar und schwerer verständlich. Um diesen Nachteil wieder auszugleichen, bietet PROLOG die DCG-Notation an, die es dem Programmierer erspart, direkt den komplizierten PROLOG-Code zu schreiben.

9.1 Aufbau der Konstituentenstruktur

In der Regel werden wir nicht mit einem Programm zufrieden sein, das nach Eingabe eines Satzes nur lapidar mit `yes` oder `no` antwortet. Wir wollen als Ergebnis der Syntaxanalyse irgendeine Darstellung des Eingabesatzes erhalten, die als Eingabestruktur für eine Weiterverarbeitung dienen kann. Je nach Anwendung können die Anforderungen an dieses Ergebnis ganz verschieden sein.

- Sind wir an der syntaktischen Analyse selbst interessiert, so könnte eine Repräsentation des Syntaxbaumes selbst ein angemessenes Ergebnis sein. Beispiele dafür haben wir bereits gesehen.
- Beim Aufbau einer natürlichsprachlichen Schnittstelle für den Datenbankzugriff erwarten wir als Ergebnis ein Kommando, das die Datenbank zur Auswahl der gewünschten Daten veranlaßt.

  Who works in department 073/62?
  $\Longrightarrow$
  ```
  SELECT name FROM personnel-db
  WHERE dpt-no = '073/62'
  ```

- Als Semantiker sind wir an einer Übersetzung des Eingabesatzes in eine prädikatenlogische Formel interessiert.

  Every man loves a woman
  $\Longrightarrow$
  $\forall$ x $\exists$ y loves(x,y)

Es gibt also eine Vielzahl verschiedener, „sinnvoller“ Ausgaben für die Analyse eines Satzes. Ausgehend von einer Grammatik, wie wir sie bisher besprochen haben, ist die Erzeugung eines Syntaxbaumes zunächst die naheliegende, da einfachste, Aufgabe. Wir müssen ja lediglich eine Struktur aufbauen, die exakt die verwendeten Produktionen der Grammatik widerspiegelt. Zur Erzeugung einer anderen Ausgabe, z.B. einer semantischen Repräsentation, muß wesentlich mehr Aufwand getrieben werden. Dies wollen wir uns aber für das nächste Kapitel aufheben.
Als erstes benötigen wir eine geeignete Kodierung für einen Syntaxbaum als PROLOG-Term. Eine allgemeine Baumstruktur läßt sich folgendermaßen charakterisieren:
Ein Baum ist

1. ein Blatt *oder*
2. ein Knoten, von dem mindestens eine Kante ausgeht. Jede Kante führt wieder zu einem Baum

Bei Syntaxbäumen sind die Knoten mit Nichtterminalsymbolen markiert und die Blätter mit Lexemen. Wir wollen Bäume so kodieren, daß wir die Markierung der Knoten als Funktoren von Termen verwenden. Der Term hat als Argumente die Unterbäume. Die Blätter sind PROLOG-Atome.

Beispiel 9.2 Der Syntaxbaum für John eats the apple aus Beispiel 8.3 wird in PROLOG kodiert als

```
s(
  np(
     n(john)
    ),
  vp(
     v(eats),
     np(
        det(the),
        n(apple)
       )
    )
)
```

Wie können wir nun einen solchen Syntaxbaum während des Parsings aufbauen? Betrachten wir eine Produktion unserer Grammatik:

S $\rightarrow$ NP VP

Diese Produktion besagt zweierlei:

1. Um eine Eingabekette als Satz analysieren zu können, muß der erste Teil der Kette als NP und der zweite Teil als VP analysierbar sein. Diese Aussage wurde bisher von unserem Parser verwendet.

2. Der Syntaxbaum für eine Konstituente vom Typ S hat als Wurzel einen Knoten, der mit S markiert ist. Außerdem hat er zwei Unterbäume, deren Wurzeln mit NP bzw. VP markiert sind und diese sind die Syntaxbäume für die Konstituenten NP bzw. VP.

Beim Zugriff auf das Lexikon können wir den betreffende Ast des Syntaxbaumes abschließen und als Blatt das Lexem einsetzen.
Dieses Konstruktionsverfahren für den Syntaxbaum können wir direkt nach PROLOG übertragen. Der DCG-Formalismus erlaubt es, die Konstituenten in einer Produktion mit zusätzlichen Argumenten zu versehen. Ein solches Argument können wir benutzen, um bei jedem Ableitungsschritt den Syntaxbaum mit aufzubauen.
Die Produktion S $\rightarrow$ NP VP sieht als DCG-Produktion folgendermaßen aus:

```
s(s(NP,VP)) -->
   np(NP),
   vp(VP).
```

Diese Regel besagt:

> Der Syntaxbaum für die Konstituente S, die mit dieser Produktion gebildet werden kann, hat die Form `s(NP,VP)`, wobei `NP` der Syntaxbaum für die Konstituente NP und `VP` für VP ist.

Die Übersetzung der DCG-Produktion in eine PROLOG-Klausel ist:

```
s(s(NP,VP),A,B) :-
   np(NP,A,C),
   vp(VP,C,B).
```

Die zusätzlichen Argumente in einer DCG-Produktion werden also immer am Anfang der Argumentlisten der PROLOG-Prädikate eingetragen. Das hat z.B. beim Ablaufprotokoll den Vorteil, daß die„wichtigen“ Argumente vorne stehen und daher besser erkennbar sind.
Hier die gesamte DCG für die Grammatik aus Beispiel 8.1 mit Konstruktion des Syntaxbaumes:

```
s(s(NP,VP)) -->
   np(NP),
   vp(VP).

np(np(DET,NP)) -->
   det(DET),
   np(NP).
np(np(N)) -->
   n(N).

vp(vp(V,NP)) -->
   v(V),
   np(NP).
vp(vp(V)) -->
   v(V).
```

```
n(n(mary))     --> [mary].
n(n(john))     --> [john].
n(n(woman))    --> [woman].
n(n(man))      --> [man].
n(n(apple))    --> [apple].

det(det(the))  --> [the].

v(v(loves))    --> [loves].
v(v(eats))     --> [eats].
v(v(sings))    --> [sings].
```

Eine Anfrage als Beispiel:

```
?- s(X,[john,eats,the,apple],[]).
X = s(np(n(john)),vp(v(eats),np(det(the),n(apple))))
yes
```

9.2 Nur kontextfrei?

Eine wichtige Bemerkung ist an dieser Stelle fällig: Der DCG-Formalismus ist zur Beschreibung von Sprachen geeignet, die nicht mehr durch eine kontextfreie Grammatik beschrieben werden können. Der Formalismus ist wesentlich mächtiger. Dies liegt daran, daß die zusätzlichen Argumente in DCG-Produktionen nicht nur zur Konstruktion von Strukturen verwendet werden können, wie dies in unserem Beispiel der Fall war. Sie können auch zur Formulierung von Bedingungen verwendet werden, die Kontextabhängigkeiten ausdrücken.

Beispiel 9.3 Das Lieblingsbeispiel der Informatiker für eine einfache Sprache, die nicht mehr durch eine kontextfreie Grammatik beschreibbar ist, ist die Menge L = $\{a^n b^n c^n | n \geq 1\}$, d.h. die Sätze der Sprache bestehen aus einer Anzahl a, gefolgt von der gleichen Anzahl b, gefolgt von der gleichen Anzahl c; also: abc, aabbcc, aaabbbccc, ..., aber nicht: aabc oder aaabbccc.
Eine DCG für diese Sprache läßt sich leicht angeben:

```
s(Anzahl) -->
   ap(Anzahl),
   bp(Anzahl),
   cp(Anzahl).

ap(1) -->
   a(1).
ap(Anzahl+1) -->
   a(1),
   ap(Anzahl).

bp(1) -->
   b(1).
bp(Anzahl+1) -->
   b(1),
```

```
   bp(Anzahl).

cp(1) -->
   c(1).
cp(Anzahl+1) -->
   c(1),
   cp(Anzahl).

a(1) --> [a].
b(1) --> [b].
c(1) --> [c].
```

Die Produktionen für `ap`, `bp` und `cp` erlauben beliebig viele a, b bzw. c. Das zusätzliche Argument wird dazu benutzt, eine Struktur aufzubauen, die die Anzahl repräsentiert. Die kritische Produktion ist die für `s`, wo über das zusätzliche Argument gefordert wird, daß die Anzahl der a, b und c übereinstimmt.

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

9.3 Erweiterungen

Bisher hatten alle Teile der DCG-Produktionen direkt mit der Verarbeitung der Eingabe zu tun. Manchmal möchte man jedoch die Möglichkeit haben, zusätzlich irgendwelche Tests oder Aktionen auszuführen, die nicht direkt zu den Produktionen der Grammatik gehören. PROLOG bietet die Möglichkeit, in geschweiften Klammern $\{\}$ beliebige PROLOG-Ausdrücke in DCG-Produktionen einzufügen. Diese werden vom DCG-Übersetzer dann einfach unverändert in die PROLOG-Klausel übernommen.
Ein Beispiel für die Anwendung dieser Erweiterung bietet die Darstellung des Lexikons in unserer Beispielgrammatik. Es ist sehr aufwendig, wie bisher für jeden Lexikoneintrag eine Grammatikregel zu formulieren. Stattdessen möchte man das Lexikon einfach als eine Mengen von Paaren (Lexem,Kategorie) darstellen können:

```
lex(mary,n).
lex(john,n).
lex(woman,n).
lex(man,n).
lex(apple,n).
lex(the,det).
lex(loves,v).
lex(eats,v).
lex(sings,v).
```

Für jede lexikalische Kategorie benötigen wir dann nur noch eine DCG-Produktion:

```
n(n(N)) -->
   [N],
   { lex(N,n) }.
```

Der DCG-Übersetzer baut daraus die PROLOG-Klausel

```
n(n(N),[N|A],A) :-
   lex(N,n).
```

Diese Klausel entsteht dadurch, daß der DCG-Übersetzer den ersten Teil der Produktion wie gewohnt umformt und den Teil in geschweiften Klammern unverändert in den Rumpf der Klausel übernimmt.
Auf diese Weise haben wir eine sehr elegante und einfache Formulierung der Grammatik und außerdem eine *einheitliche Lexikonzugriffsfunktion* durch `lex`.
Dieser Mechanismus ist besonders wertvoll, da man so sehr einfach die Realisierung des Lexikons verändern kann, ohne daß Grammatikregeln davon betroffen sind. Man muß lediglich die Schnittstelle zum Lexikon `lex` anpassen. Beispielsweise könnten wir so eine Morphologiekomponente einbauen, die nicht unmittelbar in einem Lexikon nachschaut, sondern zuerst eine morphologische Analyse des Wortes durchführt und dann anstelle eines Vollformenlexikons ein sehr viel kompakteres Stammformenlexikon benutzt.
Überhaupt ist es ein guter Programmierstil, möglichst wenige implizite Abhängigkeiten in den Programmen zu kodieren. Stattdessen sollten geeignete Schnittstellen definiert werden, die unabhängig von den internen Strukturen der Prädikate sind.

9.4 Übungen

Übung 9.1 Erweitern Sie die DCG aus Beispiel 9.1 durch ein zusätzliches Argument, mit dem die Übereinstimmung von Person und Numerus an den entsprechenden Stellen gesichert wird. Sätze wie John eats an apples oder John eat the apple sollen als ungrammatikalisch erkannt werden.

Übung 9.2 Führen Sie Verben mit verschiedenen Subkategorisierungen ein. Betrachten Sie dazu folgende Sätze:

John sleeps.
$*$ John sleeps the apple.
$*$ Mary likes.
Mary likes John.
John sings.
John sings a song.

Übung 9.3 Die Sprache L = $\{\alpha c \alpha | \alpha \in \{a, b\}^*\}$, d.h. die Sätze der Sprache sind eine beliebige Folge von a und b, gefolgt von einem c, gefolgt von der identischen Folge von a und b, ist ebenfalls nicht durch eine kontextfreie Grammatik beschreibbar. Geben Sie eine DCG an, die genau diese Sprache erzeugt.

Kapitel 10

Deutsch statt PROLOG

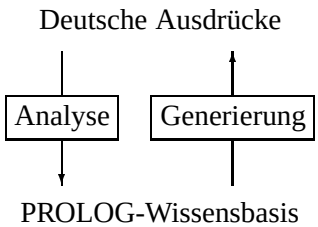
Immer mehr Information wird heutzutage in Computern gespeichert. Um auf diese Information schnell und zuverlässig wieder zugreifen zu können, wird sie in Ausdrücken formaler Sprachen abgelegt und nicht in natürlicher Sprache. Natürliche Sprachen scheinen zur kompakten und leicht verfügbaren Informationsdarstellung im Computer nicht geeignet, weil sie „mathematisch“ unerwünschte Eigenschaften wie zum Beispiel Mehrdeutigkeit besitzen. Der Bezug zwischen Ausdrücken natürlicher Sprache und den durch sie beschriebenen Objekten ist nicht immer eindeutig:

Emilie gab Johanna eine Schallplatte. Sie ....

In diesem Beispiel ist zum Zeitpunkt des Schreibens oder Sprechens des Wortes sie nicht klar, worauf es sich bezieht.

Trotz derartiger Probleme wäre es praktisch, wenn man, zumindest in eingeschränkter Art und Weise, in natürlicher Sprache mit dem Computer kommunizieren könnte. Dazu muß natürliche Sprache (zum Beispiel Deutsch) in die gewählte formale Sprache übersetzt werden und zurückübersetzt werden. Der Weg von natürlichsprachlichen Ausdrücken zu formalen Ausdrücken heißt *Sprachanalyse* . Die Sprachanalyse läßt sich wiederum in (mindestens) zwei Aufgabenteile aufgliedern: die syntaktische Analyse und den Aufbau semantischer Repräsentationen, auch *Semantikkonstruktion* genannt.

Der Begriff „semantische Repräsentation“ stammt daher, daß in der gewählten formalen Sprache (z.B. PROLOG) die „Bedeutungen“, d.h. die „Semantik“ der Sätze dargestellt werden sollen. Die Rückübersetzung formalsprachlicher Ausdrücke in natürliche Sprache heißt *Sprachgenerierung* . BeideÜbersetzungsrichtungen sind im Grunde genommen Abbildungen zwischen Ausdrücken der zu betrachtenden natürlichen Sprache und Ausdrücken der gewählten formalen Sprache. Die Sprachanalyse setzt auf der natürlichen Sprache als Quellsprache auf und erzeugt Ausdrücke der formalen Sprache als Zielsprache. Die Quellsprache für die Sprachgenerierung ist die betreffende formale Sprache, ihre Zielsprache ist die entsprechende natürliche Sprache.



Zur Demonstration der Vorgehensweise bei Analyse und Generierung wählen wir die drei folgenden Beispielsätze:

1. Alle Menschen sind sterblich.
2. Sokrates ist ein Mensch.
3. Sokrates ist sterblich.

Die Übersetzungen dieser Ausdrücke nach PROLOG sollen lauten:

```
1. sterblich(X) :- mensch(X).
2. mensch(sokrates).
3. sterblich(sokrates).
```

10.1 Sprachanalyse

In den einen Aufgabenbereich der Sprachanalyse, die syntaktische Analyse, haben wir anhand der Definite Clause Grammars schon etwas Einblick bekommen. Jetzt soll der zweite Teil hinzugefügt werden, der Aufbau einer semantischen Repräsentation, in unserem Fall von PROLOG-Ausdrücken.
Ein erster Ansatz wäre, den deutschen Satz Wort für Wort nach PROLOG zu übersetzen. Dies erweist sich doch sehr schnell als ungeschickt. Wir haben bereits gesehen, wie die „syntaktische Struktur“, der „Syntaxbaum“, eines Satzes aufgebaut werden kann. Die Struktur des Syntaxbaums entspricht der Verschachtelung der Aufrufe beim Analysevorgang. Eine Wort-für-Wort-Übersetzung würde auf der Information über die Reihenfolge der Wörter im Satz basieren. Da wir aber durch die Syntaxanalyse sozusagen kostenlos Information über die syntaktische Struktur (Konstituentenstruktur) eines Satzes zur Verfügung bekommen, könnten wir nun versuchen, unsere Übersetzungsvorschrift auf dem Syntaxbaum bzw. korrespondierend zu den Syntaxregeln zu formulieren. Die eigentliche Syntaxanalyse dient dann nicht mehr allein zum Testen der Grammatikalität von Sätzen, sondern liefert gleichzeitig das Grundgerüst, entlang dem die semantische Repräsentation des Satzes aufgebaut wird.
Wir gehen dabei davon aus, daß im Lexikon jedes einzelne Wort mit einer *partiellen semantischen Repräsentation* versehen ist. „Partiell“ deswegen, weil diese Repräsentationen auf lexikalischer Ebene noch keine akzeptablen Formeln der formalen Zielsprache sind, sondern nur Teile solcher Formeln oder mit Platzhaltern versehene Schemata für solche Formeln darstellen.
Aufbauend auf den partiellen semantischen Strukturen im Lexikon müssen wir dann jeder Syntaxregel in Form einer *Konstruktionsregel* die Information mitgeben, wie aus den partiellen Repräsentationen der in der entsprechenden Syntaxregel erwähnten Konstituenten auf der rechten Regelseite, die semantische Repräsentation für die Konstituente der linken Regelseite zusammengebaut wird. Das heißt, die Syntaxregeln werden um die semantischen Konstruktionsregeln *augmentiert* . Den Aufbau semantischer Repräsentationen entlang der syntaktischen Regeln nennt man *kompositionellen Aufbau* .
Für die Wahl der geeigneten partiellen semantischen Repräsentationen für bestimmte Wörter und Wortklassen und für die Regeln über den Zusammenbau von Formelteilen gibt es keine verbindlichen Richtlinien, jedoch Erfahrungswerte und Traditionen. Das Vorgehen ist im allgemeinen in der Art, daß man sich überlegt, wie sinnvollerweise die Repräsentation für einen kompletten Satz aussehen soll und welche Information einzelne Konstituenten maximal „von sich aus“ mitbringen können. Wenn die Satzrepäsentation dann mehr Informationen zu enthalten hätte, als durch einen kompositionellen Zusammenbau der Teilrepräsentationen möglich ist, muß man sich eventuell mit einer schwächeren Darstellung für den Gesamtsatz zufrieden geben, oder doch noch versuchen, die lexikalischen Teilrepräsentationen „schlauer“ zu machen oder die Konstruktionsregeln zu verbessern. Dabei sollte immer als Leitlinie gelten:

Die Konstruktionsregeln für die semantische Repräsentation sollten bezüglich der syntaktischen Regeln und der lexikalischen Wortklassen so systematisch und so allgemein wie möglich konzipiert werden.

10.1.1 Übersetzungskonventionen

Bevor wir zur Bearbeitung oben genannter Beispielsätze gelangen, sollen als kleiner Exkurs einige Hinweise für die Zuordnung partieller semantischer Repräsentationen zu Wortklassen gegeben werden (siehe [Bonevac 1987]). Man soll jedoch nicht übersehen, daß jede dieser Empfehlungen bei der Verwendung zur Beschreibung eines nichttrivialen Sprachfragments sofort Probleme verursacht, die Gegenstand der Semantikforschung sind.

Eigennamen

Eigennamen lassen sich in Individuenkonstanten, das heißt in PROLOG-Konstanten übersetzen.

Beispiel 10.1 *Eigennamen*

Deutsch	PROLOG
Sokrates	`sokrates`

Nomina

Ein Nomen bedeutet im allgemeinen eine Klasse von Objekten und läßt sich deswegen als (einstelliges) Prädikat übersetzen.

Beispiel 10.2 *Nomina*

Deutsch	PROLOG
Mensch	`mensch(X)`

Allquantifizierte Nominalphrasen

Die Wörter alle, jede und generisch verwendete unbestimmte Artikel werden in Allquantoren übersetzt, da in Datenbasis-Einträgen alle Variablen implizit allquantifiziert sind, heißt das einfach, daß alle, jede etc. in Variablen übersetzt werden.

Beispiel 10.3 *Allquantifizierung*

Deutsch	PROLOG
alle Menschen	`mensch(X)`

Allquantifizierende Ausdrücke machen eigentlich nur Sinn, wenn noch ein weiteres Prädikat dabei steht, wie zum Beispiel in dem Satz Alle Menschen sind sterblich. Die Übersetzung dieses Satzes wird klarer, wenn man ihn umformuliert in Wenn jemand ein Mensch ist, dann ist er sterblich.

Beispiel 10.4 *Allquantifizierung mit Modifikation*

Deutsch	PROLOG
alle Menschen sind sterblich	`sterblich(X):-mensch(X)`

Existentiell quantifizierte Nominalphrasen

Existentiell quantifizierte Nominalphrasen in Aussagesätzen werden schlichtweg in ein Prädikat mit einer Konstanten als Argument übersetzt. Das heißt, die Existenz des betreffenden Objektes wird postuliert.

Beispiel 10.5 *Existenzquantifizierung*

Deutsch	PROLOG
Es gibt einen Menschen.	`mensch(789423789).`

Existentiell quantifizierte Nominalphrasen in Fragesätzen werden in Variablen (in Zielklauseln) übersetzt.

Beispiel 10.6 *Existenzquantifizierung in Fragen*

Deutsch	PROLOG
Gibt es einen Menschen?	`?- mensch(X).`

Adjektive

Adjektive können in vielen Fällen als einstellige Prädikate übersetzt werden:

Beispiel 10.7 *Adjektive*

Deutsch	PROLOG
sterblicher Sokrates	`sterblich(sokrates).`

Dasselbe gilt für Relativsätze, durch die ebenfalls Eigenschaften bestimmter Objekte und Individuen näher spezifiert werden:

Beispiel 10.8 *Relativsätze*

Deutsch	PROLOG
Sokrates, der sterblich ist	`sterblich(sokrates)`

Präpositionen

Präpositionen stellen im allgemeinen zweistellige Prädikate dar, also Relationen, d.h. Beziehungen zwischen zwei Objekten:

Beispiel 10.9 *Präpositionen*

Deutsch	PROLOG
X auf Y	`auf(X,Y)`

Präpositionalphrasen

Präpositionalphrasen besitzen die gleiche Darstellung wie Präpositionen mit dem Unterschied, daß bereits eines der beiden durch die Präposition in Bezug gesetzten Objekte näher beschrieben ist:

Beispiel 10.10 *Präpitionalphrasen*

Deutsch	PROLOG
X auf der Straße	`auf(X,Y),strasse(Y)`

Ein, von einer Präpositionalphrase modifiziertes Nomen bekommt dann folgende Darstellung:

Beispiel 10.11 *Nomen mit PP*

Deutsch	PROLOG
Kind auf der Straße	`auf(X,Y),kind(X),strasse(Y)`

Verben

Verben werden in Prädikate übersetzt. Die syntaktische Valenz ergibt im allgemeinen die Stelligkeit des Prädikats.

Beispiel 10.12 *Verben*

Deutsch	PROLOG
Johann schläft.	`schlafen(johann)`
Peter ißt Apfelmus.	`essen(peter,apfelmus)`

10.2 Hilfsmittel

Nach diesem Exkurs wollen wir wieder zu den drei Beispielsätzen zurückkehren und zuerst einmal festhalten, durch welche Arten von Ausdrücken sie repräsentiert werden sollen. Die Ausdrücke `mensch(sokrates)` und `sterblich(sokrates)` sind einstellige Terme, `sterblich(X):-mensch(X)` ist eine Klausel, deren Rumpf nur ein Prädikat enthält.
Das erste Problem beim Aufbau von PROLOG-Ausdrücken mittels PROLOG ist, daß die Funktoren von Termen und Prädikatsnamen immer instantiiert sein müssen, variable Funktoren und Prädikatsnamen sind ausgeschlossen. Während des Aufbaus eines Ausdrucks kann es jedoch sehr wohl vorkommen, daß zwar bekannt ist, daß an einer bestimmten Stelle ein Funktor stehen muß, aber dessen Instantiierung noch nicht bekannt ist. Funktoren müssen also, zumindest vorübergehend auf Argumentstellen von Termen stehen, denn dort sind Variablen erlaubt. Wir wählen für einstellige Terme bzw. für einstellige Prädikate die Notation `term(Funktor,Argument)`.
Die zweite Schwierigkeit ist, daß Übersetzungen für ganze Sätze unterschiedlichen Schemata anzugehören scheinen. Einmal wird ein Satz als eine Klausel übersetzt: `a(X):-b(X)`, einmal als Fakt `c(Y)`. Homogenität würde jedoch den Übersetzungsprozeß erleichtern. Dieses Dilemma löst sich von selbst, wenn man die interne Struktur von Fakten ansieht: Fakten sind degenerierte Klauseln der Form: `c(Y):-true`. Damit lauten dann die Beispiele in unserer Schreibweise von PROLOG-Ausdrücken so:

1. `term(sterblich,X) :- term(mensch(X))`
2. `term(mensch,sokrates):-true`
3. `term(sterblich,sokrates):-true`

10.2.1 Die DCG für die Syntaxanalyse

Die Semantikkonstruktion braucht die Syntaxanalyse als Rahmen. Dieser soll aus folgender DCG bestehen, die wir dann sukzessive um die Semantikkonstruktion erweitern.

```
% Syntaxregeln
s --> np, vp.
np --> eigenname.
np --> artikel1, nomen.
np --> artikel2, nomen.
```

```
vp --> kopula, praedikatsnomen.
praedikatsnomen --> adjektiv.
praedikatsnomen --> np.


% Lexikon
artikel1 --> [alle].
artikel2 --> [ein].
eigenname --> [sokrates].
nomen --> [menschen].
nomen --> [mensch].
kopula --> [ist].
kopula --> [sind].
adjektiv --> [sterblich].
```

10.2.2 Augmentierung des Lexikons

Wir beginnen beim Lexikon mit den Erweiterungen, die die Semantikkonstruktion ermöglichen. Jedem Lexikoneintrag wird eine partielle semantische Repräsentation zugeordnet. Die Partialität wird dadurch ausgedrückt, daß die Ausdrücke an bestimmten Stellen Variablen als Platzhalter für später noch zuzufügende Information enthalten.

- Für folgende Wörter ist die zugeordnete Struktur leer: ein, ist, sind.
- Mensch und sterblich werden mit einstelligen Termen versehen.
- Die Übersetzung des Wörtchens alle besagt, daß die Argumente von Kopf und Rumpf der zu erstellenden Klausel gleich sein sollen, was über diese Argumente prädiziert wird, bleibt noch offen.
- Der Eigennamen Sokrates bekommt im Prinzip die gleiche Übersetzung wie alle, nur mit dem Unterschied, daß der Rumpf der Klausel sofort mit der trivialen Klausel `true` instantiiert wird.

Fassen wir dies nocheinmal in einer Tabelle zusammen:

Wort	PROLOG-Darstellung
alle	`term(_,X):-term(_,X)`
sokrates	`term(_,sokrates):-true`
mensch	`term(mensch,_)`
menschen	`term(mensch,_)`
sterblich	`term(sterblich,_)`

10.2.3 Konstruktionsregeln

So wie die Lexikoneinträge um partielle semantische Strukturen augmentiert worden sind, müssen zu den Syntaxregeln die Vorschriften über das Zusammenbauen von PROLOG-Ausdrücken hinzugefügt werden. Als erstes wird jedes Nichtterminal der DCG, dem eine nichttriviale Übersetzung zugeordnet werden soll, mit einer Variablen als Platzhalter für die zu erstellende Struktur versehen. Unter den eigentlichen Konstruktionsregeln gibt es in unserem einfachen Beispiel zwei Typen:

1. Die Struktur wird unverändert von einer Konstituenten der rechten Seite einer Syntaxregel an die Konstituente auf der linken Regelseite „weitergereicht“.

2. Zwei Übersetzungen von Konstituenten der rechten Regelseite werden miteinander zur Übersetzung der linken Regelseite verknüpft.

Das simple Weiterreichen von Strukturen wird erreicht durch die Verwendung desselben Variablennamens links und rechts des Regelpfeils. Für die Verknüpfung zweier Strukturen gibt es wiederum zwei Fälle. Die Übersetzung allquantifizierter Nominalphrasen wird erzeugt durch Unifikation der Übersetzung des Nomens mit dem Rumpf der Übersetzung des Artikels alle. Die, einem ganzen Satz entsprechende Klausel entsteht, wenn man den Kopf der Übersetzung der Nominalphrase mit der Übersetzung der Verbalphrase unifiziert. Es werden also die beiden folgenden Prädikate benötigt, mit jeweils drei Argumenten:

```
%   Konstruktionsregeln

konstruiere_s(NPStruktur,VPStruktur,SStruktur):-
    NPStruktur = (VPStruktur:-_),
    SStruktur = NPStruktur.
konstruiere_np(ArtStruktur,NStruktur,NPStruktur):-
    ArtStruktur = (_:-NStruktur),
    NPStruktur = ArtStruktur.
```

10.2.4 Die augmentierte Grammatik

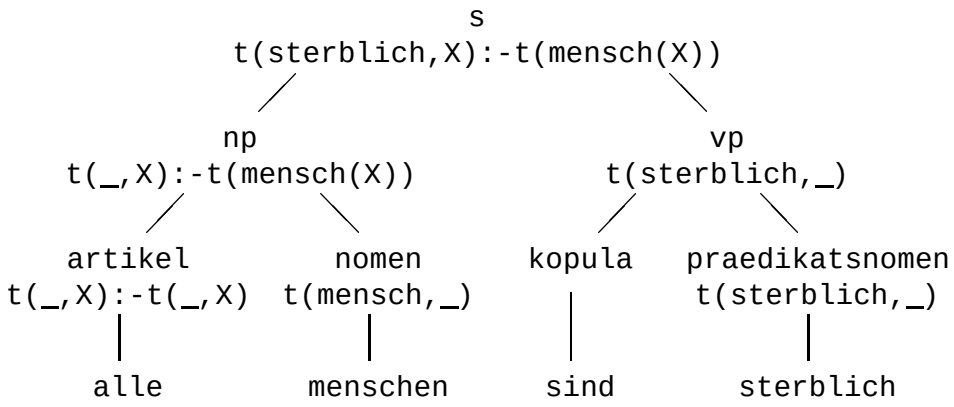
Zusammen mit den eben erwähnten Konstruktionsregeln erhalten wir nun eine funktionsfähige Grammatik, die beim Parsing PROLOG-Ausdrücke in unserer eigenen Notation aufbaut.

```
% Syntaxregeln

s(SStruktur) -->
   np(NPStruktur),
   vp(VPStruktur),
   {konstruiere_s(NPStrukur,VPStruktur,SStruktur)}.

np(Struktur) --> eigenname(Struktur).
np(NPStruktur) -->
   artikel1(ArtStruktur),
   nomen(NStruktur),
   {konstruiere_np(ArtStruktur,NStruktur,NPStruktur)}.
np(Struktur) -->
   artikel2,
   nomen(Struktur).

vp(Struktur) -->
   kopula,
   praedikatsnomen(Struktur).

praedikatsnomen(Struktur) -->
   adjektiv(Struktur).
praedikatsnomen(Struktur) -->
   np(Struktur).
```

```
% Lexikon
artikel1((term(_,X):-term(_,X))) --> [alle].
artikel2 --> [ein].
eigenname((term(_,sokrates):-true)) --> [sokrates].
nomen(term(mensch,_)) --> [menschen].
nomen(term(mensch,_)) --> [mensch].
kopula --> [ist].
kopula --> [sind].
adjektiv(term(sterblich,_)) --> [sterblich].
```

10.2.5 Beispiellauf der Grammatik

Statt des relativ unübersichtlichen Ablaufprotokolls (`trace`) wird hier der, eigentlich nur implizit vorhandene Syntaxbaum dargestellt. Jeder Knoten ist mit dem, „bis dorthin“ aufgebauten Ausdruck annotiert. Aus Platzgründen wird hier das Prädikat `term` mit `t` abgekürzt.



10.3 Generierung

Wir haben gesehen, wie man vom Satz zu einer PROLOG-Struktur kommt. Der umgekehrte Weg ist auch möglich. Die eben erarbeitete DCG läßt sich prinzipiell auch als Generator verwenden, indem man diesmal die Argumentstelle für den Satz variabel läßt und stattdessen die Variable für die dem Satz entsprechende Struktur instantiiert:

```
?- s(Satz,[],(term(sterblich,X):-term(mensch(X)))).
Satz = [alle,menschen,ist,sterblich];
Satz = [alle,menschen,sind,sterblich];
Satz = [alle,mensch,ist,sterblich];
Satz = [alle,mensch,sind,sterblich];
no
```

Dieser Beispielaufruf zeigt sofort eines der Probleme der Generierung auf: Die semantische Struktur enthält keine Angaben über morphologische Restriktionen. Die zweite Schwachstelle wird ersichtlich, wenn man sich das Ablaufprotokoll dieses Beweises anschaut. Die verschiedenen Übersetzungen werden nur mühselig durch sehr vieles Backtracking „erraten“.

Ein erster und ein wenig effizienterer Ansatz zur Generierung, der hier aber aus Platzgründen auch der einzige bleiben soll, muß folgendes berücksichtigen:

1. Die Übersetzungen von Teilen der semantischen Struktur müssen aufgrund morphologischer und anderer Restriktionen (z.B. Einbettung in eine Redesituation, in einen Text) gefiltert werden.

2. Die Generierung der Konstituenten eines Satzes sollte rekursiv über der semantischen Struktur ablaufen statt über dem noch zu erratenden Satz.

Zumindest die Forderung nach Berücksichtigung morphologischer Restriktionen läßt sich leicht erfüllen, indem man die nichtterminalen Symbole um jeweils mit einer weiteren Variablen versieht, die z.B. den Numerus-Abgleich zwischen Artikel und Nomen und zwischen Subjekt und Verb gestattet.
Eine Konsequenz der zweiten Forderung ist, daß Definite Clause Grammars zum Einsatz für die Generierung denkbar ungeeignet sind, weil sie entlang des Satzes arbeiten und nicht entlang der semantischen Struktur.
Vorweg läßt sich sagen, daß wir für die Generierung eine spezielle Notation für PROLOG-Ausdrücke unnötig wird, da die semantischen Strukturen ja immer voll instantiiert sind. Damit läßt sich die Rekursion über die semantischen Strukturen in unserem Beispiel folgendermaßen beschreiben:

> Der in den Ausdrücken auf oberste Ebene auftretende „Funktor“ `:-` stammt entweder aus der Übersetzung eines Eigennamen oder eines allquantifizierenden Artikels.
>
> 1. Der Kopf einer Klausel ist immer die Übersetzung einer Verbalphrase. Verbalphrasen beziehen in unserem Beispiel ihre Semantik entweder von
>   (a) einem prädikativ verwendeten Adjektiv
>   (b) oder einer Nominalphrase.
> 2. Der Rumpf einer Klausel ist bei Eigennamen trivial, bei allquantifizierten Nominalphrasen ist er mit der Übersetzung des Nomens gefüllt.

Insgesamt beschreiben alle Prädikate dreistellige Relationen zwischen semantischen Ausdrücken, morphologischen Restriktionen und Wortfolgen.

```
%  Syntax fuer die Generierung
s((VPStruktur:-true),_,[Name|VPString]):-
   eigenname(_,_,Name),
   vp(VPStruktur,singular,VPString).
s((VPStruktur:-NStruktur),_,[Artikel,Nomen|VPString]):-
   artikel1(_,Numerus,Artikel),
   nomen(NStruktur,Numerus,Nomen),
   vp(VPStruktur,Numerus,VPString).

np(Struktur,_,[Artikel,Nomen]):-
   artikel2(_,Numerus,Artikel),
   nomen(Struktur,Numerus,Nomen).

vp(Struktur,Numerus,[Kopula|PraedNomen]):-
   kopula(_,Numerus,Kopula),
   praedikatsnomen(Struktur,_,PraedNomen).

praedikatsnomen(Struktur,_,[Adjektiv]):-
   adjektiv(Struktur,_,Adjektiv).
```

```
praedikatsnomen(Struktur,_,NPString):-
   np(Struktur,_,NPString).

%  Festlegungen auf lexikalischer Ebene

artikel1(_,plural,alle).

artikel2(_,singular,ein).

eigenname(_,singular,sokrates).

nomen(mensch(_),singular,mensch).
nomen(mensch(_),plural,menschen).

kopula(_,singular,ist).
kopula(_,plural,sind).

adjektiv(sterblich(_),_,sterblich).
```

Aufgerufen wird dieses Programm zum Beispiel in folgender Weise, hier gleich mit der Antwort, dem generierten Satz, versehen:

Beispiel 10.13 *Beispiellauf des Generators*

```
?- s((sterblich(X):-mensch(X)),_,Satz).
Satz = [alle,menschen,sind,sterblich]
```

10.4 Zusammenfassung

In diesem Kapitel haben wir gesehen, wie zusammen mit der Syntaxanalyse semantische Strukturen aufgebaut werden können und wie aus semantischen Strukturen wiederum Sätze erzeugt werden. Prinzipiell ist für beide „Übersetzungsrichtungen“ der gleiche Mechanismus verwendbar. Legt man jedoch Wert auf Effizienz, sollte die Syntaxanalyse sich am Satz und seiner syntaktischen Struktur orientieren. Die Generierung sollte stattdessen rekursiv über ihrer Eingabestruktur, nämlich der semantischen Repräsentation, verlaufen.

10.5 Übungen

Übung 10.1 Spielen Sie mit der oben eingeführten DCG und verfolgen Sie einerseits den Aufbau von PROLOG-Ausdrücken beim Parsen als auch den Aufbau eines Satzes bei der Generierung. Vergleichen Sie damit im Gegenzug die etwas effizientere Version des Generators.

Übung 10.2 Erweitern Sie die Syntaxregeln derart, daß auch die Wortstellung in folgenden Fragesätze beschrieben wird:

1. Wer ist sterblich?
2. Ist Sokrates ein Mensch?
3. Ist Sokrates sterblich?

Die Zielrepräsentationen sollten jeweils lauten:

1. term(sterblich,_):-true

2. term(mensch,sokrates):-true
3. term(sterblich,sokrates):-true

Übung 10.3 Erweitern Sie die Grammatik, so daß Adjektive auch pränominal (und nicht nur prädikativ) verwendet werden können:

alle schlauen menschen

Dazu wird eine Erweiterung unserer selbstgebastelten Notation von PROLOG-Ausdrücken nötig sein!

Kapitel 11

Eingebaute Prädikate

Wir haben in unseren bisherigen Programmen stets nur „reines“ PROLOG, sogenanntes *pure PROLOG*, verwendet. Das bedeutet, daß wir immer nur logische Formeln zur Beschreibung eines Problems entwickelt haben und der Inferenzmaschine von PROLOG die Lösungssuche zur Beantwortung von Anfragen überlassen haben. Die Instantiierung von Variablen in der Anfrage stellte die gesuchte Lösung dar. Lediglich beim Schreiben von rekursiven Prädikaten mußten wir gelegentlich die Abarbeitungsreihenfolge der Klauseln durch den PROLOG-Interpreter berüchsichtigen, um Endlosschleifen zu vermeiden. Dies schränkte uns bei der Spezifikation der Wissensbasis etwas ein; wir konnten nicht die volle Mächtigkeit der Hornklausel-Logik ausnützen.

In diesem Kapitel wollen wir nun sehen, welche Möglichkeiten es gibt, Einfluß auf den Beweis, den PROLOG durchführt, zu nehmen und wie wir dies zu unserem Vorteil ausnützen können. Die vorgestellten Mechanismen machen PROLOG erst zu einer vollwertigen Programmiersprache – mit allen Vor- und Nachteilen. Einerseits wird es nämlich möglich, Probleme zu lösen, die nicht mit der eingeschränkten Hornklausel-Logik beschreibbar sind. Andererseits können solche Programme nicht mehr als deklarative Spezifikation eines Problems aufgefaßt werden, wobei die Lösungssuche allein Sache der Inferenzmaschine ist. Stattdessen nimmt der Programmierer die Lösungssuche in die eigene Hand und steuert die Inferenzmaschine in die von ihm gewollte Richtung. Das bedeutet aber, daß diese Programme nur noch prozedural, also durch die Analyse der zeitlichen Abfolge von Bearbeitungsschritten, verstanden werden können.

Die Integration der Erweiterungen in PROLOG geschieht durch Prädikate, die vom System zur Verfügung gestellt werden. Diese Prädikate sind fester Bestandteil des Systems; daher nennen wir sie auch *eingebaute Prädikate* oder *built-in Prädikate*.

In der Regel müssen wir immer zwei Aspekte eines solchen Prädikats betrachten:

1. die lokale logische Bedeutung

2. den globalen Seiteneffekt

Bisher haben wir uns immer nur für die logische Bedeutung eines Prädikats interessiert. Diese ist bei eingebauten Prädikaten aber meist von untergeordneter Wichtigkeit.

Der wichtigste Aspekt ist oft der *Seiteneffekt* eines Prädikats. Mit Seiteneffekt ist gemeint, daß der Aufruf des Prädikats durch die Inferenzmaschine irgeneine Aktion bewirkt, die nicht ausschließlich von lokaler Bedeutung für dieses Prädikat ist, sondern global eine Veränderung des Systemzustandes bewirkt.

Solange wir nur „reines“ PROLOG verwenden, gibt es keine Seiteneffekte.

Ohne die in diesem Kapitel vorgestellten Erweiterungen kann man in der Praxis mit PROLOG nicht auskommen. Das bedeutet aber nicht, daß wir gewillt sind, alle bisher gezeigten Vorteile von PROLOG aufzugeben. Ganz im Gegenteil. Im nächsten Kapitel wollen wir an einem großen Beispiel zeigen, wie sich prozedurale und deklarative Programmteile sauber und elegant zu einem großen Ganzen verbinden lassen.

11.1 Ein- und Ausgabe

Soll ein Programm einen Dialog mit dem Benutzer führen, so muß es in der Lage sein, Eingaben vom Benutzer entgegenzunehmen und Ergebnisse als Ausgaben auch wieder anzuzeigen. Bisher hatten wir Eingaben an PROLOG immer durch die entsprechende Instantiierung von Variablen in der Anfrage gemacht, Ausgaben waren die Werte von Variablen nach abgeschlossenem Beweis. Für uns als Programmentwickler ist dies durchaus angemessen, da wir ja auch die interne Struktur der Prädikate, ihre Argumente etc. kennen. Wollen wir jedoch ein Programm schreiben, das von anderen ohne Kenntnis über den Programmaufbau benutzt wird, so ist dies unangebracht.
Zur Ein- und Ausgabe sind in PROLOG-Systemen verschiedene eingebaute Prädikate enthalten.
Wenn die Inferenzmaschine während des Beweises auf ein Eingabeprädikat stößt, so wird der Beweis unterbrochen und auf eine Eingabe des Benutzers gewartet. Ist diese erfolgt, so wird eine Variable mit der Eingabe als Wert instantiiert und der Beweis damit fortgesetzt. Aus der Sicht der Logik betrachtet, bedeutet eine Eingabe, daß der Benutzer nach dem Ergebnis eines Beweiszieles gefragt wird.
Stößt die Inferenzmaschine auf eine Ausgabeanweisung, so wird einfach das Argument des Prädikats auf dem Bildschirm ausgegeben und der Beweis fortgesetzt. Auf den Beweis selbst hat eine Ausgabe keinen Einfluß. Wichtig ist lediglich der Seiteneffekt, das Anzeigen der gewünschten Information. Bei der Ausgabe ist dieser Seiteneffekt aus Sicht der Logik harmlos, da das Ergebnis des Beweises nicht beeinflußt wird.
Die Ein- und Ausgabe kann in PROLOG auf zwei verschiedenen Ebenen erfolgen:

1. PROLOG-Terme: `read`, `write`
2. einzelne Zeichen: `get0`, `put`

11.1.1 Ein- und Ausgabe von Termen

Wenden wir uns zunächst der ersten Möglichkeit zu. Vom Benutzer wird bei `read` erwartet, daß die Eingabe genau der Syntax von PROLOG-Termen entspricht. Im Prinzip kann also alles eingegeben werden, was so auch im Programmtext stehen könnte. Außerdem muß jede Eingabe mit einem Punkt beendet werden. Die Ausgabe durch `write` erfolgt in einer Form, die von PROLOG auch wieder als Eingabe verstanden[1] würde.
Das einfachste Interaktionsmuster für den Dialog Benutzer–PROLOG ist:

```
interaktion :-
   read(X),
   do_something(X,Y),
   write(Y).
```

Beispiel 11.1 Mit diesem Schema können wir z.B. den Parser aus Kapitel 9 aufrufen. Damit ergibt sich das folgende Programm und der dargestellte Beispieldialog.

[1] Leider funktioniert das bei den meisten PROLOG-Systemen nicht 100%-ig

```
run :-
   read(Satz),
   s(Syntaxbaum,Satz,[]),
   write(Syntaxbaum).

?- run.
[john,loves,mary].
s(np(n(john)),vp(v(loves),np(n(mary))))
yes
```

Die Verwendung der Prädikate `read` und `write` erfordert vom Benutzer immer noch, daß die Syntax von PROLOG eingehalten wird. Dies wird man aber i.a. nicht voraussetzen können. Deshalb bietet PROLOG die zweite Gruppe von Prädikaten zur zeichenweisen Ein- und Ausgabe an. Es bleibt dann vollständig dem Programmierer überlassen, die Benutzereingabe zu analysieren und für die weitere Verarbeitung geeignet aufzubereiten.

11.1.2 Ein- und Ausgabe von Einzelzeichen

Durch das Prädikat `get0(X)` wird ein einzelnes Zeichen von der Tastatur eingelesen und die Variable `X` damit instantiiert. Durch `put` wird ein Zeichen auf dem Bildschirm ausgegeben. PROLOG repräsentiert dabei ein Zeichen durch seinen *ASCII*[2]*-Code*. Im ASCII-Code wird jedem Buchstaben, jeder Ziffer und verschiedenen Sonderzeichen jeweils eine ganze Zahl zwischen 0 und 127 zugeordnet. Der Buchstabe 'a' ist z.B. durch die Zahl 97 repräsentiert, das Leerzeichen durch 32. Aber keine Angst, wir müssen nicht den ASCII-Code auswendig lernen, um mit PROLOG arbeiten zu können. Wichtig ist nur, sich folgende Eigenschaften zu merken:

1. Die Ziffern 0 – 9 haben aufsteigende Codes ohne Abstände
2. Die Buchstaben a – z haben aufsteigende Codes ohne Abstände
3. Die Buchstaben A – Z haben aufsteigende Codes ohne Abstände

Die konkreten Zahlenwerte für die Codierung einzelner Buchstaben können wir von PROLOG erfahren, z.B. liefert `"0"` den Zahlenwert 48, der der Ziffer 0 zugeordnet ist.

Beispiel 11.2 Wir wollen ein Prädikat `get_digit(X)` schreiben, das solange Zeichen von der Tastatur einliest, bis eine Ziffer gefunden wird. Mit deren ASCII-Code wird `X` intanziiert.

```
get_digit(X) :-
   get0(X),
   X >= "0",
   X =< "9".
get_digit(X) :-
   get_digit(X).
?- get_digit(X).
abcd1
X = 49
yes
```

[2] American Standard Code for Information Interchange

Alle Ein- und Ausgabeprädikate sind *deterministisch*, d.h. sie sind genau einmal mit einer eindeutigen Lösung beweisbar. Wenn die Inferenzmaschine über Backtracking eine weitere Lösung sucht, so ergibt ein solches Prädikat immer *fail*. Auch wird der Seiteneffekt, nämlich das „Verbrauchen" von Eingabezeichen nicht rückgängig gemacht – wie sollte das auch möglich sein.
Dieses deterministische Verhalten erklärt die zweite Klausel im Prädikat `get_digit`. Wenn in der ersten Klausel keine Ziffer gelesen wurde, so ist diese Ziffer aus der Eingabe bereits „verbraucht", backtracking liefert aber auch keine weitere Lösung. Ohne eine zweite Klausel würde einfach das ganze Prädikat fehlschlagen, wenn beim ersten Versuch keine Ziffer eingegeben wird. Die zweite Klausel muß also die Eingabe wiederholen, falls die erste Klausel fehlschlug. Dazu ruft sie einfach wieder die erste Klausel auf, ohne selbst eine Eingabe zu verlangen. Dies geschieht solange, bis endlich eine Ziffer eingegeben wird.
Zwei weitere Ausgabeprädikate sind in der Praxis wichtig, weil sie zur Gestaltung von Ausgaben sehr nützlich sind.

`nl` steht für *new line* und führt dazu, daß die Ausgabe am Anfang der nächsten Zeile fortgesetzt wird.

`tab(N)` gibt `N` Leerzeichen aus.

Die Programmierung von Ein- und Ausgabeprädikaten kann eine trickreiche Angelegenheit sein, die einige Erfahrung erfordert. Die entstehenden Programme haben in der Regel wenig von der Ästhetik, die sonst PROLOG-Programmen nachgesagt wird. Leider lassen sich aber in realistischen Anwendungen diese Programmteile nicht vermeiden.
Wir empfehlen, bereits jetzt die Übung 11.1 durchzuführen, um ein erstes Gefühl für die Erstellung „prozeduraler" PROLOG-Programme zu bekommen.

11.2 Kontrolle

Die Prädikate, die wir in diesem Abschnitt vorstellen werden, erlauben uns, die Inferenzmaschine bei der Suche nach einem Beweis zu kontrollieren. Dies ist manchmal nötig, um Probleme bearbeiten zu können, die in „reinem" PROLOG nicht formulierbar sind. Oft kann man auch die Effizienz eines Programmes durch geschickte Steuerung der Inferenzmaschine erheblich verbessern.
Eine Möglichkeit zur Kontrolle haben wir bereits kennengelernt: Die geschickte Anordnung von Klauseln und von Unterbeweiszielen innerhalb einer Klausel. Bei rekursiven Prädikaten haben wir z.B. die Abbruchbedingung meist als erste Klausel aufgeschrieben, um sicherzustellen, daß PROLOG nicht in eine Endlosschleife gerät. Wir haben hier also ausgenützt, daß Klauseln immer in der Aufschreibreihenfolge zu beweisen versucht werden.
In der `append`-Version unseres ersten Parsers in Kapitel 8 war die Reihenfolge der Unterbeweisziele in jeder Klausel auch nicht zufällig. Es wurde immer erst mit `append` eine Zerlegung der Liste erzeugt und diese dann als die richtige zu beweisen versucht. Prinzipiell hätte man `append` auch am Ende jeder Klausel schreiben können:

```
s(S) :-
   np(NP),
   vp(VP),
   append(NP,VP,S).
```

Aber was bedeutet das für den Ablauf des Beweises? PROLOG muß nun alle möglichen Nominalphrasen und alle möglichen Verbalphrasen der Grammatik generieren, solange bis beide zusammen zufällig den Eingabesatz ergeben. Bei einer großen Grammatik kann das natürlich beliebig lange dauern. In der von

uns gewählten Reihenfolge dagegen wird versucht, möglichst viel vorhandene Information möglichst früh für den Beweis zur Verfügung zu stellen.
Implizit haben wir also bereits seit langem Einfluß auf den Ablauf eines Beweises durch PROLOG genommen. Im folgenden wollen wir einige Prädikate besprechen, mit denen *explizit* die Inferenzmaschine gesteuert wird.

11.2.1 Success und Failure

Manchmal ist es nützlich, der Inferenzmaschine zu sagen, daß die Suche nach einem Beweis eines Prädikats zwecklos ist. Oder man möchte gerne alle Lösungen zu einem Beweisziel ausgegeben haben; dann benötigt man eine Möglichkeit, explizit Backtracking auszulösen. Dazu stellt PROLOG das Prädikat `fail` zur Verfügung.

Beispiel 11.3 Man erhält alle Permutationen der Liste `[a,b,c]` von dem Prädikat `permute` aus Übung 7.1 durch folgende Anfrage:

```
?- permute([a,b,c],X), write(X), nl, fail.
[a,b,c]
[a,c,b]
[b,a,c]
[b,c,a]
[c,a,b]
[c,b,a]
no
```

Im Zusammenhang mit dem *cut* werden wir im nächsten Abschnitt nochmals auf `fail` zu sprechen kommen.
Das Prädikat `true` hat die gleichen Eigenschaften wie ein einzelner Fakt

```
true.
```

Auch gibt es keinerlei Seiteneffekte. Also ist `true` eigentlich völlig redundant. Daß es trotzdem vorhanden ist, hat seinen Grund darin, daß so eine einheitliche Sichtweise auf Fakten und Regeln möglich wird. Die beiden folgenden Klauseln sind abolut gleichbedeutend:

```
fakt(a,b,c).
fakt(a,b,c) :- true.
```

Bei manchem Programmen wird dem Programmierer viel Arbeit abgenommen, wenn er nicht zwischen Fakten und Regeln unterscheiden muß. So werden viele Spezialfälle im Programm vermieden und alles wird wesentlich einfacher und übersichtlicher. Deshalb wurde bereits bei der Semantikkonstruktion in Kapitel 10 von dieser Möglichkeit Gebrauch gemacht.
Das eingebaute Prädikat `repeat` könnten wir auch selber schreiben durch die folgende Definition:

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

Der Name `repeat` drückt das Verhalten des Prädikates aus: Wenn Backtracking in einer Klausel zu `repeat` zurückführt, so wird der Teil der Klausel nach dem `repeat` nochmals versucht zu beweisen, also wiederholt.

Beispiel 11.4 Mit `repeat` können wir das Prädikat `get_digit` aus dem vorigen Abschnitt etwas vereinfachen:

```
get_digit(X) :-
   repeat,
   get0(X),
   X >= "0",
   X =< "9".
```

Aber Vorsicht: Ein `repeat` in einer Klausel kann sehr leicht zu einer Endlosschleife führen:

```
endlos :-
   repeat,
   fail.
```

11.2.2 Cut

Das berühmt-berüchtigte *cut*-Prädikat, geschrieben einfach als Ausrufezeichen, ist das Lieblingswerkzeug vieler PROLOG-Hacker und gleichzeitig der Alptraum jedes Logikers, der sich mit PROLOG beschäftigt. Der *cut* stellt den extremsten Eingriff in den Beweisgang der Inferenzmaschine dar, der in PROLOG möglich ist. Intensive Verwendung von *cut* in einem Programm macht dieses fast immer völlig unverständlich und zeigt außerdem, daß der Programmierer die Prinzipien der Logikprogrammierung noch nicht richtig verinnerlicht hat. Soviel als Warnung im voraus. Doch was tut der *cut* nun eigentlich?
Die Grundidee des *cut* ist es, ganze Bereiche aus dem Suchraum der Inferenzmaschine bei einer Beweissuche herauszuschneiden, daher auch der Name. Wir wollen hier aber nicht versuchen, dies exakt und formal korrekt zu beschreiben. Stattdessen werden wir zuerst ein prinzipielles Beispiel für die Verwendung des *cut* geben und dieses erläutern. Im Anschluß daran zeigen wir einige typische Anwendungen. Wer damit noch nicht genug hat, der sei hiermit auf die einschlägige Literatur verwiesen.
Wenn die Inferenzmaschine bei einem Beweis auf einen *cut* stößt, so geschieht folgendes: Der Beweis von `cut` liefert *success* und alle Instantiierungen seit dem Beweisbeginn für das Beweisziel, in dem der *cut* auftritt, werden unveränderlich festgelegt. Mit anderen Worten, es wird kein alternativer Beweis versucht, für alles, was vor dem *cut* steht.

Beispiel 11.5 Zur Erläuterung der Funktionsweise von *cut* betrachten wir die beiden Prädikate `a1` und `a2`, die auf derselben Faktenbasis arbeiten. Sie sind identisch bis auf den *cut* in `a2`.

```
a1([B,C,D,E]) :-        a2([B,C,D,E]) :-
   b(B),                   b(B),
   c(C),                   c(C),
   d(D),                   !,
   e(E).                   d(D),
                           e(E).

b(b1).     b(b2).     c(c1).     c(c2).
d(d1).     d(d2).     e(e1).     e(e2).
```

Die Ergebnisse von `a1` und `a2` sind verschieden:

```
?- a1(X), write(X), fail.
[b1,c1,d1,e1][b1,c1,d1,e2][b1,c1,d2,e1][b1,c1,d2,e2]
[b1,c2,d1,e1][b1,c2,d1,e2][b1,c2,d2,e1][b1,c2,d2,e2]
[b2,c1,d1,e1][b2,c1,d1,e2][b2,c1,d2,e1][b2,c1,d2,e2]
[b2,c2,d1,e1][b2,c2,d1,e2][b2,c2,d2,e1][b2,c2,d2,e2]
no
?- a2(X), write(X), fail.
[b1,c1,d1,e1][b1,c1,d1,e2][b1,c1,d2,e1][b1,c1,d2,e2]
no
```

Was hat die Verwendung des *cut* in unserem Beispiel verändert? Entsprechend der Beschreibung der Funktionsweise des *cut*, erfolgt eine Festschreibung der Variableninstantiierungen zum Zeitpunkt der Ausführung des *cut*. Für `a2` heißt dies konkret: `B` ist mit `b1` und `C` mit `c1` instantiiert. Beim Backtracking ergibt das Prädikat `a2` sofort insgesamt *fail*, wenn der *cut* erreicht wird. Die anderen Lösungen für die Unterbeweisziele `b(B)` und `c(C)` spielen keine Rolle mehr. Für `d(D)` und `e(E)` werden dagegen nach wie vor alle möglichen Lösungen bestimmt. Hinter dem *cut* ist Backtracking in der üblichen Weise möglich. Dies erklärt die unterschiedlichen Resultate von `a1` und `a2`.
Wir können uns die Wirkungsweise des *cut* veranschaulichen, wenn wir uns eine Tür vorstellen, die nur auf einer Seite eine Klinke besitzt. Wenn wir durch diese Tür hindurchgehen und sie hinter uns ins Schloß gefallen ist, so gibt es kein Zurück. Genauso ergeht es der Inferenzmaschine mit dem *cut*. Wenn beim Beweis des Prädikates `a2` einmal der *cut* ausgeführt wurde, so gibt es kein Zurück mehr. Stellt die Inferenzmaschine später fest, das Backtracking nötig ist und gelangt an den *cut* zurück, so bleibt nur übrig, insgesamt aufzugeben und den Beweis für `a2` mit *fail* abzuschließen.
Wie bereits erwähnt, ist die Verwendung des *cut* möglichst zu vermeiden. Es gibt jedoch einige Fälle, in denen das nicht möglich ist.
Eine dieser Anwendungen, für die der *cut* unbedingt benötigt wird, ist die Beschreibung von *Negation*; d.h. man möchte ausdrücken, daß ein Prädikat unter bestimmten Bedingungen *nicht* erfüllbar ist. Hierzu folgendes Schema:

`p` ist *unerfüllbar* , falls `q` gilt
$\Longrightarrow$
```
p :- q, !, fail.
```

Hier sehen wir eine der häufigsten Anwendungen des `fail`-Prädikats, nämlich die *cut–fail–Kombination*. Die Wirkung in unserem Fall ist einfach zu beschreiben: Sobald `q` bewiesen ist, werden mit *cut* alle weiteren Beweisversuche unterbunden und `fail` löst Backtracking aus. Zurücksetzen über den *cut* ist aber nicht möglich; somit ergibt `p` insgesamt *fail*.
Somit haben wir genau die Bedingung realisiert, daß `p` unerfüllbar ist, falls `q` gilt.

Beispiel 11.6 Auch am Finanzamt geht die Hochtechnologie nicht spurlos vorüber. Ein eifriger Beamter beschließt nach dem Besuch eines Volkshochschulkurses „Grundkurs 'PROLOG' für Finanzbeamte“, ein Programm zu entwickeln, das die Steuerschuld eines Bürgers berechnet. Teil dieses Programms ist ein Prädikat `steuerpflichtig(Person)`, das für eine gegebene Person überprüft, ob überhaupt Steuepflicht vorliegt. In diesem Prädikat werden z.B. die Bedingungen berücksichtigt, daß Hausbesitzer immer steuerpflichtig sind, daß Ausländer – unter bestimmten zusätzlichen Bedingungen – in Deutschland steuerfrei sind, Steuerpflicht erst ab einem gewissen Mindesteinkommen vorliegt, ...

```
steuerpflichtig(Person) :-
   hausbesitzer(Person).
```

```
steuerpflichtig(Person) :-
   auslaender(Person),
   zusatzbedingungen(Person),
   !, fail.
steuerpflichtig(Person) :-
   einkommen(Person,Einkommen),
   mindesteinkommen(Mindesteinkommen),
   Einkommen < Mindesteinkommen,
   !, fail.
...
```

Eine weitere zentrale Anwendung des *cut* ist es, für gewisse Prädikate *Eindeutigkeit* zu fordern. Wir wollen also Prädikate, die höchstens eine Lösung ergeben und beim Versuch, sie durch Backtracking ein weiteres Mal zu beweisen, scheitern. Solche *deterministischen Prädikate* haben wir bereits bei der Ein- und Ausgabe kennengelernt.
Wir erhalten hier folgendes Schema:

p ist deterministisch
$\Longrightarrow$

```
p :- ..., !.          % ALLE Klauseln
p :- ..., !.          % werden mit '!'
...                   % abgeschlossen
```

Falls sich nun eine der Klauseln für p beweisen läßt, so wird die Inferenzmaschine auf diese Lösung für p festgelegt, und es kann keine weitere Klausel über Backtracking erreicht werden. Somit ist höchstens eine Lösung für p möglich.

11.3 Meta-Programmierung

Eine besondere Eigenschaft von PROLOG ist es, daß wir nicht nur logische Formeln als Programme angeben, sondern auch *Aussagen über logische Formeln* machen können. Wenn wir dies tun, so befinden wir uns damit in einer *meta-logischen* Ebene. Wir alle kennen *Meta-Ebenen* aus unserem Alltag. Wenn z.B. während einer Diskussion plötzlich über die Form der Diskussion und nicht mehr über das Thema gesprochen wird, haben wir eine solche Meta-Ebene betreten. Nur macht sich eigentlich niemand einen solchen Wechsel wirklich bewußt; wir können ganz selbstverständlich damit umgehen. Bei der PROLOG-Programmierung müssen wir uns allerdings stets im klaren darüber sein, welcher Art die Aussagen sind, die wir im Programm aufschreiben. Verlassen wir die Ebene der Hornklausellogik, um Aussagen über Formeln zu machen, sprechen wir von *Meta-Programmierung*.

11.3.1 Klassifikation von Termen

Wie wir bereits wissen, können Terme in PROLOG Variablen, Atome, Zahlen oder komplexe Strukturen sein. Manchmal ist es für uns interessant, im Programm den Typ eines Terms zu erfahren. Dazu stellt uns PROLOG folgende eingebaute Prädikate zur Verfügung:

`var(X)` gilt, falls `X` eine uninstantiierte Variable ist

`nonvar(X)` gilt, falls `X` keine uninstantiierte Variable ist; d.h. `X` ist ein Atom, eine Zahl oder eine komplexe Struktur

`atom(X)` gilt, falls `X` ein Atom ist

`integer(X)` gilt, falls `X` eine Zahl ist

`atomic(X)` gilt, falls `X` ein Atom oder eine Zahl ist

Die folgenden Beispiele sollen das Verhalten dieser Prädikate verdeutlichen:

```
?- var(X).
yes
?- var(hallo).
no
?- atom(X).
no
?- atom(hallo).
yes
?- atom(4711).      % Zahlen sind keine Atome!
no
?- atom('4711').    % koennen aber welche werden!
yes
?- integer(4711).
yes
?- integer('4711'). % Atome sind keine Zahlen!
no
?- atomic(4711).
yes
?- atomic('4711').
yes
```

11.3.2 `=..`

Wenn wir normalerweise in einem Programm Strukturen verwenden wollen, so tun wir dies, indem wir sie einfach benutzen. Ein Prädikat, das verschiedene Strukturen behandeln soll, enthält einfach für jede einzelne eine eigene Klausel mit der jeweiligen Struktur als Argument. Was aber tun, wenn wir in *einem* Prädikat *alle* möglichen Strukturen behandeln wollen? Denken wir z.B. an den „pretty printer“ aus Übung 11.1. Wenn wir dieses Programm so erweitern wollen, daß es beliebige Strukturen ausgibt, so benötigen wir eine Möglichkeit, auf Funktor und Argumente von Termen zuzugreifen, die wir zum Zeitpunkt der Programmerstellung nicht kennen.
Zu diesem Zweck gibt es ein eingebautes Prädikat `=..`, aus historischen Gründen *univ* genannt.

`X =.. L` bedeutet: `X` ist ein Term, dessen Funktor das erste Element der Liste `L` ist und dessen Argumente die übrigen Elemente der Liste sind

Die folgenden Beispiele illustrieren die Möglichkeiten von `=..`:

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

```
yes
?- a(X,c,d) =.. [Y,b|Z].
X = b
Y = a
Z = [c,d]
yes
?- X =.. [Y,b,c,d].
no
?- X =.. [a|_].
no
```

Die letzten beiden Fälle zeigen eine wichtige Einschränkung: Zur Zeit der Ausführung von `=..` muß der Funktor und die Anzahl der Argumente des Terms bekannt sein. PROLOG ist nicht in der Lage, Terme mit variablem Funktor oder variabler Stelligkeit zu behandeln. Die Reaktion auf eine solche falsche Verwendung von `=..` ist bei verschiedenen PROLOG-System unterschiedlich. Manchmal ergibt sich lediglich ein *fail* – wie in unserem Beispiel –, manchmal auch eine Systemfehlermeldung.

Beispiel 11.7 Es soll ein Prädikat `listvar`[3] entwickelt werden, das alle in einer gegebenen Struktur enthaltenen Variablen in einer Liste aufsammelt.

```
listvar(Struct,ListOfVariables) :-
   listvar(Struct,[],ListOfVariables),
   !.

listvar(Var,In,[Var|In]) :-
   var(Var) .
listvar(Atomic,In,In) :-
   atomic(Atomic) .
listvar(Complex,In,Out) :-
   Complex =.. [Functor|Args],
   listvars(Args,In,Out).

listvars([],In,In).
listvars([Arg|Args],In,Out) :-
   listvar(Arg,In,Out1),
   listvars(Args,Out1,Out).

?- listvar(a(X,Y),L).
X = _005D
Y = _006D
L = [_005D,_006D]
yes
```

Bei der Implementierung von `listvar` wurde ein gängiger Trick zum Aufsammeln von Ergebnissen bei rekursiven Prädikaten benutzt. Dieser Trick ist so häufig in der Praxis zu finden, daß er hier einmal erläutert werden soll. Das zweistellige Prädikat `listvar` ruft als einzige Aktion ein dreistelliges Prädikat `listvar` auf. Die Argumente haben dabei folgende Bedeutung:

[3] Wir werden dieses Prädikat im nächsten Kapitel wieder benötigen

1. die aktuell zu bearbeitende Struktur
2. die Liste der bereits gefundenen Variablen
3. die Liste der Variablen nach vollständiger Abarbeitung dieses Aufrufes von `listvar`

Das zweite Argument von `listvar` dient dem Aufsammeln aller bislang gefundenen Ergebnisse. Man spricht bei einer solchen Konstruktion auch von einem *Akkumulator*. Aus der Bedeutung der Argumente ergeben sich die Bedingungen, die in den einzelnen Klauseln ausgedrückt sind:

1. die aktuelle Struktur ist eine Variable: nimm die gefundene Variable zur Liste der bereits bekannten Variablen hinzu
2. die aktuelle Struktur ist atomar: die Liste der bereits gefundenen Variablen bleibt unverändert
3. die aktuelle Struktur ist ein komplexer Term: zerlege den Term mit `=..` in Funktor und Liste der Argumente und füge alle Variablen, die in den Argumenten enthalten sind, zur Eingabeliste hinzu

Das Prädikat `listvars` benutzt einfach `listvar`, um für jedes Argument des komplexen Terms die Variablen zur Eingabeliste hinzuzufügen.

11.3.3 Manipulation der Datenbasis

Wir wissen bereits, daß in PROLOG Prädikate und Terme absolut identisch aufgebaut sind. Ein Fakt ist nichts anderes als ein Term mit dem Prädikatsnamen als Funktor. Eine Regel ist ein Term mit dem Funktor `:-`, der Kopf ist das erste Argument und der Rumpf das zweite. Für bestimmte Funktoren wie z.B. `:-`, `=..`, ... erlaubt uns PROLOG lediglich eine vereinfachte Schreibweise. Solche speziellen Funktoren heißen *Operatoren*.

Beispiel 11.8 Die dritte Klausel aus Beispiel 11.3.2 müßte ohne Verwendung von Operatoren die folgende Form erhalten:

```
:-(listvar(Complex,In,Out),
   ',(=..(Complex,[Functor|Args]),
       listvars(Args,In,Out)
     )
  )
```

Der PROLOG-Interpreter weiß über die spezielle Bedeutung der Funktoren `:-` und `,` Bescheid und benutzt sie zur Steuerung der Inferenzmaschine.
Wenn nun aber Terme und Prädikate genau dieselbe Form haben, was liegt dann näher als Prädikate ebenfalls wie Terme als Daten zu behandeln, sie zu kontruieren oder zu verändern? Uns fehlen lediglich noch Möglichkeiten, so konstruierte Prädikate in die Datenbasis einzutragen oder sie der Inferenzmaschine als Beweisziel zu übergeben.
Zur Manipulation der Datenbasis stellt PROLOG die folgenden eingebauten Prädikate zur Verfügung:

`assert(X)` trägt den Term `X` in die Datenbasis ein.

`retract(X)` löscht die erste Klausel der Datenbasis, die mit `X` als Term unifizierbar ist. Über Backtracking werden weitere passende Einträge gelöscht.

Sehr wichtig ist es, zu beachten, daß die Seiteneffekte von `assert` und `retract`, also die Veränderungen in der Datenbasis, beim Backtracking nicht zurückgenommen werden.

Beispiel 11.9 Das Prädikat `retractall(X)` soll alle Fakten, die mit `X` als Term unifizierbar sind und alle Regeln, deren Kopf mit `X` unifizierbar ist, aus der Datenbasis löschen. Dazu nützen wir aus, daß `retract` beim Backtracking jeweils einen weiteren Eintrag löscht, solange bis keine weitere Möglichkeit besteht.

```
retractall(X) :-
   retract(X),
   fail.
retractall(X) :-
   retract((X:-Y)),
   fail.
retractall(X).
```

Die erste Klausel von `retractall` löscht alle passenden Fakten, die zweite alle passenden Regeln und die dritte sorgt dafür, daß `retractall` immer mit *success* beendet wird.

11.3.4 Call

Im vorigen Abschnitt haben wir die Datenbasis manipuliert, die die Inferenzmaschine für ihre Beweise benützt. Jetzt wollen wir das eingebaute Prädikat `call` vorstellen, mit dem wir der Inferenzmaschine einen Term als Beweisziel übergeben können.

Beispiel 11.10 Das Prädikat `unique(P)` soll die erste Lösung des Prädikats `P` liefern. Backtracking ergibt dann sofort *fail*.

```
unique(P) :-
   call(P),
   !.
```

Die Realisierung ist sehr einfach: es wird lediglich mit `call` die Inferenzmaschine angewiesen, `P` zu beweisen; geht dies gut, folgt sofort der *cut*, wodurch weitere Lösungen für `unique` abgeschnitten werden.
Das Prädikat `unique` ist ein gutes Beispiel dafür, wie in einem Programm die Verwendung des *cut* vermieden werden kann. Statt im Prädikat `P` den *cut* einzusetzen, realisiert man eine *cut-freie* Version von `P` und benützt beim Aufruf von `P` das Prädikat `unique`, um auszudrücken, daß `P` hier deterministisch ist. Dies hat außer der Reduzierung der Anzahl von *cut* im Programm zwei weitere schöne Effekte:

1. der Leser des Programms sieht sofort beim Aufruf von `P`, daß hier ein deterministisches Prädikat vorliegt
2. das Prädikat `P` kann allgemeiner geschrieben werden und dadurch eher auch in einem anderen Programm wiederverwendet werden

Im Normalfall liefert ein Beweis in PROLOG immer eine Lösung zu einer Anfrage und dann über Backtracking evtl. weitere. In manchen Fällen ist es aber nützlich, statt einer Lösung sofort eine Liste mit allen möglichen Lösungen für ein Prädikat zu erhalten. Dazu stellt PROLOG zwei eingebaute Prädikate zur Verfügung:

`bagof(X,P,L)` ergibt in `L` die Liste aller Lösungen für `X`, die für das Prädikat `P` möglich sind

`setof(X,P,L)` arbeitet wie `bagof`, aber doppelte Lösungen werden aus der Liste `L` entfernt und außerdem wird `L` in der Regel sortiert

Damit diese Prädikate sinnvolle Lösungen ergeben, muß natürlich die Variable `X` in dem Prädikat `P` vorkommen.

Beispiel 11.11 Die Prädikat `member(X,L)` versucht `X` mit einem Element der Liste `L` zu unifizieren. Über Backtracking erhält man nacheinander alle Lösungen. Betrachten wir damit folgende Beispiele von `bagof` und `setof`:

```
?- bagof(X,member(X,[b,a,c,b]),L).
L = [b,a,c,b]
yes
?- setof(X,member(X,[b,a,c,b]),L).
L = [a,b,c]
yes
```

Mit den Möglichkeiten zur Meta-Programmierung, die PROLOG uns bietet, können wir leicht eine eigene Variante von `bagof` realisieren.

Beispiel 11.12 Das Prädikat `my_bagof(X,P,L)` soll alle Ergebnisse von `P` für die Variable `X` in der Liste `L` ergeben.

```
my_bagof(X,P,L) :-
   compute_all(X,P),
   list_all(L),
   !.

compute_all(X,P) :-
   call(P),
   assert(bagof_solution(X)),
   fail.
compute_all(X,P).

list_all([X|L]) :-
   retract(bagof_solution(X)),
   list_all(L).
list_all([]).
```

Mit `compute_all` werden über Backtracking alle Lösungen des Prädikats `P` erzeugt und der jeweilige Wert für `X` mit `assert` in die Datenbasis eingetragen. Dabei muß `X` natürlich eine in `P` auftretende Variable sein, sonst ist dies (meist) sinnlos. Mit `list_all` werden die Lösungen nacheinander wieder aus der Datenbasis gelöscht und dabei in einer Liste aufgesammelt.

11.3.5 Not

Im Zusammenhang mit dem *cut* haben wir bereits eine Form der Negation von Prädikaten gesehen. Durch Verwendung von `call` läßt sich Negation auch noch etwas universeller ausdrücken. Wir können uns ein Prädikat `not(P)` schreiben, das gerade dann beweisbar ist, wenn `P` nicht beweisbar ist:

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

In der ersten Klausel von `not` wird versucht, `P` zu beweisen. Falls dies gelingt, folgt sofort eine *cut-fail*-Kombination, wodurch `not(P)` *fail* ergibt. Falls `P` nicht beweisbar ist, erhält man durch die zweite Klausel dann *success*.

In PROLOG ist meist auch ein eingebautes Prädikat für `not` vorhanden, das als `\+` geschrieben wird.

Leider wird durch `not` keine volle Negation im Sinne der Logik ausgedrückt. Was PROLOG realisiert ist die sogenannte *negation as failure*. Dies führt manchmal zu etwas unintuitiven Ergebnissen, wie es das folgende Beispiel illustriert.

Beispiel 11.13 Das Prädikat `unverheirater_student(X)` soll überprüfen, ob eine gegebene Person ein unverheirater Student ist:

```
unverheirateter_student(X) :-
   \+ verheiratet(X),
   student(X).
student(peter).
verheiratet(hans).

?- unverheirateter_student(peter).
yes
?- unverheirateter_student(X).
no
```

Das Ergebnis der ersten Anfrage ist völlig klar: Peter ist Student und es gibt keinen Beweis dafür, daß er verheiratet ist. Also folgert PROLOG messerscharf, daß Peter ein unverheirateter Student ist. Soweit funktioniert das Prädikat also. Die zweite Anfrage, bei der das Prädikat mit einer Variablen aufgerufen wird, soll die Frage beantworten, wer ein unverheirateter Student ist. Eigentlich würden wir erwarten, daß als Ergebnis Peter bestimmt wird. Dies funktioniert leider nicht. PROLOG versucht nämlich zunächst zu zeigen, daß es niemanden gibt, der verheiratet ist. Dies gelingt aber nicht und schon scheitert das gesamte Prädikat. Würde man die beiden Unterbeweisziele im Rumpf vertauschen, erhielte man das gewünschte Ergebnis.

Dies liegt daran, daß dann im negierten Beweisziel keine uninstantiierte Variable mehr auftritt. Allgemein gilt:

> Negation in PROLOG liefert nur dann immer ein korrektes Ergebnis, wenn in dem negierten Beweisziel keine unintantiierten Variablen vorkommen.

Dies muß der Verwendung von Negation immer berücksichtigt werden und evtl. die Reihenfolge der Unterbeweisziele in einer Klausel entsprechend gewählt werden.

11.4 Sonstiges

Heutige PROLOG-Systeme bieten in der Regel eine Vielzahl eingebauter Prädikate, die alle für die verschiedensten Aufgaben mehr oder weniger nützlich sind. Es würde aber zu weit führen, hier auch nur alle jene zu besprechen, die sogar in eigentlich allen Systemen vorhanden sind. Wir wollen uns auf eine Auswahl beschränken, die ein Minimum darstellt, um in der Praxis realistische Programme zu entwickeln. Dazu fehlen uns noch einige wenige Prädikate, die in diesem Abschnitt zusammengefaßt sind.

11.4.1 Arithmetische Ausdrücke

In manchen Fällen genügt es nicht, durch Unifikation einfach imme komplexere Terme aufzubauen, sondern man möchte insbesondere für arithmetische Ausdrücke auch gerne mal ihren Wert berechnen können. Dazu bietet PROLOG einige arithmetische Operatoren, wie z.B. `+`, `-`, `*`, `/` und `mod`, und einen Evaluationsoperator `is` an.

`X is Y` gilt, falls `Y` ein arithmetischer Ausdruck ist, also nur arithmetische Operatoren, Zahlen, Variablen und Klammern enthält, in dem alle Variablen mit Zahlen instantiiert sind und X mit dem Ergebnis der Berechnung – *Evaluation* – von `Y` unifiziert wird.

Außerdem besteht die Möglichkeit, die Werte von arithmetischen Ausdrücken ihrer Größe nach zu vergleichen.

`X < Y` gilt, falls die Evaluation von `X` eine Wert ergibt, der *kleiner* als der Wert der Evaluation von `Y` ist

`X =< Y` entsprechend mit *kleiner oder gleich*

`X >= Y` entsprechend mit *größer oder gleich*

`X > Y` entsprechend mit *größer*

11.4.2 Strings

Unter einem *String* versteht man eine Folge von Zeichen, die man bearbeiten können möchte. Im Gegensatz zu Atomen muß bei Strings auch der Zugriff auf Teile möglich sein. Die Verarbeitung von Zeichenketten stellt eine sehr häufig auftretende Aufgabe für den Programmierer dar. In den meisten Programmiersprachen gibt es deshalb eine spezielle Datenstruktur für Strings. PROLOG dagegen kennt keine solche Spezialstruktur. Stattdessen verwendet man einfach *Listen von ASCII-Codes*. Dies hat den Vorteil, daß alle Möglichkeiten der Listenverarbeitung für Strings verwendet werden können, was PROLOG eine sehr große Flexibilität bei der Stringverarbeitung gibt. Außerdem haben die meisten Anwendungen mit Ein- und Ausgabe zu tun. Und hier verwendet PROLOG ebenfalls ASCII-Codes.
Da aber Stringverarbeitung sehr häufig vorkommt und Listen von ASCII-Codes für den Programmierer sehr unleserlich sind, bietet PROLOG eine spezielle Schreibweise für Strings an. Statt `[97,110,116,111,110]` darf einfach auch `"anton"` geschrieben werden. PROLOG übersetzt dies zur Verarbeitung in die entsprechende Liste.

Beispiel 11.14 Eines der wichtigsten Prädikate der Stringverarbeitung ist `concatenate(S1,S2,S)`, das Verketten zweier Strings `S1` und `S2` zu einem Gesamtstring `S`. Durch die Darstellung als Listen ist dies trivial:

```
concatenate(S1,S2,S) :-
   append(S1,S2,S).
```

Beispiel 11.15 Eine weitere wichtige Aufgabe ist es, festzustellen ob ein gegebener String einen zweiten String als Teil enthält. Dazu schreiben wir das Prädikat `substring(S,SubS,N)`, wobei `N` die Position des Teilstrings `SubS` in `S` ist.

```
substring(S,SubS,N) :-
   append(Prefix,Rest,S),
   append(SubS,Suffix,Rest),
   length(Prefix,N).
```

Die Realisierung von `substring` verwendet `append`, um den String `S` in drei Teile so zu zerlegen, daß `SubS` gerade den mittleren Teil darstellt[4]. Mit `length` wird die Länge des Teiles von `S` bestimmt, der vor `SubS` liegt, was gerade der Position von `SubS` in `S` entspricht, falls man die Positionszählung bei Null beginnt.

Beispiel 11.16 Ein Prädikat zur Berechnung der Länge eines Strings können wir leicht definieren durch:

```
length([],0).
length([First|Rest],N) :-
   length(Rest,M),
   N is M+1.
```

Gelegentlich möchte man Atome in Strings verwandeln oder umgekehrt. Dazu gibt es in PROLOG das eingebaute Prädikat `name`.

`name(A,S)` gilt, falls `A` ein Atom und `S` ein String ist, sodaß die einzelne Elemente der Liste `S` gerade den ASCII-Codes der in `A` enthaltenen Buchstaben entsprechen

Die Parser für unsere bisherigen Grammatiken erwarten Sätze immer als Listen von Atomen. Wenn wir allerdings einen Satz mit `read_sentence` einlesen, so erhalten wir Listen von Strings. Um unsere Parser weiterverwenden zu können, müssen wir eine Konvertierung durchführen. Dazu schreiben wir ein Prädikat `convert`.

Beispiel 11.17 Das Prädikat `convert(Atoms,Strings)` soll eine Liste von Atomen auf eine entsprechende Liste von Strings abbilden und umgekehrt.

```
convert([],[]).
convert([Atom|Atoms],[String|Strings]) :-
   name(Atom,String),
   convert(Atoms,Strings).
```

[4]Natürlich ist diese Realisierung sehr ineffizient. Ein gezielt arbeitendes Suchvervafhren wäre besser als das blinde Probieren von Zerlegungen durch `append`.

11.5 Beispiel

Wir wollen ein Prädikat `read_sentence(X)` entwickeln, das einen Satz von der Tastatur einliest und ihn als eine Liste von Strings in der Variablen `X` ablegt. Ein Beispiel zur Illustration:

```
?- read_sentence(X).
dies ist ein satz.
X = ["dies","ist","ein","satz","."]
```

Die Ausgabe wurde etwas aufbereitet, weil sie tatsächlich sehr unübersichtlich aussieht, da die Strings von PROLOG als Listen von ASCII-Codes ausgegeben werden.

11.5.1 Entwurf

Die gewünschte Ausgabe von `read_sentence` steht bereits fest: eine Liste von Wörtern, die durch Strings dargestellt sind. Strings wiederum sind Listen von ASCII-Codes.
Die erste Grobskizze für das Programm sieht also so aus:

- Einlesen eines Satzes bedeutet Einlesen einer Liste von Wörtern
- Einlesen einer Liste von Wörtern bedeutet Einlesen eines Wortes gefolgt von einer Liste von Wörtern
- Einlesen eines Wortes bedeutet Einlesen eines einzelnen Zeichens gefolgt von einer Liste von Zeichen

Über die Tastatur werden einzelne Zeichen eingegeben. Wir erhalten also lediglich eine Folge von ASCII-Codes. Wie kann dieser Folge die gewünschte Struktur gegeben werden? Dazu halten wir folgendes fest:

- die Eingabe eines Punktes oder Fragezeichens beendet den Satz
- ein Wort ist eine zusammenhängende Kette von Zeichen, die keine Leerzeichen sind. Oder anders formuliert: ein Wortende erkennt man durch das Einlesen eines Leerzeichens

Daraus ergeben sich die Abbruchbedingungen für das Einlesen von Wortlisten und Zeichenlisten.
An dieser Stelle ist es sehr wichtig, sich daran zu erinnern, daß sämtliche Ein- und Ausgabeprädikate deterministisch sind und ihr Seiteneffekt beim Backtracking nicht rückgängig gemacht werden kann. Wenn wir also beim Einlesen eines Leerzeichens dieses als Wortende behandeln wollen, so dürfen wir in der entsprechenden Klausel nicht einfach ein Zeichen einlesen und prüfen, ob ein Leerzeichen vorliegt. Ein solches Vorgehen würde, falls es sich um kein Leerzeichen handelt, dieses auf Nimmerwiedersehen aus der Eingabe verbrauchen, sodaß eine andere Klausel, die eigentlich diesen Fall behandeln soll, dieses Zeichen gar nicht mehr sieht.
Wir müssen also sicherstellen, daß ein Zeichen nur einmal eingelesen und dann auch wirklich verarbeitet wird, bevor ein weiteres Zeichen eingelesen wird.

11.5.2 Realisierung

Hier ist nun also das vollständige Programm:

```
% read_sentence(Sentence)
%
% Zweck:        Einlesen eines Satzes von der Tastatur
% Argumente:    Sentence
%                 bei CALL: uninstantiierte Variable
%                 bei EXIT: LISTE von STRINGS
% Beispiel:     ?- read_sentence(X).
%              > dies ist ein satz.
%                X = ["dies","ist","ein","satz","."]
% Hinweise:     - zwei aufeinanderfolgende Woerter
%                 muessen durch genau ein Leerzeichen
%                 getrennt sein
%               - die Eingabe eines Satzes wird durch
%                 einen Punkt oder ein Fragezeichen
%                 unmittelbar nach dem letzten Wort
%                 beendet. Insbesondere darf vorher
%                 KEIN Leerzeichen stehen
%               - innerhalb des Satzes duerfen keine
%                 Satzzeichen verwendet werden

read_sentence(ListOfWords) :-
   get0(Char),
   read_wordlist(Char,ListOfWords),
   !.

% read_wordlist(Char,WordList)
%    liest eine Liste von W"ortern, deren erstes
%    Zeichen Char ist

read_wordlist(Char,["."]) :-
   Char is ".".
read_wordlist(Char,["?"]) :-
   Char is "?".
read_wordlist(Char,[FirstWord|RestOfWordList]) :-
   read_word(Char,FirstWord,FirstCharOfNextWord),
   read_wordlist(FirstCharOfNextWord,RestOfWordList).

% read_word(Char,Word,FirstCharOfNextWord)
%    liest eine Zeichenliste Word, deren erstes
%    Zeichen Char ist und gibt auf als 3. Argument
%    das erste Zeichen des naechsten Wortes zurueck
read_word(Char,[],NextChar) :-
   Char is " ",
   get0(NextChar).
read_word(Char,[],Char) :-
```

```
   Char is ".".
read_word(Char,[],Char) :-
   Char is "?".
read_word(Char,[Char|RestOfWord],FirstCharOfNextWord) :-
   get0(NextChar),
   read_word(NextChar,RestOfWord,FirstCharOfNextWord).
```

11.6 Zusammenfassung

Wenn Sie in die Verlegenheit kommen, einmal ein größeres Programm zu entwickeln, dann besorgen Sie sich das Handbuch Ihres PROLOG-Systems und verschaffen sich einen Überblick über die dort vorhandenen Prädikate. Aber ein wichtiger Tip ist angebracht: Halten Sie sich mit der Verwendung spezieller Konstruktionen, die Ihr System erlaubt zurück. Wenn Sie Ihr Programm später einmal auf einem anderen PROLOG-System verwenden wollen, ist die Frustration sonst meist sehr groß. Verwenden Sie also nach Möglichkeit nur die hier und darüber hinaus vielleicht gerade noch die in [DECsystem-10 PROLOG User's Manual] beschriebenen Prädikate. Diese werden Sie in fast allen PROLOG-Systemen finden.
Die Implementierung von Prädikaten zur Ein- und Ausgabe ist in PROLOG oft eine extrem kniffelige Angelegenheit. Um dem Programmierer diese lästigen und zeitraubenden Arbeiten zu erleichtern, bieten viele Systeme zusätzlich weitere eingebaute Prädikate an. Lesen Sie doch einfach mal im Handbuch zu Ihrem System nach, bevor Sie graue Haare beim Schreiben einer Benutzerschnittstelle für eines Ihrer Programme bekommen!
Wenn Sie Spezialitäten Ihres PROLOG-Systems ausnützen, sollten Sie deren Gebrauch immer gut dokumentieren und außerdem darauf achten, daß sie nur innerhalb weniger Prädikate vorkommen. Der Großteil sollte immer in Standard-PROLOG geschrieben sein.

11.7 Übungen

Übung 11.1 Schreiben Sie ein Prädikat, das eine Liste übersichtlich ausgibt, z.B. auf jeder Zeile ein Element. Ein Programm, das bestimmte Datenstrukturen in einer speziellen, gut lesbaren Form ausgibt, nennen wir *pretty printer*.

Übung 11.2 Erweitern Sie das Prädikat aus Übung 11.1 so, daß Listenelemente, die wiederum Listen sind, genauso aufbereitet werden, aber durch Einrückung gekennzeichnet sind.
Beispiel:

```
?- pp_list([a,[b,c],d]).
a
  b
  c
d
yes
```

Übung 11.3 Das folgende Programm zum Umdrehen einer Liste wird oft auch „naive reverse" genannt, weil es unmittelbar die Idee, wie eine Liste umzudrehen ist, in ein rekursives Prädikat umsetzt. Leider benötigt es einen enormen Rechenaufwand.

```
reverse([],[]).
reverse([First|Rest],ReversedList) :-
```

```
reverse(Rest,ReversedRest),
append([First],ReversedRest,ReversedList).
```

Benutzen Sie den im Abschnitt 11.3.2 beschriebenen Trick, um ein effizientes Programm zum Umdrehen einer Liste zu schreiben.

Übung 11.4 Erweitern Sie für das Problem `wegsuche` aus Kapitel 5 die Menge der Fakten über mögliche Verbindungen. Lassen Sie zu, daß einzelne Verbindungen auch in der Gegenrichtung begangen werden können. Untersuchen Sie das Ablaufverahlten von `?- wegsuche(Start,Ende,Weg).`. Welches Problem ist enstanden? Wie könnte dem abgeholfen werden? Verbessern Sie das Programm entsprechend.

Übung 11.5 Erweitern Sie das Prädikat `read_sentence` aus Abschnitt 11.5 so, daß

1. zwischen Wörtern beliebig viele Leerzeichen stehen können
2. Satzzeichen wie Komma, Semikolon, ... innerhalb eines Satzes verwendet werden dürfen. Satzzeichen sollen als eigenständige Wörter behandelt werden
3. vor dem Satzzeichen am Satzende beliebig viele Leerzeichen stehen können

Kapitel 12

Ein kleines Dialogsystem

In diesem Kapitel wollen wir ein kleines Dialogsystem entwickeln und dabei die Früchte der Arbeit der letzten Kapitel ernten. Es soll gezeigt werden, wie die einzelnen Teile, die wir bereits kennen, sinnvoll und einfach zu einem größeren Ganzen zusammengesteckt werden können.
Unser Dialogsystem soll z.B. zu folgendem philosophischen Dialog in der Lage sein:

```
?- dialog.
> ist sokrates sterblich?
nein
> alle menschen sind sterblich.
ok
> sokrates ist ein mensch.
ok
> ist sokrates sterblich?
ja
> wer ist sterblich?
sokrates
> ist xanthippe sterblich?
nein
> xanthippe ist ein mensch.
ok
> ist xanthippe sterblich?
ja
> wer ist sterblich?
sokrates xanthippe
> ade.
ade
```

Die Aufgabe besteht also darin, Sätze einzulesen, zu erkennen, ob es sich um Aussagesätze oder Fragen handelt, bei Aussagen deren Bedeutung abzuspeichern und Fragen entsprechend dem vorhandenen Wissen zu beantworten.
Wir können sofort das Prädikat `dialog` angeben:

```
dialog :-
   prompt,
   input(Type,Input),
```

```
      nl,
      process(Type,Input,Output),
      output(Output),
      nl,
      dialog.
   dialog :-
      write(ade),
      nl.
```

Mit `dialog` wird eine Schleife realisiert, in der zunächst ein Satz eingelesen wird (`input`); dabei wird gleich festgestellt, ob eine Aussage oder Frage vorliegt. Diese Eingabe wird verarbeitet (`process`). Anschließend wird das Ergebnis ausgegeben (`output`) und schließlich der Dialog von vorne begonnen. Mit der Eingabe `ade` kann der Benutzer den Dialog beenden. Wir wollen später `process` so realisieren, daß die Verarbeitung von `ade` ein *fail* ergibt. Dies führt uns dann in die zweite Klausel von `dialog`, in der nur noch `ade` ausgegeben wird und dann der Dialog beendet wird.
Vor Aufruf des Prädikats `input` erzeugen wir mit `prompt` ein Symbol auf dem Bildschirm, das dem Benutzer andeuten soll, daß er mit der Eingabe an der Reihe ist. Ein solches Symbol nennt man *Prompt*.

```
   prompt :-
      X1 is ">" ,
      X2 is " " ,
      put(X1),
      put(X2).
```

Zur Realisierung von `input` benutzen wir das Prädikat `read_sentence`. Durch erhalten wir eine Liste von Strings. Da unsere Parser aber eine Liste von Atomen als Eingabe erwarten, benutzen wir `convert` zur Umwandlung. Das letzte Element der Liste ist ein Satzzeichen. Mit `delete_last` löschen wir dieses aus der Liste. Die Liste ohne das Satzzeichen ist die eigentliche Eingabe, die zu analysieren ist. Gleichzeitig verwenden wir das Satzzeichen als Kodierung des Typs der Eingabe; eine Frage endet immer mit einem Fragezeichen, eine Aussage mit einem Punkt. Wir beenden `input` mit einem *cut*, da es sich ganz klar, wie bei allen anderen Ein- und Ausgabeprädikaten, um ein deterministisches Prädikat handelt.

```
   input(Type,Input) :-
      read_sentence(ListOfStrings),
      convert(ListOfAtoms,ListOfStrings),
      delete_last(Type,ListOfAtoms,Input),
      !.
```

Bei der Verarbeitung der Eingabe sind zunächst drei Fälle zu unterscheiden:

1. Ende des Dialogs
2. Bearbeitung einer Aussage
3. Bearbeitung einer Frage

Zur Verarbeitung von Aussagen und Anfragen müssen wir diese zunächst syntaktisch analysieren und eine semantische Repräsentation aufbauen. Dazu verwenden wir den im Kapitel 10 entwickelten Parser, der gleichzeitig mit der Analyse eine logische Formel in PROLOG-Notation für den Eingabesatz erzeugt. Das dritte Argument von `process` ist das Ergebnis der Verarbeitung. Wir gehen dabei von einer Liste von Atomen aus, da bei Fragen ja auch mehrere Antworten möglich sind.

```
% Ende
process('.',[ade],_) :-
   !, fail.

% Aussage
process('.',In,[ok]) :-
   s(Sem,In,[]),
   assert(Sem),
   !.
% Aussage nicht analysierbar
process('.',In,[wie,'bitte?']).

% Frage
process('?',In,Out) :-
   s(Sem,In,[]),
   prove(Sem,Out),
   !.
% Frage nicht analysierbar
process('?',In,[wie,'bitte?']).
```

Das Ende des Dialogs wird durch die Eingabe `ade` ausgelöst. Die erste Klausel prüft diesen Fall ab und erzeugt gegebenenfalls ein *fail*.
Die zweite Klausel analysiert die Eingabe, eine Aussage, (`s(Sem,In,[])`) und speichert die Semantik mit `assert` in der Datenbasis ab. Damit steht die Information später zum Beantworten von Fragen zur Verfügung.
Die dritte Klausel dient dazu, bei einem nicht-analysierbaren Satz dem Benutzer dieses Unverständnis kundzutun.
Die vierte Klausel analysiert eine eingegebene Frage mit demselben Parser wie bei Aussagen. Die Semantik erhält man wieder als PROLOG-Formel. Diese wird an das Prädikat `prove` gegeben, das einen Beweis für diese Formel sucht.
Die fünfte Klausel ist wieder für nicht-analysierbare Eingaben vorgesehen.
Bei der Beantwortung von Fragen sind zwei Fälle zu unterscheiden:

1. Ja-Nein-Fragen
2. Ergänzungsfragen

Diese Fälle sind leicht auseinanderzuhalten, wenn wir uns noch einmal das Ergebnis der Semantikkonstruktion für diese beiden Fragetypen ansehen:

```
?- s(Sem,[ist,sokrates,sterblich],[]).
Sem = (term(sterblich(sokrates)):-true)
?- s(Sem,[wer,ist,sterblich],[]).
Sem = (term(sterblich(_067D)):-true)
```

In beiden Fällen erhalten wir ein Fakt, d.h. der Rumpf enthält nur das Atom `true`. Bei einer Ja-Nein-Frage enthält der Kopf keine Variable, bei einer Ergänzungsfrage wird das, wonach gefragt ist, durch eine Variable ausgedrückt[1].

[1]Dies ist übrigens kein Programmiertrick, sondern durchaus auch linguistisch motivierbar. Ergänzungsfragen sind eben gerade so gebaut, daß nach einem unbekannten Argument im Satz gefragt wird und das wird in der Semantik durch eine Variable ausgedrückt.

Wir verwenden deshalb `listvar`, um eine Liste aller Variablen im Kopf zu erhalten. Ist diese Liste leer, handelt es sich um eine Ja-Nein-Frage.
Die Beantwortung einer Frage ist nun ganz einfach. Bei Ja-Nein-Fragen versuchen wir einen Beweis der Formel, die die Anfrage repräsentiert. Dies geschieht durch `call(Head)`. Falls der Beweis gelingt, lautet die Antwort `ja`.
An dieser Stelle zahlt es sich aus, daß wir als semantische Repräsentation der Eingabe PROLOG-Formeln gewählt haben. So können wir die gesamte Lösungssuche dem PROLOG-Interpreter überlassen und brauchen uns darüber nicht den Kopf zu zerbrechen.
Bei einer Ergänzungsfrage könnten wir ebenfalls `call(Head)` benutzen und als Ergebnis die Instantiierung der Variablen in `Head`, die wir mit `listvar` bestimmt haben, zurückgeben. Da wir aber *alle* möglichen Instantiierungen als Ergebnis erhalten möchten, wurde `bagof` verwendet. Die Antwort ist somit die Liste aller möglichen Lösungen.
Falls eine Frage nicht beweisbar ist, scheitert das Unterbeweisziel `call` bzw. `bagof`. Damit scheitert auch die jeweilige Klausel von `prove`. In diesem Falle erhalten wir durch die dritte Klausel die Antwort `nein`.

```
% Antwort erzeugen
%  Ja-Nein-Frage
prove(Sem,[ja]) :-
   Sem = (Head:-true),
   listvar(Head,ListOfVars),
   ListOfVars = [] ,
   call(Head).
%  Ergaenzungsfrage
prove(Sem,Result) :-
   Sem = (Head:-true),
   listvar(Head,ListOfVars),
   ListOfVars = [Var] ,
   bagof(Var,call(Head),Result).
%  keine Antwort gefunden, also "nein" sagen
prove(Sem,[nein]).
```

Als letzte Teilaufgabe bleibt nur noch die Ausgabe der Ergebnisse übrig. Dazu schreiben wir das Prädikat `output`, das einfach eine Liste von Atomen jeweils getrennt durch ein Leerzeichen ausgibt:

```
output([OnlyOne]) :-
   write(OnlyOne),
   !.
output([First|Rest]) :-
   write(First),
   write(' '),
   output(Rest).
```

Damit ist die Realisierung unseres kleinen Dialogsystems abgeschlossen. Natürlich haben wir damit kein in der Praxis einsatzfähiges Produkt geschaffen. Für ein realistisches Dialogsystem wäre viel mehr Arbeit nötig. Es müßte vor allem in den folgenden Bereichen sehr viel getan werden:

1. Man benötigt ein umfangreiches syntaktisches Fragment der Sprache und die zugehörigen Semantikkonstruktionsregeln

2. Die Interaktion mit der Wissensbais muß sehr viel ausgeklügelter sein
3. Vernünftige Dialoge sind nicht ohne *Benutzermodellierung* und *Dialogplanung* möglich
4. Wir brauchen eine „richtige“ Generierung
5. …

Die Liste ließe sich wahrscheinlich beliebig fortsetzen.
Was aber bei unserem Minimalsystem deutlich wurde, ist, daß man in PROLOG mit sehr wenig Aufwand schon ein funktionierendes System, einen *Prototyp*, aufbauen kann. PROLOG erlaubt schnelles Umsetzen von Ideen, die auf logischer Ebene bereits formuliert sind oder leicht formulierbar sind, in ein lauffähiges Programm. Durch Verbesserung einzelner Komponenten kann dann die Leistungsfähigkeit nach und nach erhöht werden. Man spricht deshalb oft von *rapid prototyping*. Diese Eigenschaft ist eine der großen Stärken von PROLOG, die zur raschen Verbreitung dieser Programmiersprache auch über den Universitätsbereich hinaus in den Entwicklungslabors der Industrie geführt hat.

Literaturverzeichnis

[Belli 1986] Belli, F. (1986): Einführung in die logische Programmierung mit PROLOG. Bibliographisches Institut, Mannheim.

[Bonevac 1987] Bonevac, D. (1987): Deduction. Introductory Symbolic Logic. Mayfield Publishing Company, Palo Alto, California.

[Bratko 1987] Bratko, I. (1987): PROLOG. Programmierung für künstliche Intelligenz. Addison-Wesley, Bonn.

[Clark/McCabe 1987] Clark, K.L. und F.G. McCabe (1987): Micro-PROLOG. Logische Programmierung. Mit Beiträgen von M.H. van Emden, J.R. Ennals, S. Gregory, P. Hammond, R.A. Kowalski, F. Kriwaczek, M.J. Sergot. Oldenbourg Verlag, München.

[Clocksin/Mellish 1987] Clocksin, W.F. und C.S. Mellish (1987): Programming in PROLOG. 3rd edition, Springer-Verlag, Berlin.

[Conlon 1985] Conlon, T. (1985): Start Problem-Solving with PROLOG. Addison-Wesley, Reading, Mass.

[Cordes et al. 1988] Cordes, R.; R. Kruse; H. Langendörfer; H. Rust (1988): PROLOG. Eine methodische Einführung. Vieweg, Braunschweig.

[DECSystem-10 PROLOG User's Manual] Department of Artificial Intelligence, University of Edinburgh, Scotland.

[Dickenberger 1987] Dickenberger, U. (1987): Wie die Alten den Tod bedichten. Die Inschriften des Stuttgart Hoppenlau-Friedhofs. Magisterarbeit, Institut für Literaturwissenschaft, Universität Stuttgart.

[Gazdar/Mellish 1987] Gazdar, G. and C. Mellish (1987): Natural Language Processing In PROLOG. An Introduction To Computational Linguistics. New Horizons in Linguistics 2. Ed. by John Lyons, Richard Coates, Gerald Gazdar and Margaret Deuchar. Penguin Books.

[Giannesini et al. 1988] Giannesini, F.; H. Kanoui; R. Pasero und M. van Caneghem (1988): PROLOG. Addison-Wesley, Bonn.

[Hanus 1987] Hanus, M. (1987): Problemlösen mit PROLOG. Teubner, Stuttgart.

[Holland 1986] Holland, G. (1986): Problemlösen mit micro-PROLOG. Teubner, Stuttgart.

[Kleine Büning/Schmitgen 1988] Kleine Büning, H. und S. Schmitgen (1988): PROLOG. Grundlagen und Anwendungen. Leitfäden der angewandten Informatik, Teubner Verlag, Stuttgart, 2. Auflage.

[Kluzniak/Szpakowicz 1985] Kluzniak, F. und S. Szpakowicz (1985): PROLOG for Programmers. Academic Press, London.

[Kononenko/Lavrac 1988] Kononenko, I. und N. Lavrac (1988): PROLOG Through Examples. A Practical Programming Guide. Sigma Press, Wilmslow, Cheshire, U.K.

[Kowalski 1979] Kowalski, R. (1979): Logic for Problem Solving. North-Holland, New York.

[Malpas 1987] Malpas, J. (1987): PROLOG: A Relational Language and its Applications. Prentice Hall, Englewood Cliffs, N.J.

[Marcus 1987] Marcus, C. (1987): PROLOG Programming. Applications for Database Systems, Expert Systems, and Natural Language Systems. Addison-Wesley Publications, Reading, Mass.

[Pereira/Shieber 1987] Pereira, F. und S.M.Shieber (1987): PROLOG And Natural Language Analysis. Center for the Study of Language and Information. Stanford University, Stanford, California.

[Pereira/Warren] Pereira, F.C.N. and D.H.D. Warren (1980): Definite Clause Grammars for Natural Language Analysis. A Survey of the Formalism and a Comparison with Augmented Transition Networks. In: Artificial Intelligence, 13, 231-278.

[Rogers 1987] Rogers, J.B. (1987): A PROLOG Primer. Addison-Wesley, Reading, Mass.

[Schnupp 1986] Schnupp, P. (1986): PROLOG: Einführung in die Programmierpraxis. Hanser, München.

[Schöning 1987] Schöning, U. (1987): Logik für Informatiker. Bibliographisches Institut Wissenschaftsverlag, Mannheim. Reihe Informatik, Band 56.

[Sterling/Shapiro 1988] Sterling, L. und E. Shapiro (1988): PROLOG. Fortgeschrittene Programmiertechniken. Addison-Wesley, Bonn.

[Walker et al. 1987] Walker, A., M. McCord, J.F. Sowa und W.G. Wilson (1987): Knowledge Systems and PROLOG. A Logical Approach to Expert Systems and Natural Language Processing. Addison-Wesley Publishing Company, Reading, Massachusetts.

Anmerkungen zur Literatur

Die Popularität von PROLOG drückt sich auch in der wachsenden Zahl von Veröffentlichungen aus. Dieses Literaturverzeichnis stellt eine Auswahl der neueren Veröffentlichungen dar und verweist außerdem auf einige etwas speziellere Quellen, die für das vorliegende Buch verwendet wurden.
Nur vier Titel, die für das weitere Arbeiten mit PROLOG für Sie wahrscheinlich von größerer Bedeutung sind, sollen hier kurz besprochen werden:

- [Clocksin/Mellish 1987]: Eine der ersten leicht verständlichen PROLOG-Einführungen. Standardwerk. Verwendet den de facto Standard von PROLOG.
- [Giannesini et al. 1988]: Einführung und Vertiefung, ausführliche Anwendungsbeispiele für natürlichsprachliche Systeme, kleines Expertensystem. Verwendet PROLOG II.
- [Pereira/Shieber 1987]: Gute Übersicht über gängige Algorithmen der Computerlinguistik und deren Kodierung in PROLOG. Setzt Kenntnisse in Logik, Automatentheorie und Computerlinguistik voraus. Geeignet als weiterführende Literatur zu diesem Buch.
- [Sterling/Shapiro 1988]: Fortgeschrittene Programmiertechniken. Sehr gute Einführung in die Theorie und Praxis der Logikprogrammierung.

Index