Einführung in die Computerlinguistik

Vorlesung WS 1995/96
Centrum für Informations-
und Sprachverarbeitung

Universität München

Karsten Wenger

Inhalt

Einleitung

Writing a grammar for a problem is very
much like composing a program..
Michael A. Harrison

Was ist COMPUTERLINGUISTIK ?

Die Frage, was Computerlinguistik (auch Sprachverarbeitung genannt) ist, wollen wir nicht direkt beantworten, sondern über den Umweg, ein typisches Ziel der Computerlinguistik zu beschreiben. Das erste praktische Vorhaben der künstlichen Intelligenzforschung im sprachlichen Bereich hieß *Maschinelle Übersetzung*. Gemeint war damit die Übertragung von, sagen wir, französischen Sätzen in englische Sätze. Die Form der Sätze beider Sprachen war dabei rein textuell oder schriftlich, da von *akustischer Spracherkennung* noch keine Rede sein konnte. Dennoch war diesen frühen computerlinguistischen Gehversuchen kein allzu großer Erfolg beschieden. Eine genaue Betrachtung einer konkreten Aufgabenstellung wird uns zeigen warum: Nehmen wir an, jemand bittet uns, eine Passage aus einer französischen Zeitung ins Englische zu übersetzen. Vorausgesetzt wir haben beide Sprachen gelernt, wird unsere Vorgehensweise in etwa die folgende sein:

(1) Wir versuchen zunächst den Ausgangstext zu verstehen. Dazu lesen wir Wort für Wort und Satz für Satz bis ans Ende des Textes. Während des Lesens suchen wir nach den deutschen Wörtern, die den französischen entsprechen. Außerdem müssen wir die von den deutschen abweichenden grammatischen Regeln des Französischen beherrschen.

(2) Ist es uns erst mal gelungen, den Quelltext zu verstehen, stehen wir vor dem noch schwierigeren Problem, den Inhalt ins Englische zu übertragen. Zu diesem Zweck suchen wir nach den englischen Worten, die den deutschen Worten entsprechen, die wiederum Übersetzungen der französischen Worte darstellen. Schließlich müssen die gefundenen englischen Worte in eine grammatikalisch korrekte Form des Englischen gebracht werden.

Der Erfolg einer solchen Übersetzung hängt dabei von vielen Komponenten ab. Die rein sprachlichen sind die folgenden:

(i) die Größe unseres französischen Lexikons
(ii) die Größe unseres englischen Lexikons
(iii) unsere Kenntnis der beiden Grammatiken

Daneben gibt es noch eine Reihe von weiteren Aspekten, die außerhalb des sprachlichen Bereichs liegen:

(iv) den Sachverhalt, um den es in dem Text geht

(v) die Kenntnis der kulturellen, politischen etc. Fakten, auf die sich der Text bezieht

(vi) die Eigenschaft unter verschiedenen möglichen Übersetzungen, die korrekte bzw. die beste zu finden

Die Vorgehensweise bei der maschinellen Übersetzung ist im Kern genau die gleiche wie bei dieser menschlichen Übersetzung. Daher müssen auch dieselben Voraussetzungen, die man beim menschlichen Übersetzer findet, in computerisierter Form vorliegen. Der Computer benötigt, wie der menschliche Übersetzer, zunächst eine Übertragung der Quellsprache in seine interne Sprache, d.h. in eine der Programmiersprachen, die er versteht. In der Computerlinguistik sind dies *Prolog* und in geringerem Maße *Lisp*. Dazu muß der Computer zumindest über das Lexikon und die Grammatik der natürlichen Sprache verfügen. Dann kann die Erkennung oder Analyse der Quellsprache erfolgen. Um die Sätze der Zielsprache zu produzieren oder zu generieren, müssen die entsprechenden Grundlagen (Grammatik) der Zielsprache in einer maschinenverstehbaren Form vorhanden sein. Wir haben gerade argumentiert, daß für eine gute Übersetzung aber auch die Eigenschaften (iv) bis (vi) vonnöten sind. Im Unterschied zu den linguistischen Grundlagen ist aber dieser Bereich (v), also des Wissens um bestimmte Weltzusammenhänge, weit weniger gut absteckbar und daher auch schlecht implementierbar. Menschliche Eigenschaften, wie (vi), die auch erfahrungsabhängig sind, entziehen sich sogar noch stärker einer *Implementierung*. Aus diesen Gründen ist die maschinelle Übersetzung auch am erfolgreichsten in Bereichen, die die Eigenschaften (iv) bis (vi) in möglichst geringer Ausprägung erfordern, also z.B. in Fachgebieten mit einem geringen bzw. wohlbekannten Anteil an *Weltwissen*.

Mit gewissen Abstrichen ist dieses Bild der maschinellen Übersetzung auf das ganze Fach der Computerlinguistik übertragbar. Die Schritte (1) und (2) von oben können so als Beispiele für die beiden wichtigsten Gebiete des Faches verstanden werden: *Parsing* und *Generierung*. Die Hauptarbeit des Computerlinguisten besteht zum einen Teil in der Bereitstellung der linguistischen und außersprachlichen Fakten, wie sie oben beschrieben wurden. Mindestens genau so viel Gewicht liegt aber auf der Übertragung dieser Fakten in eine maschinenverstehbare Form. Dieser Prozeß, *Formalisierung* genannt, endet in einer wie auch immer gearteten Form der Darstellung, die als *Repräsentation* bezeichnet wird. Dies darf nicht als eine untergeordnete Frage unterschätzt werden, da die Frage der Repräsentation eng mit der Frage der *Verarbeitung*, also z.B. mit Eigenschaften wie Effizienz und Korrektheit, zusammenhängt. Daher kann es auch nicht verwundern, daß Computerlinguisten gerne Anleihen in einem Fach machen, das sich die Untersuchung von der *Komplexität* und Korrektheit gegebener Darstellungen (oder Repräsentationen) auf die Fahne geschrieben hat. Die Rede ist von der formalen Logik und den logischen Sprachen bzw. den Kalkülen. Typischerweise findet man daher in computerlinguistischen Anwendungen *logische Repräsentationen* der behandelten Wissensgebiete.

Der Ansatz, die Frage nach der Definition der Computerlinguistik über die Beschreibung einer seiner Anwendungen zu beantworten, ist nicht zufällig gewählt. Der Computerlinguist versteht sich in erster Linie als Ingenieur, der eine gestellte Aufgabe lösen will. Die grundsätzlichen Fragen der Disziplinen, aus denen sich die Computerlinguistik zusammensetzt, müssen daher notgedrungen in den Hintergrund rücken. So ist die Frage nach dem Vorhandensein eines angeborenen Sprachvermögens beim Menschen, die einen Großteil der amerikanischen Tradition der Linguistik beschäftigt, im Bereich der Computerlinguistik von untergeordneter Bedeutung. Wichtiger erscheint es hier, Algorithmen und Mechanismen zu finden, die es erlauben, konkrete Ziele im Bereich der Sprachverarbeitung zu erreichen. Man kann dafür die Analogie zu einem anderen Fach zu hilfe nehmen. Die Ingenieurswissenschaft hat das Problem des Fliegens gelöst, indem sie das Flugzeug konstruierte, ohne das biologische Modell des Vogelflugs nachgeahmt zu haben. Ähnlich läßt sich das Problem der maschinellen Übersetzung in speziellen Bereichen lösen, ohne jemals einen menschlichen Übersetzer mit allen seinen Fähigkeiten simulieren zu können. So wie das Flugzeug in einigen Eigenschaften die biologischen Modelle weit übertrifft (etwa in der Geschwindigkeit), so kann ähnliches auch für computerlinguistische Verfahren (etwa bei der Übersetzung von Massentexten) zutreffen. In diesem Sinne läßt sich Computerlinguistik auch nicht in einfacher Weise auf eine Addition seiner zugrundeliegenden Disziplinen zurückführen. In gewisser Hinsicht läßt sich Computerlinguistik sogar als Innovation der klassischen Fächer der Linguistik und der Informatik verstehen, die aus anderer Motivation heraus zu Resultaten führt, die ihrerseits wieder in die Grundlagenfächer zurückfliessen. So wirken die Erkenntnisse von Noam Chomsky über das Gebiet der natürlichen Sprachen in das der formalen Sprachen hinein (man denke etwa an die *Chomsky-Hierarchie*), das zu den Fundamenten der Informatik zählt. Ein anderes Beispiel ist das Konzept der *Unifikation*, das ursprünglich als eine Eigenschaft von Prolog für das Pattern-Matching entwickelt wurde, in letzter Zeit aber auch Eingang in die Theoriebildung der formalen Linguistik gefunden hat. Sogar die Programmiersprache Prolog selbst ist für die Lösung von Problemen der Computerlinguistik entwickelt worden, während ihre heutige Verbreitung weit über dieses Fach hinaus in die Informatik (als Modellierungssprache) und in die Logik (als Theorembeweiser) geht.

Man kann daher Computerlinguistik zwar als eine Verbindung von Linguistik, Informatik und Logik zu einer Disziplin betrachten. Man muß dabei jedoch beachten, daß die spezielle Ausrichtung des Faches auf die Konstruktion funktionierender Mechanismen, eine eigenständige Dynamik hervorbringt, die zu Resultaten führt, die von den zugrundeliegenden Fächern nur selten oder gar nicht erreicht werden.

Dieses Einführung in die Computerlinguistik folgt der gerade beschriebenen ingenieurswissenschaftlichen Ausrichtung. Der Schwerpunkt der Einführung liegt daher nicht auf der Vermittlung der linguistischen oder formalen Grundlagen des Faches, sondern auf der technisch präzisen Darstellung typischer Anwendungen der Computerlinguistik. Das Ziel besteht in der Beschreibung eines möglichst repräsentativen Querschnitts aller Teilbereiche

und deren Zusammenwirkens. Um dies zu erreichen steht die Implementierung der Teilprobleme im Vordergrund. Neben Deutsch verwenden wir Prolog als zweite Metasprache. Dies hat den großen Vorteil, daß nicht nur **jeder** Schritt einer Problemlösung vom Leser nachvollzogen und die Korrektheit überprüft werden kann, sondern die vorgestellten Programme auch als Motivation für das eigene Arbeiten dienen können. Diese Vorteile erfordern allerdings einen nicht zu unterschätzenden Preis: die Beherrschung der Programmiersprache Prolog (!!!). Diese Sprache hat den Vorzug, daß sie von der prozeduralen oder maschinenmodellabhängigen Seite der Implementierung stark abstrahiert. Außerdem stellt sie eine Art Lingua Franka der Computerlinguistik dar.

Aus dem Gesagten geht hervor, daß die vorliegende Einführung sich nicht in erster Linie an den absoluten Anfänger richtet, sondern schon eine Reihe von Fähigkeiten beim Leser voraussetzt. Idealerweise beherrscht dieser die inhaltlichen Grundlagen der Linguistik und wenigstens eine (symbolische) Programmiersprache. Auf dieser Basis läßt sich die vorliegende Einführung als begleitendes Arbeitsbuch verwenden, das dem linguistisch interessierten Leser einen Kernbereich der Computerlinguistik erschließt.

Die Module der Computerlinguistik

Typische Anwendungen der Computerlinguistik bestehen immer mindestens aus einem rein linguistischen Teil. Dazu kommen je nach Bereich weitere linguistische Module wie im Fall der maschinellen Übersetzung oder nicht-linguistische Module wie z.B. bei der natürlichsprachlichen Datenbankabfrage. Es ist aus verschiedenen Gründen wünschenswert, diese Aufteilung der verschiedenen Aufgaben in der Implementierung nachzuvollziehen. Daher wird eine Formulierung der obersten Programmebene diese Aufteilung reflektieren. Eine erste Annäherung an das zu lösende computerlinguistische Problem hat also das folgende allgemeine Aussehen:

```
computerlinguistisches_problem :-
    grammatikalische_analyse,
    anwendung.
```

Die Semantik dieser Definition lautet: Das jeweilige computerlinguistische Problem ist gelöst, wenn sich die beiden Unteraufgaben der grammatikalischen Analyse, also des linguistischen Teils der Aufgabe und der spezifischen Anwendung lösen lassen. Der Funktor ":-", der diese Regel unterteilt, ist also eigentlich als umgekehrte Implikation "<-" zu lesen, in der die Folgerung links vom Pfeil (oder von ":-") steht.

Typischerweise erscheint das spezielle computerlinguistische Problem in Form eines natürlichsprachlichen Ausdrucks wie eines Satzes. Besteht der Anwendungsteil dieses Problems aus der Abfrage einer Datenbank, dann resultiert aus der Erkennung eines Satzes letztlich ein Informationsdatum dieser Datenbank. Damit läßt sich die erste Definition so abändern, daß die Teilaufgaben diese spezifischen Informationen als explizite Parameter (Satz, Daten) enthalten:

```
computerlinguistisches_problem(Satz , Daten ):-
    grammatikalische_analyse(Satz),
    anwendung(Daten ).
```

Nehmen wir als Beispiel eine Datenbank über Astronomen und Himmelskörper, dann wären die folgenden Belegungen dieser Parameter mögliche Instanzen einer Lösung des computerlinguistischen Problems:

```
Satz = [ welcher,astronom, entdeckte,uranus ]
Daten = [ herschel ]
```

Bei der letzten Definition der Regel haben wir die Schnittstelle zwischen Grammatik und Anwendung nicht explizit gemacht. Denn natürlich muß die spezifische Information, dessen Träger ein natürlichsprachlicher Satz ist, an den Datenbankteil ebenfalls als expliziter Parameter weitergeleitet werden. Diese Information hat die Struktur einer logischen Formel, die

das Resultat der grammatikalischen Analyse darstellt. Daher besteht die Eingabe an den Anwendungsteil, also die Datenbankabfrage, auch aus dieser Formel, die z.B. für obigen Satz das folgende Aussehen hat:

```
LogischeRepraesentation = existiert(X, astronom(X) & entdecken(X,uranus))
```

Somit läßt sich die Definition der Regel der obersten Ebene endgültig formulieren:

```
computerlinguistisches_problem(Satz,   Daten):-
    grammatikalische_analyse(Satz, LogischeRepraesentation ),
    anwendung(LogischeRepraesentation , Daten).
```

Die Bedeutung dieser Regel läßt sich also auch so beschreiben: Jeder Teil der Regel besteht aus einer Relation, die sich aus einem Eingabe- und einem Ausgabeparameter zusammensetzt. Da der jeweils letzte Ausgabeparameter den Eingabeparameter für den nächsten Teil der Regel (von oben nach unten) darstellt, kann man von einer sukzessiven Modifikation der ursprünglichen Eingabe sprechen, deren letztes Resultat aus der finalen Ausgabe besteht.

Diese Art der Problemlösung ist so allgemein, daß sie sich leicht auf andere Aufgaben anpassen läßt. Nehmen wir an, das computerlinguistische Problem bestände in der Übersetzung eines deutschen Satzes ins Englische, dann würde es ausreichen, das Modul "anwendung" so abzuändern, daß aus seiner Eingabe, also der logischen Repräsentation des deutschen Satzes, der entsprechende englische Satz berechnet würde. Regel und Belegungen, analog zu den obigen Beispielen hätten z.B. folgende Form:

```
computerlinguistisches_problem(DeutscherSatz,   EnglischerSatz):-
    grammatikalische_analyse(DeutscherSatz, LogischeRepraesentation ),
    anwendung(LogischeRepraesentation , EnglischerSatz).

DeutscherSatz = [ welcher,astronom, entdeckte,uranus ]
LogischeRepraesentation = existiert(X, astronom(X) & entdecken(X,uranus))
EnglischerSatz = [ which,astronomer,discovered,uranus ]
```

Die Parameter einer solchen Regel erhalten ihre Bedeutung also nicht aus ihrer Benennung, sondern aus der Tatsache, daß sie den gleichen Namen besitzen wie andere Parameter der gleichen Regel. Außerhalb der Regel sind diese Namen jedoch bedeutungslos. Daher könnte man ihnen auch Namen geben, die nur ihren relationalen Charakter reflektieren und von der Art der jeweiligen Problemlösung abstrahieren. Eine entsprechende Definition der obigen Regel wäre die folgende:

```
computerlinguistisches_problem(X, Z):-
    grammatikalische_analyse(X, Y),
    anwendung(Y, Z).
```

Da wir uns im folgenden aber sowieso im wesentlichen auf eine einzige Beispielsanwendung, nämlich auf natürlichsprachliche Datenbankabfrage, konzentrieren werden, können wir die sprechenden Namen der Parameter auch beibehalten.

Mit der Definition der hierarchisch obersten Prozedur haben wir gleichzeitig auch die Art und Weise der Explikation der noch folgenden Konzepte dargelegt: Der Kern der Besprechung jeder Funktionalität besteht aus der Definition des entsprechenden logischen Programms, dessen Semantik wir mit Beispielen klarzumachen versuchen.

Der Schwerpunkt der folgenden Kapitel liegt auf der Definition des Moduls Grammatikalische Analyse. Die oberste Ebene dieses Moduls läßt sich, wenn man den herkömmlichen Einführungen in die Linguistik folgt, in unserem Rahmen z.B. als folgende Prozedur implementieren:

```
grammatikalische_analyse( Eingabe, Repräsentation):-
    morphologische_analyse(Grundformen, Vollformen),
    lexikalische_analyse(Eingabe, Getaggte_Eingabe),
    syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
    semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
    pragmatische_analyse(SemantischeRepräsentation, Repräsentation).
```

Wir halten uns aus Gründen der Vollständigkeit an diese Unterteilung der Aufgaben, werden jedoch die beiden ersten Module, Morphologische und Lexikalische Analyse, nur anreißen und uns vor allem auf Syntax und Semantik konzentrieren. Für die Anwendung auf die natürlichsprachliche Datenbankabfrage werden wir die beiden ersten Module aus der Definition herausnehmen und von einer direkten Verarbeitung der Eingabesätze ausgehen, so daß die abgekürzte Fassung der Definition des Grammatikmoduls wie folgt lautet:

```
grammatikalische_analyse( Eingabe, Repräsentation):-
    syntaktische_analyse(Eingabe, SyntaktischerBaum),
    semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
    pragmatische_analyse(SemantischeRepräsentation, Repräsentation).
```

Lexikon

Morphologie

Das Lexikon stellt innerhalb der Grammatik gewissermaßen das Ausgangswissen dar, auf das z.B. die Syntax zugreift, um Satzstrukturen aufbauen zu können. Das heißt aber nicht, daß das Lexikon selbst unveränderlich sei. Bestimmte morphologische Prozeße (Derivation, Kompositabildung) erlauben eine im Prinzip unbeschränkte Erweiterung des Wortbestands, die innerhalb des Moduls "morphologische_analyse" stattfindet:

```
grammatikalische_analyse(Eingabe, Repräsentation):-
    morphologische_analyse(Grundformen, Vollformen),
    lexikalische_analyse(Eingabe, Getaggte_Eingabe),
    syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
    semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
    pragmatische_analyse(SemantischeRepräsentation, Repräsentation).
```

Für den Moment gehen wir aber davon aus, daß das Lexikon über eine feste Anzahl von Basiseinträgen verfügt. Dann würden wir den Anschluß an unser Programm wie folgt definieren:

```
morphologische_analyse(Grundformen, Vollformen):-
    wort(Grundformen,Vollformen, Person, Numerus, Zeitform, Flexion, SemantischerTyp).
```

Die einem Wort zugeordneten Informationen umfassen orthographische, phonologische, semantische und morpho-syntaktische Einheiten. So könnte der Lexikoneintrag eines Verbs z.B. das folgende Aussehen haben:

```
wort(entdecken,entdecken, drittePerson, plural, präsens, schwach, agens(menschlich)).
```

Derartige Festlegungen eines Wortes werden Merkmale genannt. So sind "drittePerson, plural, präsens, schwach" alles morpho-syntaktische Merkmale, während "agens(menschlich)" als semantisches Merkmal bezeichnet wird. Die morpho-syntaktischen Merkmale sorgen in der Syntax dafür, daß das Wort "entdecken" in dem Satz

```
Die Reporter entdecken Waffenschiebereien.
```

vorkommen kann, aber nicht in:

* Ein Reporter entdecken Waffenschiebereien.

Ebenso legt das semantische Merkmal fest, daß bestimmte Sätze zulässig sind und andere nicht.

* Die Häuser entdecken Waffenschiebereien.

Natürlich ist der obige Lexikoneintrag nur ein Beispiel - normalerweise verfügen derartige Einträge über eine wesentlich reichhaltigere Informationsstruktur. Außerdem sollte das Lexikon auch alle anderen Formen eines Wortes abdecken, also morphologische Vollständigkeit besitzen. Dies kann z.B. in Form von einer Menge von Prolog-Fakten geschehen:

```
wort(entdecken,entdecke,erste_person,singular,praesens,schwach, agens(menschlich)).
wort(entdecken,entdeckst,zweite_person,singular,praesens,schwach, agens(menschlich)).
wort(entdecken,entdeckt,dritte_person,singular,praesens,schwach, agens(menschlich)).
wort(entdecken,entdecken,erste_person,plural,praesens,schwach, agens(menschlich)).
wort(entdecken,entdeckt,zweite_person,plural,praesens,schwach, agens(menschlich)).
wort(entdecken,entdecken,dritte_person,plural,praesens,schwach), agens(menschlich)).
```

Diese Art ein Lexikon aufzubauen, ist in Prolog sehr bequem zu bewerkstelligen. Allerdings entgehen einem auf diese Weise eine Reihe von Redundanzen. So benötigt man allein für die Präsens-Formen des Verbs "entdecken" sechs Einträge, obwohl man das Verb auch anders repräsentieren könnte, wenn man es als Kombination aus einem Stamm-Morphem oder Lexem "entdeck" und verschiedenen Endungen oder Suffixen wie "e", "st", "t", "en" auffaßt. Alles, was man dafür braucht, ist eine Reihe von Anweisungen, wie die Kombinationen stattzufinden haben. Der Anschluß an unser Grammatikmodul müßte zunächst so definiert werden, daß ein Wort nicht direkt in der Datenbank steht, sondern über ein Unterprogramm (hier: verb_flexion) erst erzeugt wird.

```
morphologische_analyse(VerbStamm, Vollform):-
        verb_flexion(VerbStamm,StarkSchwach,Tempus,Person,Numerus,Vollform).
```

Das folgende Prologprogramm erfüllt diese Aufgabe nach der Formel:

VerbStamm + TempusMorphem + PersonNumerus = Vollform

```
verb_flexion(VerbStamm,StarkSchwach,Tempus,Person,Numerus,Vollform):-
        person_numerus(Person,Numerus,StarkSchwach,Tempus,PersonNumerus),
        tempus_affix(Tempus,StarkSchwach,Person,Numerus,TempusMorphem),
        name(TempusMorphem,TempusMorphemAscii),
        name(PersonNumerus,PersonNumerusAscii),
        name(VerbStamm,VerbStammAscii),
        append(VerbStammAscii,TempusMorphemAscii, ZwischenMorphemAscii),
        append(ZwischenMorphemAscii,PersonNumerusAscii,VollformAscii),
        name(Vollform, VollformAscii).

tempus_affix(praesens,_,_,_,'').

person_numerus(erste_person,singular,StarkSchwach,Tempus,'e').
person_numerus(zweite_person,singular,StarkSchwach,Tempus,'st').
person_numerus(dritte_person,singular,StarkSchwach,Tempus,'t').
person_numerus(erste_person,plural,StarkSchwach,Tempus,'en').
person_numerus(zweite_person,plural,StarkSchwach,Tempus,'t').
person_numerus(dritte_person,plural,StarkSchwach,Tempus,'en').

append([],X,X).
append([X|Y],Z,[X|U]):-
        append(Y,Z,U).
```

Die Prozedur "verb_flektion" holt sich aus "person_numerus" die Endungsinformationen für ein Verb, dessen Stamm im Aufruf der Variablen "VerbStamm" angegeben wird. Das tempus_affix für den Präsens ist in diesem Fall leer (= ''), wird aber in der Kodierung für den Imperfekt notwendig. Die Suffixe und Affixe werden über "name" in ihre Ascii-Darstellung übersetzt, um die Wörter und Endungen via "append" zusammenzusetzen. Dieses Programm ist nun in der Lage, mithilfe der Stammform eines Verbs dessen Vollformen zu generieren.

Der folgende Aufruf zeigt dies:

```
?- verb_flexion(entdeck,schwach,praesens,Person,Numerus,Vollform).

Person=erste_person,Numerus=singular,Vollform=entdecke;
Person=zweite_person,Numerus=singular,Vollform=entdeckst;
Person=dritte_person,Numerus=singular,Vollform=entdeckt;
Person=erste_person,Numerus=plural,Vollform=entdecken;
Person=zweite_person,Numerus=plural,Vollform=entdeckt;
Person=dritte_person,Numerus=plural,Vollform=entdecken;
```

Das klappt nicht nur für das Verb "entdecken", sondern natürlich für jedes regelmäßige deutsche Verb, das schwach konjugiert wird. Somit wird der Speicheraufwand auf ein Sechstel dessen reduziert, was mit der Methode der Abspeicherung der Vollformen erreicht wird.

Übungsaufgabe 1:
Erweitern Sie das obige Morphologieprogramm, so daß auch die Imperfektformen des Verbs "entdecken" erzeugt werden, also:
endeckte, endecktest, endeckte, entdeckten, entdecktet, endeckten.

Übungsaufgabe 2:
Mit der Lösung von Übungsaufgabe 1 deckt das Programm alle Präsens- und Imperfektformen für schwache Verben ab. Erweitern Sie das Programm auf starke Verben wie z.B. "singen". Hier gibt es unterschiedliche Stämme für die beiden Zeitformen und auch unterschiedliche Endungen:

	Präsens	Imperfekt
Stamm	sing	sang
1. Person Singular	singe	sang
2. Person Singular	singst	sangst
3. Person Singular	singt	sang
1. Person Plural	singen	sangen
2. Person Plural	singt	sangt
3. Person Plural	singen	sangen

Übungsaufgabe 3:
Zu diesen regelmäßigen Endungen kommen nun Verben wie "finden" dazu, deren Stamm auf "d" oder "t" endet, die in der 2. und 3. Person Singular sowie in der 2. Person Plural ein "e" einschieben. Programmieren Sie auch diese phonologische Besonderheit.

Ein weiteres Beispiel aus der deutschen Morphologie ist die Flexion der Adjektive in attributiver Stellung. Diese richten sich in ihren Endungen nicht nur nach dem Nomen sondern auch nach dem vorangehenden Artikel. So unterscheidet man starke, schwache und gemischte Flexion. Die folgende Tabelle (aus ENGEL,ULRICH: Deutsche Grammatik) enthält diese Formen beispielsartig:

Flexionsarten:		singular			plural
		masc	fem	neu	masc/fem/neu
schwach					
z.B. in:	nom	wirkliche	wirkliche	wirkliche	wirklichen
der Grund	akk	wirklichen	wirkliche	wirkliche	wirklichen
die Ursache	gen	wirklichen	wirklichen	wirklichen	wirklichen
das Ziel	dat	wirklichen	wirklichen	wirklichen	wirklichen
stark					
z.B. in:	nom	wirklicher	wirkliche	wirkliches	wirkliche
ein Grund	akk	wirklichen	wirkliche	wirkliches	wirkliche
eine Ursache	gen	wirklichen	wirklichen	wirklichen	wirklicher
ein Ziel	dat	wirklichen	wirklichen	wirklichen	wirklichen
gemischt					
z.B. in:	nom	wirklicher	wirkliche	wirkliches	wirkliche
dessen Grund	akk	wirklichen	wirkliche	wirkliches	wirkliche
deren Ursache	gen	wirklichen	wirklicher	wirklichen	wirklicher
dessen Ziel	dat	wirklichem	wirklicher	wirklichem	wirklichen

Auch hier läßt sich die entsprechende Kodierung leicht als Prologfaktum wiedergeben:

```
wort(wirkliche, def, nom, singular, masc).
```

Um alle Formen zu kodieren benötigt man 48 Prologfakten. Wenn man Variablen verwendet, lassen sich aber gewisse Verallgemeinerungen treffen, die zu einer Reduzierung der Zahl der Einträge führen. So geben z.B.:

```
wort(wirklichen, def, Kasus, plu, Genus).
wort(wirklichen, def, gen, Numerus, Genus).
wort(wirklichen, def, dat, Numerus, Genus).
```

die Verallgemeinerung wieder, daß bei schwacher Deklination alle Formen im Plural gleich sind, bzw. daß die Genitiv- und Dativformen unabhängig von Numerus und Genus gelten. Auf diese Weise kommt man mit 3 Definitionen anstatt 10 aus. Natürlich eignet sich diese Kodierung nur für das Testen bestimmter Formen, nicht aber für das Generieren. So bringt die Abfrage:

```
?- wort(wirklichen, def, Kasus, plu, Genus).
```

nur die Numerusinformation, aber keine weiteren. Wenn also die konkreten Werte gebraucht werden, muß die Information dafür zur Verfügung gestellt werden, etwa in der Form:

```
?- wort(wirklichen, def, Kasus, plu, Genus), member(Kasus, [nom,akk,gen,dat]).

Kasus= nom;
Kasus= akk;
Kasus= gen;
Kasus= dat
```

Dies ist zwar weniger aufwendig kodiert, aber nicht effizienter als die ausführliche Version alle Kombinationen als Fakten zu kodieren. Für das Formentesten ist es jedoch eine Einsparung, da in diesen Fällen kein Backtracking stattfindet.

Übungsaufgabe 3.1:

Finden Sie weitere Verallgemeinerungen, um die Adjektive möglichst kompakt zu kodieren. Welche Anzahl von Prologfakten braucht man minimal, um alle Formen testen zu können?

Tagging

Unabhängig davon, ob das Lexikon über Vollformen aufgebaut wird oder über eine morphologische Komponente, ist das Tagging notwendige Voraussetzung für die Teilaufgabe **lexikalische_analyse** unseres Grammatikmoduls:

```
grammatikalische_analyse( Eingabe, Repräsentation):-
    morphologische_analyse(Grundformen, Vollformen),
    lexikalische_analyse(Eingabe, Getaggte_Eingabe),
    syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
    semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
    pragmatische_analyse(SemantischeRepräsentation, Repräsentation).
```

Die Aufgabe der lexikalische Analyse besteht in der Zuordnung der Wörter der Eingabe zu Typ- oder Kategorialbezeichern. So soll etwa dem Verb "entdecken" die terminale Kategorie "v" für Verb zugeordnet werden. Die folgende Übersicht zeigt eine Reihe von terminalen Kategorien:

Kategorie	Typbezeichner	Beispiel
Verb	v	entdecken
Nomen	n	mann
Adverb	adv	gestern
Adjektiv	adj	neu
Präposition	p	über
Artikel	det	der
Konjunktion	conj	daß
Relativpronomen	rpro	dessen

Nehmen wir einen Beispielssatz wie

```
Die Reporter entdecken Waffenschiebereien.
```

dann ist das gewünschte Resultat von der Form:

```
det(Die) n(Reporter) v(entdecken) n(Waffenschiebereien)
```

Der Sinn dieser Typisierung der Wörter einer Eingabe besteht einerseits darin zu überprüfen, ob sie im Lexikon sind. Andernfalls braucht man nämlich keine weitere syntaktische Analyse angehen, da diese an der Stelle des unbekannten Wortes stehenbleiben würde. Außerdem kann eine solche Vorabzuordnung der Syntaxanalyse die Arbeit erleichtern, indem sie ihr

terminale Informationen zur Verfügung stellt. Schließlich ist es möglich, über die Bestimmung des kategorialen Status der Wörter im Eingabesatz das Lexikonmodul drastisch zu reduzieren und nur die Informationen im Speicher zu halten, die diesen Wort-Kategorie-Paaren entsprechen. Ein Nachteil bei unserer modularen Vorgehensweise kann allerdings die Tatsache bedeuten, daß es Homonyme gibt. So ist das Wort "der" kategorial ambig zwischen Relativpronomen und Artikel. Die lexikalische Analyse wird sich in diesem Fall für eine Typisierung entscheiden und im Falle des Scheiterns in der Syntaxanalyse per Backtracking auf die zweite Möglichkeit zurückgreifen. Wir werden später auf die Nachteile dieses Ansatzes zurückkommen und zeigen, wie sie umgangen werden können.

Nehmen wir die folgenden Definitionen der terminalen Kategorien Nomen, Verb und Artikel an, so können wir das Prädikat **lexikalische_analyse** einfach über eine rekursive Prozedur definieren, die die Liste der Worte durchgeht und jedem Wort, das sie im Lexikon auffindet, eine Kategorie zuordnet.

```
lexikalische_analyse(Eingabe, Getaggte_Eingabe):-
    ordne_zu(Eingabe, Getaggte_Eingabe).

ordne_zu([],[]).
ordne_zu([Wort|Worte], [Lemmatisierung|Lemmatisierungen]):-
    terminal(Wort,Lemmatisierung),
    ordne_zu(Worte, Lemmatisierungen).

terminal(W,v(W)):-  v(W,_,_,_,_,_).
terminal(W,n(W)):-  n(W,_,_,_).
terminal(W,det(W)):-  det(W,_,_,_).

v(entdecken,praesens,schwach,erste_person,plural,entdecken).
v(decken,praesens,stark,erste_person,plural,decken).

n(reporter,plural,nominativ,maskulin).
n(reporter,singular,nominativ,maskulin).
n(waffenschiebereien,plural,nominativ,feminin).
n(politiker,plural,nominativ,maskulin).

det(die,plural,nominativ,maskulin).
det(der,singular,nominativ,maskulin).
```

Der folgende Aufruf zeigt, daß der erste Beispielssatz lexikalisch ambig ist, da es zwei Lexikoneinträge für "Reporter" gibt. Der zweite Beispielssatz dagegen ist völlig eindeutig, da das Wort "Politiker" nur auf eine Weise charakterisierbar ist.

```
?- lexikalische_analyse([ die, reporter, entdecken, waffenschiebereien ], Getaggte_Eingabe).

Getaggte_Eingabe=[det(die),n(reporter),v(entdecken),n(waffenschiebereien)];
Getaggte_Eingabe=[det(die),n(reporter),v(entdecken),n(waffenschiebereien)]
```

```
?- lexikalische_analyse([ die, politiker, decken, waffenschiebereien ], Getaggte_Eingabe).

Getaggte_Eingabe=[det(die),n(politiker),v(decken),n(waffenschiebereien)]
```

Natürlich ist die Struktur, die wir in den Beispielen dargestellt haben, eine vereinfachte. Anstatt ein Wort nur mit seinem Kategoriebezeichner zu versehen kann man es auch mit allen Lexikoninformationen repräsentieren, also z.B.:

```
n(reporter,plural,nominativ,maskulin)
```

Aus den obengenannten Gründen ist aber eine zu genaue Festlegung eher hinderlich. Daher sollten nur die Merkmale spezifiziert werden, die unzweideutig sind, also in diesem Fall Kasus und Genus. Der Numerus ist ambig, daher steht an seiner Stelle eine Variable:

```
n(reporter,_,nominativ,maskulin)
```

Übungsaufgabe 4:
Erweitern Sie das obige Programm, so daß die zuletzt gezeigten informativeren Strukturen erzeugt werden.

Übungsaufgabe 5:
Überlegen Sie sich wieviele Lexikoneinträge für den Artikel "die" notwendig sind, um alle korrekten Merkmalskombinationen für Numerus, Genus und Kasus abzudecken und implementieren Sie sie analog zu den Einträgen oben.

Phrasenstrukturgrammatiken und Prologparsing

Die Syntax einer Sprache legt fest, welche Ausdrücke wohlgeformt, d.h. korrekt sind. Dies ist als Definition noch zu allgemein, da es z.B. auch semantische oder logische Korrektheits–bedingungen gibt. Der Wohlgeformtheitsbegriff bezieht sich in der Syntax auf die morpho-syntaktischen Kombinationsmöglichkeiten der Teilausdrücke in einer Sequenz. Zwei Begriffe sind hier vor allem (und speziell für das Deutsche) einschlägig: Stellung und Kongruenz.

In dem Satz

Peter hat Fritz getroffen.

legen die **Stellungsregeln** der deutschen Syntax fest, daß das Wort "hat" im Prinzip (d.h. in allen korrekten syntaktischen Kontexten) alle Positionen in diesem Satz einnehmen kann, außer derjenigen zwischen "Fritz" und "getroffen". Die **Kongruenzregeln** dagegen legen fest, daß es "hat" und nicht "haben" heißen muß. Natürlich gibt es eine große Zahl solcher Stellungs- und Kongruenzregeln, deshalb werden wir hier nur auf wenige Beispiele eingehen. Innerhalb unseres Grammatikprogramms nimmt die syntaktische Analyse die zentrale Rolle ein:

```
grammatikalische_analyse( Eingabe, LogischeRepräsentation):-
    morphologische_analyse(Grundformen, Vollformen),
    lexikalische_analyse(Eingabe, Getaggte_Eingabe),
    syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
    semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
    pragmatische_analyse(SemantischeRepräsentation, LogischeRepräsentation).
```

Die Teilaufgabe "syntaktische_analyse" hat die zweifache Funktion, sowohl die Korrektheit eines Satzes entsprechend deutscher Syntaxregeln festzustellen, als auch eine Strukturbeschreibung dieses Satzes zu liefern. Letztere Funktion ist sekundär, da die syntaktische Repräsentation als Seiteneffekt des Parsingvorganges geliefert wird. Die Hauptaufgabe, der wir uns nun widmen wollen, ist die Angabe und Implementierung grundlegender syntaktischer Regeln des Deutschen.

Die Aufgabe der syntaktischen Analyse definieren wir vorläufig so:

```
syntaktische_analyse(Eingabe, SyntaktischerBaum) :-
   s .
```

Das heißt, zunächst soll das Symbol "s" für Satz das Startsymbol für eine Grammatik sein, die wir später erweitern, so daß wir die Eingabe direkt an diese weiterleiten können. Als Standardbeschreibungsmittel für Grammatiken gelten sog. Phrasenstrukturregeln. Sie beschreiben, in welcher Weise eine Wortkette wie z.B. "max mag max" abgeleitet werden kann:

```
s --> np vp
np --> max
vp --> v np
v -->  mag
```

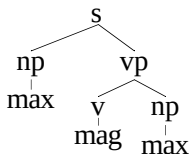
Bei dieser Grammatik gibt es beginnend mit dem Startsymbol s nur die folgende Ableitungsmöglichkeit:

s - np - **max** - vp -v - **mag** - np -**max**

Bei diesem Verfahren ersetzt man also das jeweilige Grammatiksymbol links vom Pfeil durch die Symbole rechts vom Pfeil. Dieser Prozeß endet für ein Symbol dann, wenn die Wortebene erreicht wird. Das Wort wird deshalb auch Terminal genannt. Wie man später sehen wird, ist es oft notwendig, den genauen Weg der Herleitung einer Wortkette zu kennen. Statt der obigen Darstellung verwendet man dann eine Klammerstruktur, auch Phrasemarker genannt:

```
s(np(max),  vp(v(mag),np(max))  )
```

Weniger platzsparend ist die folgende äquivalente Darstellung, die jedoch leichter überschaubar ist.



Von der Darstellung einer Phrasenstrukturgrammatik zu ihrer Implementierung in Prolog kommt man fast direkt. Man ersetzt den Ableitungspfeil der Ersetzungsregeln durch ":-", den Ableitungsoperator von Prolog, setzt ein Komma zwischen mehrfache Symbole im rechten Regelteil und schließt jede Regel mit Punkt ab. Terminale Symbole definiert man als Prologfakten:

```
s :- np , vp.
np  :- max.
vp  :- v, np.
v  :- mag.
mag.
max.
```

Um sich zu überzeugen, daß dieses Programm exakt die obige Ableitung erzeugt, muß man nur das Programm mit dem Startsymbol "s" aufrufen:

```
?- s.
yes
```

Wer mit dieser Antwort nicht zufrieden ist, der muß den "trace"- oder "debug"-Modus von Prolog bemühen, um den Weg nachzuvollziehen, den Prolog gegangen ist. Das ist natürlich eine sehr umständliche Art und Weise, um zu dem Phrasemarker zu gelangen, der ja das Ergebnis unserer Prozedur "syntaktische_analyse" sein soll. Auch haben wir bislang keine Möglichkeit, einen bestimmten Satz zu analysieren, wenn mehrere davon durch die Regeln erzeugt werden können, wie es bei der folgenden Grammatik der Fall ist:

```
s :- np , vp.
np  :- max.
np  :- musik.
vp  :- v, np.
v  :- mag.
max.
musik.
mag.
```

Obwohl wir nur eine Regel hinzugefügt haben, beschreibt diese Grammatik nun vier Sätze statt einen:

```
max mag max
max mag musik
musik mag max
musik mag musik
```

Angenommen uns interessiert nur, ob der zweite Satz durch die Grammatik analysierbar ist, dann müßten wir zunächst die Herleitung des ersten Satzes abwarten, um diese Information zu bekommen. Es liegt daher nahe, die Angabe, welchen Satz man analysiert haben möchte, in das Grammatikprogramm gleich mit einzubauen.

```
syntaktische_analyse(Eingabe, SyntaktischerBaum) :-
   s(Eingabe).
```

Der Programmaufruf sollte also z.B. lauten:

```
?- s( [max, mag, musik] ).
```

Die Idee ist also, den Satz z.B. als Liste dem Analyseprozeß zu unterwerfen, wobei man dafür sorgen muß, daß an jedem Punkt, an dem ein Terminal, also ein Wort abgeleitet wird, eine Überprüfung stattfindet, ob dieses Wort in der Eingabeliste ist. Da nun einfache Elementschaft nicht ausreicht, weil sonst z.B. auch

```
mag max musik
```

zulässig wäre, muß man sich auf eine bestimmte Reihenfolge der Ableitung eines Phrasensymbols zu seinen Terminalen einigen. In diesem Fall ist die Strategie, die von Prolog vorgegeben wird, die von links nach rechts, also wird zuerst innerhalb von "s" "np" und dann "vp" abgearbeitet. Alles was noch zu tun bleibt, ist dafür zu sorgen, daß jedes Symbol nur den Teil der Wortkette als Parameter erhält, den es für ihre Ableitung benötigt. Zu diesem Zweck brauchen wir etwas, was die Eingabekette in unterschiedliche Teile aufspaltet. Die Prozedur "append" ist eine geeignete Funktion, da sie eine Liste nicht nur zusammenfügt, sondern auch reversibel ihre verschiedenen Verkettungen erzeugt. Die beiden folgenden Aufrufe illustrieren das:

```
?- append( [max, mag], [musik], X).
X = [ max, mag, musik]
```

```
?- append(X , Y, [ max, mag, musik] )

X = [], Y = [max,mag,musik] ;
X = [max], Y = [mag,musik] ;
X = [max,mag], Y = [musik] ;
X = [max,mag,musik], Y = [] ;
```

Offensichtlich wäre die zweite Lösung auch eine Lösung für die Aufteilung unserer Eingabekette in einen Teil für "np" und einen für "vp", so daß unsere erste Grammatikregel mit Parameter für die Wortkette lauten würde[1]:

[1]Die Position der append-Prozedur innerhalb der Regel s ist dabei vollkommen beliebig und ändert nichts an der Korrektheit der Regel.

```
s( Wortkette ) :-
    append(WortketteNP, WortketteVP, Wortkette),
    np(WortketteNP),
    vp(WortketteVP).
```

Die Prozedur "append" zerlegt die Wortliste in ihren "np" und ihren "vp" Bestandteil, wobei dies ein blindes Verfahren ist, da "append" noch nichts über die korrekte grammatische Aufteilung weiß, sondern nur alle Möglichkeiten ausrechnet. Innerhalb der "np" wird dann zunächst

```
[]
```

weitergeleitet und scheitert, da die leere Liste kein zulässiges Wort ist. Der nächste Versuch ergibt dann

```
np( [max] )
```

und auch dieser scheitert, aber nur daran, daß das Terminal "max" nicht als Liste enthalten ist, sondern als Konstante, während "append" Listen als Parameter verlangt. Mit der entsprechenden Änderung in der Grammatik erhalten wir schließlich folgende Regeln:

```
s( Wortkette) :-
    append(WortketteNP, WortketteVP, Wortkette),
    np(WortketteNP),
    vp(WortketteVP).

vp( Wortkette) :-
    append(WortketteV, WortketteNP, Wortkette),
    v(WortketteV),
    np(WortketteNP).

np([max] ).
np([musik] ).
v([mag] ).
```

Dieses Programm funktioniert nun als Parser für unsere Grammatik, indem es die Eingabekette in die syntaktischen Symbole führt und sie abarbeitet, bis alle Wörter erkannt worden sind. Das Bemerkenswerte an Grammatiken, die in dieser Weise als Prologprogramme spezifiziert werden, ist, daß sie keinen prinzipiellen Unterschied machen zwischen einem Erkennungs- und einem Erzeugungsprozeß. D.h. der folgende Aufruf ist nur eine Möglichkeit das Programm zu verwenden:

```
?- s( [max, mag, musik] ).
```

Die andere Möglichkeit besteht darin, sich die Eigenschaft der **Reversibilität** der Grammatik zunutze zu machen, indem man die Eingabekette vollständig oder teilweise variabel läßt:

```
?- s( X).

?- s( [ErstesWort, mag, musik] ).

?- s( [max, ZweitesWort, musik] ).

?- s( [ErstesWort, ZweitesWort, DrittesWort] ).
```

Die Grammatik zählt alle Ersetzungen der unspezifizierten Worte auf, und zwar im jeweiligen Kontext der anderen Wörter der Eingabekette. Alle Grammatiken, die noch folgen, entsprechen im Prinzip dieser Funktionalität. Eventuelle Abweichungen von diesem Grundtypus werden nur aus Effizienz-, Darstellungs- oder Erweiterungsgründen vorgenommen, belassen aber dieses Kerngerüst einer Grammatik als Prologprogramm.

Betrachtet man unsere Grammatik nur als Menge von Regeln, die eine bestimmte syntaktische Struktur vorschreiben, so läßt sich nichts weiter verbessern. Die Grammatik als Parser dagegen ist optimierbar. So werden in dieser Formulierung mit append eine Reihe von Listenzerlegungen gemacht, die das Laufzeitverhalten des Programms beträchtlich verschlechtern. Dies liegt daran, daß zunächst arbiträre Zerlegungen erzeugt werden, die erst im weiteren Programmverlauf auf Korrektheit überprüft werden. Besser wäre es die Wortkette jeweils mitzuführen, um an der frühestmöglichen Stelle Eingabewörter mit Terminalen zu vergleichen. Wählt man diesen Ansatz braucht man allerdings zwei Variablen - eine für die in eine Regel eingehende Wortkette, und eine zweite für den Teil dieser Kette, der nicht innerhalb der jeweiligen Regel abgeleitet werden konnte und an die nächste Regel weitergeleitet wird. Für unsere Grammatik sieht diese Variante so aus:

```
s( WortketteRein WortketteRaus) :-
    np(WortketteRein , WortketteNpRaus),
    vp(WortketteNpRaus, WortketteRaus).

vp( WortketteRein WortketteRaus)) :-
    v(WortketteRein , WortketteVRaus),
    np(WortketteVRaus, WortketteRaus).
```

```
np([max | Wortkette], Wortkette).
np([musik | Wortkette], Wortkette).
v([mag | Wortkette], Wortkette).
```

Diese Verfahren unterscheidet sich vom vorherigen nur dadurch, daß nun keine unnötigen Zerlegungen mehr geschehen. Die Wortkette wird wie an einer Schnur durch die einzelnen Regeln geführt, und an der Stelle eines Terminals, an der ein gleichlautendes Wort am Anfang der Eingabekette steht, wird es aus der Kette herausgenommen und nur noch der Rest der Wortkette an die nächste Regel übergeben. Um den Parsingprozeß zu verdeutlichen, geben wir die obige Grammatik mit allen Belegungen für den Satz "max mag musik" an:

```
s( [max, mag, musik] []) :-
    np([max, mag, musik] , [mag, musik] ),
    vp([mag, musik], []).

vp( [mag, musik],[])) :-
    v([mag, musik], [musik]),
    np([musik], []).

np([max|mag, musik] , [mag, musik] ).
np([musik| []], []).
v([mag| [musik]], [musik]).
```

In der Regel "s" unterscheidet sich der Eingabeparameter vom Ausgabeparameter durch das Wort max, ebenso ist die Differenz zwischen Ein- und Ausgabe der Kategorie "v" innerhalb von "vp" das Wort "mag". Weil die Eingabe- und Ausgabeparameter der Regeln sich jeweils nur in dem von der Grammatik erkannten Wortkettenteil unterscheiden, nennt man diese Technik auch **Differenzlistentechnik**.

Wenn man sich die Regel genau ansieht, so stellt man fest, daß diese Technik, den jeweiligen Stand der Abarbeitung als Differenz zweier Parameter darzustellen, leicht schematisierbar ist. Es gilt das allgemeine Muster:

```
p( X1, Xk ) :-
    q( X1, X2 ),
    s( X2, X3 ),
    ...
    r( Xk-1, Xk).
```

Dabei können beliebig viele Aufrufe zwischen q und r liegen. Wichtig ist nur, daß der letzte Ausgabeparameter den Eingabeparameter der nächsten Regel darunter bildet. Aus Gründen der Darstellungsästhetik und der besseren Übersicht liegt es nun nahe einen Übersetzungsprozeß einzuschalten, der es erlaubt, eine Grammatikregel ohne Ein- und Ausgabeparameter zu schreiben, die Regel aber automatisch in eine Form mit Parametern überführt. Mit anderen Worten, gewünscht ist eine automatische Programmtransformation von Prologregeln der Art:

```
s :- np , vp.
```

zu Regeln mit Parametern:

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

Es ist relativ einfach ein Übersetzungsprogramm für diese Aufgabe zu schreiben. Tatsächlich hat sich sogar ein Standardverfahren etabliert, das praktisch in jeder Prologversion als eingebaute Funktion zur Verfügung steht. Will man einen Prologparser ohne die Differenzlistenparameter schreiben, sie aber direkt beim Einlesen übersetzt bekommen, so muß man die Regeln nur in einem bestimmte Format schreiben, damit Prolog diese erkennt, und sie von "normalen" Programmen unterscheiden kann. Dieses Format wird **DCG** (**Definite Clause Grammar**) genannt und geht auf eine Arbeit von Warren/Pereira (WARREN/ PEREIRA 1986) zurück. Das Format läßt sich am Beispiel unserer Grammatik darstellen:

```
s -->np , vp.
np  --> [max ].
np  --> [musik ].
vp  -->v, np.
v  --> [mag] .
```

Im wesentlichen muß der Pfeil "-->" für den Prologoperator ":-" stehen und die Terminalregeln rechts in eine Terminalliste übergehen. Das Übersetzungsprogramm geht davon aus, daß alle Symbole rechts vom Pfeil in Symbole übersetzt werden mit 2 Parametern für die Ein- und Ausgabe der Wortkette. Manchmal ist es allerdings notwendig, innerhalb einer Grammatikregel einer DCG noch weitere Prozeduren zu verwenden wie z.B. eine Prozedur, die die Kongruenz bestimmer Elemente überprüft. Damit der Übersetzer diese Prologaufrufe von den Grammatikregelteilen unterscheiden kann, schreibt man sie in geschweifte Klammern, also am obigen Beispiel:

```
s -->np(A) , vp(B), { kongruenz(A,B) }.
```

Von nun an werden wir diese DCG-Schreibweise verwenden, weil sie uns unnötige Schreibarbeit erspart. Nicht vergessen darf man, daß es sich bei den DCG-Grammatiken letztlich um Prologprogramme handelt, die zwar die gleichen Prozedurennamen wie die Grammatikregeln haben, aber eine um 2 erhöhte Stelligkeit besitzen. Der Aufruf des der DCG entsprechenden Regel in Prolog muß also (in unserem Beispiel) so lauten:

```
?- s( [max, mag, musik], []).
```

Damit müssen wir die Definition unseres Grundprädikates syntaktische_analyse nochmals kurz abändern in:

```
syntaktische_analyse(Eingabe, SyntaktischerBaum) :-
   s(Eingabe, [] ).
```

Als nächstes präsentieren wir eine kompakte Version eines Übersetzers von DCGs nach Prolog. Ausgespart sind "{}" Ausdrücke, sowie Cuts und Operatoren:

```
uebersetze( (P1 --> P2), (G1 :- G2) ):-
   linke_seite( P1, S0, S, G1 ),
   rechte_seite( P2, S0, S, G2 ).

linke_seite( P0, S0, S, G ):-
   fuege_hinzu( P0, S0, S, G ).

rechte_seite( (P1, P2) , S0, S, G ):-
   rechte_seite( P1, S0, S1, G1 ),
   rechte_seite( P2, S1, S, G2 ),
   und( G1, G2, G).
rechte_seite( P , S0, S, true ):-
   append( P, S, S0 ).

rechte_seite( P , S0, S, G ):-
fuege_hinzu( P, S0, S, G ).

fuege_hinzu( P, S0, S, G ):-
   atom(P),
   G =.. [ P, S0, S ].

und( true, G, G ).
und( G, true, G ).
und( G1, G2, (G1, G2) ).
```

In diesem Übersetzungsprogramm repäsentieren die Variablennamen, die mit "P" beginnen, Grammatiksymbole wie z.B. "np", bzw. die linken und rechten Knoten des Grammatikbaumes. Die Variablennamen, die mit "S" beginnen, stellen die Differenzliste von Variablen dar, die in der Prologversion der DCG als Eingabe- und Ausgabeparameter fungieren. Mit "G"

beginnende Variablen bezeichnen schließlich das Resultat der Überetzung: Die Prologregeln als Parser.

Bei diesem Übersetzer geht man am besten davon aus, daß die DCG-typischen Operatoren als Prologoperatoren definiert sind und daß die Regeln der DCG als Fakten mit einem Prädikat wie "clause" (bzw. einem Aufruf "X --> Y") aus der Datenbasis evaluierbar sind.

Nachdem wir nun ein Format für das Grammatikschreiben gefunden haben, ist es nur noch ein kleiner Schritt bis zu einer prinzipiellen Lösung für das Prädikat

```
syntaktische_analyse(Eingabe, SyntaktischerBaum)
```

Was noch fehlt ist die Erzeugung eines syntaktisches Baumes als Repräsentation der Analyse des jeweiligen Satzes. Dazu ändern wir den Aufruf des Satzprädikates "s" so ab, daß ein weiterer Parameter (die Variable "SyntaktischerBaum") für die Weiterleitung dieser Information zur Verfügung steht:

```
syntaktische_analyse(Eingabe, SyntaktischerBaum) :-
    s(SyntaktischerBaum, Eingabe, [] ).
```

Das Ergebnis des Aufrufs unserer Grammatik für den Satz "max mag musik" soll so aussehen, daß die natürlichsprachliche Eingabe als Entsprechung oder Resultat die syntaktische Beschreibung (als Klammerstruktur oder Baum) erhält:

```
syntaktische_analyse( [max , mag, musik ], s(np(max), vp(v(mag),np(max)) ))
```

Um eine solche Strukturrepräsentation zu bekommen, werden wir zunächst ein Konstruktionsverfahren vorschlagen, das über explizite Prozeduraufrufe den Baum zusammenbaut. Dieses Verfahren ist eigentlich unnötig umständlich, da alles was man über Unifikation konstruieren kann, auch ohne zusätzliche Prozeduraufrufe wie "baum" möglich ist. Auf der anderen Seite ist es wahrscheinlich instruktiver, zunächst das Konstruktionsverfahren als solches zu beschreiben und es in einem zweiten (Übungs-) Schritt zu vereinfachen:

```
s(S)-->np(NP) , vp(VP),          { baum(S,s,NP,VP) }.
np(NP)  --> [max ],              { baum(NP,np,max) }.
np(NP)  --> [musik ],            { baum(NP,np,musik) }.
vp(VP)  -->v(V), np(NP),         { baum(VP, vp,V,NP) }.
v(V)  --> [mag],                 { baum(V,v,mag) }.
```

```
baum(Var, Mutter, Tochter1, Tochter2):-    Var =.. [Mutter, Tochter1, Tochter2].
baum(Var, Mutter, Tochter):-               Var =.. [Mutter, Tochter].
```

In den Bezeichnungen halten wir uns an die übliche Praxis und teilen die Knoten eines Baumes in Mutter und Tochterknoten auf. Das Prädikat "baum" kodiert die Knotenrepräsentation aus den Bezeichnungen der Knoten der jeweiligen Regel, so wird der Variablen VP in der folgenden Regel:

```
vp(VP)  -->v(V), np(NP),        { baum(VP, vp,V,NP) }.
```

nach Ausführung der Prozedur "baum" ihre Baumrepräsentation zugewiesen:

```
VP = vp( V , NP)
```

Zu diesem Zeitpunkt sind die Töchterbäume aber bereits berechnet, da "v" und "np" schon ausgeführt wurden, so daß gilt:

```
V = v( mag)
NP = np( musik)
```

Mit den Einsetzungen der Werte der Variablen "V" und "NP" in den Term "vp(V,NP)" ist der Endzustand für "vp" erreicht und die Strukturinformation für diesen Knoten komplett:

```
VP = vp( v( mag) , np( musik))
```

Übungsaufgabe 6:
Erweitern Sie die obige Grammatik im DCG-Format, so daß auch der Satz "max lacht" abgeleitet werden kann. Welche inkorrekten Sätze werden dadurch erzeugt. Wie muß der Aufruf von s aussehen, damit alle Sätze erzeugt werden, die das Verb "lacht" enthalten und keine anderen ?

Übungsaufgabe 7:
Das Verfahren der Konstruktion eines syntaktischen Baumes kann effizienter gemacht werden, indem man die Terme direkt konstruiert, anstatt über eine Prozedur "baum". So läßt sich für die vp-Regel das Verfahren so abkürzen:

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

Schreiben Sie die anderen Regeln der obigen Grammatik analog dazu um.

Übungsaufgabe 8:
Das eben gezeigte Verfahren läßt sich automatisieren, indem man einen einfachen DCG-Übersetzer so abändert, daß er aus einer DCG nicht einen Prolog-Parser mit 2 sondern mit 3 zusätzlichen Argumenten erzeugt. Dieses dritte Argument steht für den syntaktischen Baum (der an seinen Terminalen auch die volle Merkmalsspezifikation haben kann). Versuchen Sie den einfachen DCG-Übersetzer entsprechend zu erweitern.

Übungsaufgabe 9:
Schreiben sie eine DCG für Zahlen zwischen 1 und 99, die in der Lage ist, zwischen Zahl und Zahlwort eine Verbindung herzustellen, so daß die folgenden Aufrufe möglich sind.

```
?-  zahl1_99(X,[neun,und,neunzig],[]).
X = 99 ;
no

?- zahl1_99(99,X,[]).
X = [neun,und,neunzig] ;
no
```

Mit der Verwendung von DCG-Regeln erspart man sich die explizite Angabe der Ein- und Ausgabeparameter für den Satz. Mit der gleichen Technik lassen sich aber noch weitere Parameter automatisch in die Regeln einfügen. So muß der zuvor dargestellte DCG-Compiler nur leicht erweitert werden, um aus DCG-Regeln Prologregeln zu erzeugen, die neben den beiden Parametern für die Differenzliste einen dritten Parameter für die Konstruktion des syntaktischen Baumes enthalten:

```
uebersetze( (P1 --> P2), (G1 :- G2) ):-
    linke_seite(B2, P1, S0, S, G1 ),
    rechte_seite( B, P1, P2, S0, S, G2 ),
    P1 =.. [Phrasensymbol|_],
    B2 =.. [Phrasensymbol|B].

linke_seite(B, P0, S0, S, G ):-
    fuege_hinzu(B, P0, S0, S, G ).

rechte_seite( B, P,  (P1, P2) , S0, S, G ):-
    rechte_seite( B1, P,  P1, S0, S1, G1 ),
    rechte_seite( B2, P, P2, S1, S, G2 ),
    append(B1,B2,B),
    und( G1, G2, G) , !.
rechte_seite([], _,{X} , S, S, X ).
rechte_seite([P1],P0,  P , S0, S, true ):-
    P0 =.. [_|T],
    P = [Wort],
    P1 = [Wort|T],
    append( P, S, S0 ).
rechte_seite([[]],_, [] , S, S, true ).
rechte_seite([P0],_, P , S0, S, G ):-
    fuege_hinzu(P0, P, S0, S, G ).

fuege_hinzu(B, P, S0, S, G ):-
    P =.. [Phrasensymbol|Argumente],
    not((Phrasensymbol = (,); Phrasensymbol = (.))),
    append( Argumente, [B, S0, S], AlleArgumente),
    G =.. [ Phrasensymbol|AlleArgumente].

und( true, G, G) .
und( G, true, G) .
und( G1, G2, (G1, G2)).
```

Dieser Compiler baut die Struktur für den syntaktischen Baum auf, indem analog zu dem Verfahren, das die Prozedur "baum" benützt, der jeweilige Knoten aus dem Funktor der linken Regelseite und den Funktoren der rechten Regelseite gebildet wird. So übersetzt der Compiler unsere Beispielsgrammatik:

```
s -->np , vp.
np  --> [max ].
np  --> [musik ].
vp  -->v, np.
v  --> [mag] .
```

in das folgende Format mit drei zusätzlichen Argumenten:

```
s(s(_71,_91),_34,_35) :-
            np(_71,_34,_67),
            vp(_91,_67,_35).

np(np([max]),[max|_35],_35) :-  true.
np(np([musik]),[musik|_35],_35) :- true.

vp(vp(_71,_91),_34,_35) :-
            v(_71,_34,_67),
            np(_91,_67,_35).

v(v([mag]),[mag|_35],_35) :-  true.
```

Das drittletzte Argument steuert den Aufbau des syntaktischen Baumes in der zuvor beschriebenen Weise. Der Vorteil eines solchen automatischen Verfahrens zur Erweiterung eines Parsers besteht in der Ersparung der Schreibarbeit sowie in einer verringerten Fehlerquote beim Schreiben. Der Nachteil bei der Benutzung eines Compilers besteht generell darin, daß man auf zwei Beschreibungsebenen arbeitet, was gewöhnungsbedürftig ist. Außerdem produzieren die meisten Prologversionen als Resultat ein Programm mit umbenannten Variablen, die aus nichtssagenden Bezeichnungen wie in unserem Beispiel bestehen. Aus diesen Gründen werden manche auf die Compilerbenützung verzichten und z.B. die Ausgabe des Compilers lieber von Hand vornehmen, was natürlich zu äquivalenten Programmen führt. In dieser Einführung werden wir später auf diesen Compiler zurückgreifen um unsere Grammatik mit einer Baumrepräsentation auszustatten, die wir als Schnittstelle zur Semantik benützen werden.

Wir wenden uns nun der Syntax zu. Zunächst geht es uns nur darum, die Komplexität dieses Bereichs herauszustellen. Dazu muß man sich vor Augen halten, daß der Computerlinguist vor die Aufgabe gestellt ist, alle korrekten sprachlichen Äußerungen anzugeben, die in einer computerlinguistischen Anwendung vorkommen können. Anders ausgedrückt heißt das, daß alle korrekten sprachlichen Äußerungen von dem sprachverstehenden System **akzeptiert**, alle übrigen aber zurückgewiesen werden müssen. Ein kleines Beispiel zeigt, daß diese Aufgabe nicht in einer simplen Weise zu lösen ist. Angenommen unser Inventar von unterschiedlichen Wörtern, auch Lexikon genannt, bestände aus nur 8 Einträgen. Weiter nehmen wir an, daß alle Sätze, die wir aus diesem Lexikon bilden, aus jeweils 5 Wörtern bestehen dürften. Dann wäre die Anzahl der unterschiedlichen Sätze, die möglich wären, 32768. Selbstverständlich wären die allermeisten dieser Sätze keine korrekten Sätze einer natürlichen Sprache. Die Frage, wie die Maschine diese von den korrekten unterscheiden soll, wäre aber damit noch nicht beantwortet. Die Antwort müßte natürlich lauten: über einen Filter von Regeln, die bestimmen, was ein korrekter Satz der betrachteten Sprache ist. Die folgende Minigrammatik ist ein typisches Beispiel für ein solches Regelwerk:

Syntax:

```
satz ->            nominal_teil   verbal_teil
nominal_teil->   artikel  nomen
verbal_teil ->     verb nominal_teil
```

Lexikon:

```
artikel ->        der | die
nomen ->          mann | maenner | frau | frauen
verb ->           liebt | lieben
```

Im Lexikonteil dieser Grammatik werden Wörter des Deutschen jeweils zu Gruppen zusammengefaßt. Das Kriterium für die Gruppierung ist hier nur dasjenige derselben Stellungsmöglichkeit. So werden "der" und "die" als Elemente einer Wortklasse verstanden, weil beide typischerweise vor einem Substantiv auftauchen. Die wesentliche Aussage dieser Grammatik besteht aber in der Festlegung der Reihenfolge von Wörtern verschiedenen Typs. Um diese Regeln lesen zu können, denke man sich den Pfeil als Abkürzung für "besteht aus". So besteht ein Satz aus einem Nominalteil und einem Verbalteil. Um dies vollständig zu verstehen, ersetzt man diese beiden Begriffe durch ihre Definitionen weiter unten. Dadurch erfährt man, daß ein Nominalteil aus einem Artikel und einem Nomen besteht und ein Verbalteil aus einem Verb und einem Nominalteil. Da wir letzteren Begriff bereits

weiter vorne aufgeschlüsselt hatten, müssen wir nun nur noch die lexikalischen Begriffe durch die entsprechenden Wörter ersetzen, also Artikel durch z.B. durch "der" oder "die", Nomen durch "mann" oder "frau" und Verb durch "lieben" (der senkrechte Strich in den Regeln des Lexikons steht für "oder"). Bei vollständiger Ersetzung aller grammatischen Begriffe gemäß der Regeln bleiben nun die Sätze übrig, die diese Grammatik zuläßt:

der mann liebt die frau
die frau liebt den mann
die männer lieben die frauen
etc.

Regeln wie die obigen werden auch Ersetzungsregeln genannt. Die Wirkung der Regeln wird vor allem an der Aufzählung von Sätzen erkennbar, die diese Grammatik nicht erlaubt:

der der der der
der mann mann mann
liebt der liebt die frau der mann die männer

Erlaubt werden von dieser Grammatik insgesamt nur noch 128 Sätze und damit sehr viel weniger als ohne syntaktische Regeln. Aber auch die allermeisten dieser Sätze sind noch ungrammatikalisch. Das liegt daran, daß unsere Grammatik noch zuwenig "Wissen" über die betrachtete Sprache enthält. So ist die Zusammenstellung von Wörtern bestimmten Typs in einem Satz nicht zufällig. Beispielsweise ist die beliebige Zusammenstellung von einem Artikel und einem Nomen nicht möglich:

die mann liebt die frau

Möglich sind nur solche Nominalteile, in denen Artikel und Nomen hinsichtlich des Geschlechts oder Genus übereinstimmen. Es muß also heißen:

der mann liebt die frau

Genausowenig berücksichtigt haben wir die Übereinstimmung von Artikel und Nomen bezüglich der Zahl. Während unsere Grammatik auch den folgenden Satz erlaubt:

der maenner lieben die frau

sollte eigentlich nur der Satz

die maenner lieben die frau

möglich sein. Auch eine dritte Kongruenz haben wir übersehen. Denn normalerweise müssen im Deutschen der einleitende Nominalteil eines Satzes und das folgende Verb ebenfalls in der Zahl identisch sein. Unsere Grammatik erlaubt aber auch den Satz:

```
die maenner liebt die frauen
```

Ohne an dieser Stelle auf eine alternative Formulierung der Grammatik von oben einzugehen, die auch diese Kongruenzen berücksichtigen würde, stellen wir nur fest, daß diese verbesserte Grammatik schließlich nur noch 16 Sätze zulassen würde (wobei eine vierte Inkongruenz noch nicht ausgeschlossen ist, die für weitere drei ungrammatikalische Sätze verantwortlich ist).

Als Resultat dieser Diskussion können wir daher festhalten, daß auch schon winzige Ausschnitte aus einer natürlichen Sprache genügend kombinatorische Komplexität enthalten, um einen simplen Beschreibungsweg zunichte zu machen. Ohne eine wie auch immer geartete Menge von Regeln, die die Zusammenstellung von Wörtern zu einem Satz so präzise wie möglich beschreiben, scheitert Sprachverarbeitung schon an der schieren Zahl von sprachlichen Ausdrücken, die sich aus dem Grundvokabular bilden lassen.

Bevor wir mit dem nächsten Kapitel tiefer in die Syntax einsteigen, wollen wir noch etwas Motivation betreiben. Syntax ist wie gerade gesehen eine zu komplexe Angelegenheit, um Fragmente zu entwickeln, deren Umfang nicht vorher bekannt ist. Wir wollen das Grammatik-Modul ja letztlich als eine **Schnittstelle** zu einer Datenbank benutzen und stellen diese nun zuerst vor und zeigen anhand einer Sammlung von Beispielsanfragen, welche Satztypen man braucht, um sie abfragen zu können. Als Datenbankmodell wählen wir ORBIS (GUENTHNER 1987). Dabei handelt es sich um Informationen über unser Sonnensystem. Gespeichert sind für jeden Himmelskörper sein Name, sein stellarer Typ, sein Durchmesser, sein Entdecker und der Himmelskörper, den er umkreist. Wir haben also Datenbankeinträge wie den folgenden:

```
db(uranus,planet,51800,herschel,sonne).
```

Die Datenbank ist als Sammlung von Prologfakten repräsentiert und besteht aus 54 Einträgen. Zwischen den Spalten einer Relation und den verschiedenen Relationen bestehen gewisse Beziehungen. So kann man nach dem Entdecker des Planeten Uranus fragen und nach dem Durchmesser desselben. Ebenso sind Fragen über die Gesamtheit aller Einträge denkbar wie die nach der Anzahl der Planeten, die ein Astronom entdeckt hat. Natürlich sind auch eine Menge von unsinnigen Sätzen über dieser Datenbank formulierbar, wie die Frage wieviele Durchmesser Uranus besitzt oder die Frage, welchen Mond Herschel umkreist. Diese Nonsens-Sätze auszuschließen, ist jedoch im Bereich der Syntax nicht möglich, sondern fällt in die Zuständigkeit der semantischen Restriktionen. Die Syntax sollte alle morphosyntaktischen und positionellen Beschränkungen der Sprache beschreiben, nicht

jedoch die Welt, über die in der Sprache kommuniziert wird. Die folgende Beispielssammlung enthält die beschriebenen unsinnigen Konstruktionen nur deshalb nicht, weil sie wenig motivierend wirken. Nichtsdestotrotz sollte man sich darüber im Klaren sein, daß eine Grammatik, die syntaktisch korrekte Sätze beschreibt, immer auch semantisch in der einen oder anderen Weise abweichende Sätze definiert.

Die folgenden Sätze wurden der deutschen Orbis-Variante entnommen und sind sozusagen der Maßstab für die weiteren Diskussionen, da wir versuchen wollen, uns bei der Abdeckung an dieser Satzsammlung zu orientieren.

wieviele Astronomen, die einen Planeten entdeckten, haben einen Mond entdeckt ?
welcher Astronom, der einen Planeten entdeckte, entdeckte einen Mond ?
ist Herschel der Astronom der einen Planeten entdeckte ?
ist der Durchmesser Jupiters kleiner als der Durchmesser von Uranus ?
ist der Durchmesser von Jupiter groesser als der Durchmesser von Uranus ?
welche Astronomen, die einen Planeten entdeckten, haben einen Mond entdeckt ?
welcher Mond, den Herschel entdeckte, umkreist den Uranus ?
welchen Planeten, den Oberon umkreist, entdeckte Herschel ?
entdeckte Herschel einen Planeten, den Titania umkreist ?
welchen Planeten, den ein Astronom entdeckte, umkreist ein Mond ?
ist Tombaugh der Astronom, der Pluto entdeckte ?
ist Uranus ein Planet, der die Sonne umkreist ?
welchen Planeten, dessen Durchmesser kleiner als 50000 km ist, entdeckte Herschel ?
ist Miranda der Mond, dessen Durchmesser kleiner als der Durchmesser Oberons ist ?
welcher Mond, dessen Durchmesser groesser als 47170 km ist, umkreist den Jupiter ?
welchen Planeten, dessen Durchmesser groesser als der Durchmesser von Uranus ist, umkreist ein Mond, den Kuiper entdeckte ?
welche Monde entdeckte Kuiper ?
ist der Durchmesser Neptuns groesser als der Durchmesser von Uranus ?
welchen Planeten umkreist Miranda ?
welchen Planeten, dessen Durchmesser groesser als 47169 km ist, umkreist ein Mond, den Kuiper entdeckte ?
hat Herschel Uranus entdeckt ?
welche Monde, deren Durchmesser kleiner als 3001 km sind, umkreisen den Pluto ?

Eine deutsche Grammatik

Verbstellung

Ausgehend von der Sammlung von Beispielssätzen werden wir versuchen, eine Grammatik zu erstellen, die sich schrittweise der Erfassung dieser Sätze nähert. Viel wichtiger als die komplette Abdeckung des anvisierten Fragments ist es aber, das jeweilige Regelwerk konsistent und korrekt zu erhalten. Mit anderen Worten, letztlich wird ein Großteil der Aufgabe darin bestehen, auch die Sätze zu "erfassen" , die wir **nicht** in der Grammatik haben wollen. Wir beginnen mit unserer Minigrammatik der Form:

```
s   --> np, vp.
np  --> [herschell].
np  --> [uranus].
vp  --> v, np.
v   --> [entdeckte].
```

Diese Grammatik läßt für Verben nur die einfache Zeitform wie z.B. "entdeckte" zu, das die zweite Position in der Phrasenabfolge einnimmt, also:

herschel **entdeckte** uranus

Wir werden nun eine Erweiterung durchführen, die es erlaubt, auch zusammengesetzte Zeitformen zu benützen:

herschel **hat** uranus **entdeckt**

Dazu müssen wir die Definition unsere Vp-Regel ändern, so daß die zweite Nominalphrase zwischen dem Hilfsverb "hat" und der infiniten Form "entdeckt" stehen kann. Damit aber eine einzige Verbalphrasenregel auch für den Verb-Zweit-Satz verwendet werden kann, erlauben wir, daß das Verb leer sein kann:

```
s --> np, vp.

np --> [herschel].
np --> [uranus].
vp --> aux, np, v.

v --> [entdeckt].
```

```
v --> [entdeckte].
v --> [].
aux --> [entdeckte].
aux --> [hat].
```

Mit dieser Erweiterung können wir tatsächlich beide Zeitformen des Verbs erzeugen. Wenn wir aber unser Startsymbol für die Erzeugung aller Sätze verwenden, sehen wir, daß wir leider auch eine Menge von syntaktisch inkorrekten Sätzen generieren (die semantisch inkorrekten lassen wir zunächst beiseite):

```
?- s(X,[]).
   X = [herschel,entdeckte,herschel,entdeckt] ;
   X = [herschel,entdeckte,herschel] ;
   X = [herschel,entdeckte,uranus,entdeckt] ;
   X = [herschel,entdeckte,uranus] ;
   X = [herschel,hat,herschel,entdeckt] ;
   X = [herschel,hat,herschel] ;
   X = [herschel,hat,uranus,entdeckt] ;
   X = [herschel,hat,uranus] ;
   X = [uranus,entdeckte,herschel,entdeckt] ;
   X = [uranus,entdeckte,herschel] ;
   X = [uranus,entdeckte,uranus,entdeckt] ;
   X = [uranus,entdeckte,uranus] ;
   X = [uranus,hat,herschel,entdeckt] ;
   X = [uranus,hat,herschel] ;
   X = [uranus,hat,uranus,entdeckt] ;
   X = [uranus,hat,uranus] ;
   no
```

Insbesondere muß man versuchen, die Kombination von einem finiten Verb wie "entdeckte" mit einer infiniten Form wie "entdeckt" zu vermeiden. Der Fehler liegt darin, daß wir zwischen einem echten Hilfsverb wie "hat" und einem Vollverb wie "entdeckte" nicht unterschieden haben. Dies holen wir nun nach, indem wir das Hilfsverb mit dem Merkmal ausstatten, daß es mit einem Vollverb am Ende des Satzes kombinieren kann, während dies für ein Vollverb ausgeschlossen ist.

```
s --> np, vp.

np --> [herschel].
np --> [uranus].
```

```
vp --> aux(Form), np, v(Form).

v(infinit) --> [entdeckt].
v([]) --> [].

aux([]) --> [entdeckte].
aux(infinit) --> [hat].
```

Wenn wir das Programm ausprobieren, sehen wir, daß nun nur noch syntaktisch korrekte Sätze erzeugt werden und zwar diejenigen, die wir wollen:

```
?- s(X,[]).
   X = [herschel,entdeckte,herschel] ;
   X = [herschel,entdeckte,uranus] ;
   X = [herschel,hat,herschel,entdeckt] ;
   X = [herschel,hat,uranus,entdeckt] ;
   X = [uranus,entdeckte,herschel] ;
   X = [uranus,entdeckte,uranus] ;
   X = [uranus,hat,herschel,entdeckt] ;
   X = [uranus,hat,uranus,entdeckt] ;
  no
```

Der Trick besteht also darin, dem Lexikoneintrag des Hilfsverbs die Information, daß ein infinites Verb folgen muß, als Merkmal (infinit) mitzugeben, während das finite Vollverb die Information ([]) erhält, daß kein Verb mehr folgen kann.

Mithilfe dieser Technik sollte es relativ einfach sein, auch weitere Verbpositionen zuzulassen, z.B. die Frageformen:

```
hat herschel uranus endeckt
entdeckte herschel uranus
```

Wir benötigen dafür in jedem Fall eine Regel, die die Stellung des Verbs vor einem Satz erlaubt:

```
s --> aux( _ ),  s.
```

Zunächst stellen wir fest, daß diese Regel auch eine Rekursion mit sich selbst erlaubt, also Sätze der Form:

```
hat hat hat ...
```

Zweitens gibt es noch nichts, das verhindert, daß die erste Satzregel, die mit Hilfe der zweiten Satzregel aufgerufen wird, an Verbzweit-Stelle ein Verb oder Hilfsverb generiert, also etwa:

```
hat herschel hat uranus entdeckt
hat herschel entdeckt uranus
```

Wir müssen also die Rekursion begrenzen, indem wir ein Merkmal mitgeben, das definiert, daß aus der zweiten Satzregel heraus nur die erste Satzregel aufgerufen werden kann. Außerdem müssen wir die Information, welcher Verbtyp an Verberst-Stelle realisiert wurde, an den Satz und die Verbalphrase darunter weitergeben. Die folgende revidierte Grammatik enthält all dies:

```
s(Form) --> np, vp(Form).
s(finit) --> aux(finit/Form), s(Form).

np --> [herschel].
np --> [uranus].

vp(Form) --> aux(Form/Form2), np, v(Form2).

v(infinit) --> [entdeckt].
v([]) --> [].

aux(finit/[]) --> [entdeckte].
aux(finit/infinit) --> [hat].
aux(X/X) --> [], {(X = infinit ; X = [])}.
```

Die gravierendste Änderung betrifft den Lexikoneintrag für Hilfsverben (aux). Dieser fungiert nunmehr nicht als einfacher Klassifikator, sondern als Schalter. Je nachdem, welche Verbform als erste im Satz auftritt, gibt dieser Schalter die notwendige Information an die Phrasen, die folgen, weiter. Wir greifen den Belegungszustand der Satzregel und der Verbalphraseregeln für den folgenden Satz heraus:

```
entdeckte herschel uranus
```

```
s(finit) --> aux(finit/[]), s([]).
s([]) --> np, vp([]).
vp([]) --> aux([]/[]), np, v([]).
```

Wir sehen, daß innerhalb der Verbalphrase die Verbaufrufe leer laufen, während für den Satz

```
hat herschel uranus entdeckt
```

```
s(finit) --> aux(finit/infinit), s(infinit).
s(infinit) --> np, vp(infinit).
vp(infinit) --> aux(infinit/infinit), np, v(infinit).
```

nur das aux innerhalb der Verbalphrase leerläuft, die letzte Verbstelle aber realisiert wird.

Das Programm erzeugt damit die folgenden Sätze:

```
?- s(finit,X,[]).
   X = [herschel,entdeckte,herschel] ;
   X = [herschel,entdeckte,uranus] ;
   X = [herschel,hat,herschel,entdeckt] ;
   X = [herschel,hat,uranus,entdeckt] ;
   X = [uranus,entdeckte,herschel] ;
   X = [uranus,entdeckte,uranus] ;
   X = [uranus,hat,herschel,entdeckt] ;
   X = [uranus,hat,uranus,entdeckt] ;
   X = [entdeckte,herschel,herschel] ;
   X = [entdeckte,herschel,uranus] ;
   X = [entdeckte,uranus,herschel] ;
   X = [entdeckte,uranus,uranus] ;
   X = [hat,herschel,herschel,entdeckt] ;
   X = [hat,herschel,uranus,entdeckt] ;
   X = [hat,uranus,herschel,entdeckt] ;
   X = [hat,uranus,uranus,entdeckt] ;
   no
```

Nun fehlt uns für das Deutsche nur noch ein wesentlicher Verbstellungstyp. Relativsätze und Nebensätze verlangen gewöhnlich, daß das finite Verb am Ende des Satzes stehen muß (wie "muß" in diesem Satz):

entdeckte ein astronom der uranus **entdeckte** oberon
hat ein astronom der uranus entdeckt **hat** oberon entdeckt

Um diese Konstruktionen zu ermöglichen, muß unsere Grammatik etwas stärker erweitert werden. Insbesondere benötigen wir die folgenden Kategorien:

- Nominalphrasen (**np**) die aus Artikel und Nomen bestehen wie "ein astronom"
- Ein Kategorie Relativpronomen (**rp**) für "der" in "der uranus entdeckte"
- Eine Kategorie Verbalgruppe (**vg**), die die Stellung infinites-finites Verb erlaubt wie in "der uranus entdeckt hat "
- Eine Kategorie Relativsatz (**rs**), die den untergeordneten Satz von einem Hauptsatz unterscheidet.

Schließlich definieren wir ein Hilfsprädikat (**p**), das zusammen mit dem Merkmal **rel** die Verbstellungskombinatorik regelt. Mit diesen Änderungen erhalten wir nun eine Grammatik, die im Prinzip alle Verbstellungsmöglichkeiten des Deutschen enthält:

```
s(Form) --> np, vp(Form).
s(finit) --> aux(finit/Form), s(Form).

np --> [herschel].
np --> [uranus].
np --> det, n, rs(rel).

det --> [der].
n --> [astronom].
rp --> [der].

vp(Form) --> aux(Form/Form2), np, vg(Form2).

vg(Form) --> v(Form2), aux(Form1), {p(Form,Form1,Form2)}.

v(infinit) --> [entdeckt].
v(finit) --> [entdeckte].
v([]) --> [].
```

```
aux(finit/[]) --> [entdeckte].
aux(finit/infinit) --> [hat].
aux(X/X) --> [], {(X = infinit ; X = [] ; X = rel)}.

rs(Form) --> rp, vp(Form).
rs(Form) --> [].

p(rel,finit/infinit,  infinit).
p(rel,[]/[], finit).
p(X,[]/[], X):-  (X = infinit ; X = [] ; X = finit).
```

Die Definition der Regel **rs** zeigt, daß Relativpronomen wie **der** in einem Relativsatz die gleiche Funktion einnehmen wie Nominalphrasen in einem normalen Hauptsatz. Elaborierte Grammatiktheorien behandeln deshalb Relativpronomen auch als Subklasse von Nominalphrasen und Relativsätze als Subklasse von Hauptsätzen. Wir belassen es der Einfachheit halber bei den speziellen Kategorien **rp** und **rs**.

Für die beiden Beispielssätze

entdeckte ein astronom **der uranus entdeckte** oberon
hat ein astronom **der uranus entdeckt hat** oberon entdeckt

sieht die Belegung der Kategorien rs, vp und vg wie folgt aus (die erste vg-Belegung bezieht sich auf den ersten, die zweite auf den zweiten Satz):

```
rs(rel) --> rp, vp(rel).
vp(rel) --> aux(rel/rel), np, vg(rel).
vg(rel) --> v(finit), aux([]/[]), {p(rel,[]/[], finit)}.
vg(rel) --> v(infinit), aux(finit/infinit,), {p(rel,finit/infinit,  infinit)}.
```

Den Beweis für die Korrektheit dieser Grammatik können wir nun jedoch nicht mehr ohne weiteres erbringen, indem wir wie zuvor die Grammatik generieren lassen. Mit der Hinzunahme von Relativsätzen haben wir eine rekursive Struktur in der Grammatik zugelassen, die bei freier Variable für die Wortkette unendlich viele Relativsätze zu konstruieren versucht, also:

hat ein astronom den ein astronom den ein astronom den ein astronom ....

Der Analysemodus funktioniert unverändert, wir müßten jedoch alle Sätze explizit angeben, die wir testen wollen, also auch alle möglichen inkorrekten. Eine pragmatische

Herangehensweise ist es, den Wortkettenparameter nicht vollkommen variabel zu lassen, sondern über die explizite Nennung der Positionen der Worte eine Längenbeschränkung einzuführen wie in:

```
?- s(finit,[hat,ein,astronom,X,Y,Z,K,U,L],[]).

?- s(finit,[hat,der,astronom,X,Y,Z,K,U,L],[]).

X = der, Y = uranus, Z = entdeckt, K = hat, U = uranus, L = entdeckt ;
X = der, Y = uranus, Z = entdeckte, K = der, U = astronom,L = entdeckt ;
X = der, Y = der, Z = astronom, K = entdeckte, U = uranus,L = entdeckt ;
X = der, Y = astronom, Z = der, K = uranus, U = entdeckte, L = entdeckt ;
no
```

Mit dieser Methode können wir die Teile der Grammatik, die neu hinzugekommen sind, untersuchen. Wir stellen fest, daß die Implementierung des Relativsatzes korrekt zu sein scheint. Außer acht lassen wir dabei die Tatsache, daß innerhalb der letzten drei generierten Sätze, die Kasusendungen des Artikels nicht korrekt sind. Dies ist darauf zurückzuführen, daß wir Kongruenz noch nicht behandelt haben und nicht eine Folge falscher Kodierung der Verbstellungsmöglichkeiten. Übrigens kann man natürlich die Relativsatzregel für das Testen der restlichen Grammatik auch einfach herausnehmen, da sie außer der Verbstellung keine anderen Strukturen generiert.

Unsere Grammatik verfügt mit dieser Erweiterung über alle Verbstellungsmöglichkeiten des Deutschen:

hat ein astronom uranus **entdeckt**
endeckte ein astronom uranus
ein astronom **hat** uranus **entdeckt**
ein astronom **entdeckte** uranus
entdeckte ein astronom der uranus **entdeckte** oberon
hat ein astronom der uranus **entdeckt hat** oberon entdeckt

Fragewörter

Bislang haben wir nur eine Möglichkeit für Fragesätze in unserer Grammatik: Ja/Nein-Fragen wie

hat ein astronom uranus **entdeckt**

Bei Datenbankenabfragen ist ein anderer Fragentyp aber viel relevanter, derjenige nach Werten wie:

welcher astronom hat uranus **entdeckt**

Die Implementierung dieser Sätze scheint syntaktisch nichts neues zu bringen, alles was erforderlich ist, ist eine neue Art von Artikel im Lexikon einzutragen. Also statt wie bisher Nominalphrasen der Form:

ein astronom

soll auch ein Frageartikel möglich sein:

welcher astronom

Leider würde ein Frageartikel in dieser allgemeinen Form dann in jeder Nominalphrase auftauchen dürfen und das würde auch die folgenden Sätze ermöglichen:

hat **welcher** astronom uranus entdeckt
endeckte **welcher** astronom uranus
entdeckte ein astronom der **welchen** planeten entdeckte uranus
hat ein astronom der uranus entdeckt hat **welchen** planeten entdeckt

D.h. die Kombination von Ja/Nein- und Wert-Fragen ergibt inkorrekte Sätze, die man ausschließen möchte. Offensichtlich gelingt dies, wenn man die Information, um welchen Verbstellungstyp es sich bei einem Satz jeweils handelt, an die Stelle weiterleitet, an der ein bestimmter Artikel realisiert wird. Mit anderen Worten, wir müssen in der Grammatik das Merkmal "Form" auch in die Nominalphrasen weiterleiten. Außerdem müssen wir die Artikel in Frageartikel und alle anderen teilen. Mit diesen Setzungen bekommen wir die folgende Grammatik, in der das Merkmal "finit" innerhalb der Kategorie "det" dafür sorgt, daß beliebige Artikel realisiert werden können, also auch Frageartikel, während die komplementären Werte von "Form" "infinit" und "[]" nur definite (def) und indefinite (indef) Artikel gestatten (Die entsprechenden Stellen sind in der Grammatik fett unterlegt).

Übungsaufgabe 10:
Erweitern Sie die umseitige Grammatik um Sätze der Form "wer hat uranus entdeckt" und "wen hat herschel entdeckt". Definieren Sie die Fragewörter "wer" und "wen" als Nominalphrasen.

Übungsaufgabe 10.1:
Erweitern Sie die Grammatik so, daß auch folgende verneinte Sätze möglich sind:

hat herschel uranus nicht **entdeckt**
endeckte herschel uranus nicht
herschel **hat** uranus nicht **entdeckt**
herschel **entdeckte** uranus nicht
entdeckte ein astronom der uranus nicht **entdeckte** oberon
hat ein astronom der uranus nicht **entdeckt hat** oberon entdeckt

Definieren Sie eine terminale Regel "neg":

```
neg --> [ nicht ].
```

und fügen Sie "neg" als Nonterminal in diejenige syntaktische Regel ein, die sich am besten dafür eignet.

Übungsaufgabe 10.2:

Mit den obigen Erweiterungen sind auch Sätze möglich, die Fragewörter und Negationen gleichzeitig enthalten:

wer **hat** uranus nicht **entdeckt**

Syntaktisch und semantisch ist an solchen Sätzen nichts inkorrekt. Unter pragmatischen Gesichtspunkten möchte man solche Kombinationen aus Fragewort und Negation aber vielleicht ausschließen. Implementieren Sie für die Grammatik diese Ausschlußbedingung, indem sie ein zusätzliches Merkmal verwenden, das über die Art von Nominalphrase (Fragewort/kein Fragewort) und über das Vorhandensein/Nichtvorhandensein einer Negation Auskunft gibt und ihre Realisierung regelt.

```
s(Form) --> np(Form), vp(Form).
s(finit) --> aux(finit/Form), s(Form).

vp(Form) --> aux(Form/Form2), np(Form2), vg(Form2).
vg(Form) --> v(Form2), aux(Form1), {p(Form,Form1,Form2)}.

rs(Form) --> rp, vp(Form).
rs(Form) --> [].

np(Form) --> det(Form), n, rs(rel).
np(_) --> en.

en --> [herschel].
en --> [uranus].

v(infinit) --> [entdeckt].
v(finit) -->[entdeckte].
v([]) --> [].

aux(finit/[]) --> v(finit).
aux(finit/infinit) --> [hat].
aux(X/X) --> [], {(X = infinit ; X = [] ; X = rel)}.

det(finit) --> dets(_).
det(Form) --> dets(F), {(Form = infinit; Form = []; Form = rel),(F = def; F = indef)}.

dets(def) --> [der].
dets(indef) --> [ein].
dets(frage) --> [welcher].

n --> [astronom].
rp --> [der].

p(rel,finit/infinit, infinit).
p(rel,[]/[], finit).
p(X,[]/[], X):- X = infinit ; X = [] .
```

Subkategorisierung

Die letzte Version unserer Grammatik ist noch immer nicht ganz fehlerfrei. Sie erzeugt Sätze, die zwar den Wortstellungsregeln des Deutschen entsprechen, aber gegen bestimmte Konfigurationsbestimmungen verstoßen, wie der folgende Satz:

hat **der astronom** der uranus entdeckte **der astronom** entdeckt

Hier tauchen die beiden Nominalphrasen des Hauptsatzes im gleichen Kasus auf, im Nominativ, während die korrekte Version des Satzes jeweils eine Nominalphrase im Nominativ und eine im Akkusativ verlangt:

hat **der astronom** der uranus entdeckte **den astronomen** entdeckt

Natürlich gibt es Sätze in denen auch zwei Nominativ-Nominalphrasen vorkommen dürfen wie in:

der mann der uranus entdeckte war **ein astronom**

Die jeweilige Kombination von Nominalphrasen im Satz hängt also vom Verb ab. Dies betrifft auch die Anzahl der Nominalphrasen. So gibt es eine Reihe von Verben mit verschiedenen Stelligkeiten bzw. verschiedener Valenz:

der mann schlief
der mann entdeckte **uranus**
der mann gab d**em planeten den Namen Uranus**

Die genaue Anzahl und die Form der Nominalphrasen läßt sich nur dann im Satz spezifizieren, wenn die entsprechenden Lexikoneinträge des Verbs diese Informationen enthalten, die auch Subkategorisierungsmuster heißen. Genauso müssen auch die Nominalphrasen bzw. ihre bestimmenden Elemente (Köpfe) die Nomen über gleichartige Informationen in den Lexikoneinträgen verfügen, also anstatt der bisherigen Einträge wie

```
v(finit) --> [entdeckte].
dets(indef) --> [ein].
n --> [astronom].
rp --> [der].
```

benötigen wir Einträge der Form:

```
v(finit,Subkat) --> [entdeckte],{Subkat = [[nom,sing,Gen1],[akk,Num2,Gen2]]}.
v(finit,Subkat) --> [entdeckten],{Subkat = [[nom,plu,Gen1],[akk,Num2,Gen2]]}.

dets(indef,[nom,sing,masc]) --> [ein].
dets(indef,[akk,sing,masc]) --> [einen].

n([nom,sing,masc]) --> [astronom].
n([akk,sing,masc]) --> [astronomen].

rp([nom,sing,masc]) --> [der].
rp([akk,sing,masc]) --> [den].
```

Beim Verb legen wir fest, daß "entdecken" zwei Nominalphrasen mit verschiedenem Kasus kombiniert. Außerdem soll die Nominalphrase im Nominativ im Singular sein, während dies für die Akkusativ-Nominalphrase offen ist, ebenso wie das Genus beider Nominalphrasen. Der entsprechende Eintrag für die Pluralform des Verbs legt fest, das die Nominativphrase auch im Plural stehen muß. Alles, was nun noch zu tun bleibt, ist sicherzustellen, daß diese Informationen aus dem Lexikon in die syntaktische Struktur an die richtigen Stellen gelangt. Wir greifen daher einige Regeln unserer Grammatik heraus

```
s(Form) -->
   np(Form),
   vp(Form).
vp(Form) -->
   aux(Form/Form2),
   np(Form2),
   vg(Form2).
```

```
vg(Form) -->
   v(Form2),
   aux(Form1),
   {p(Form,Form1,Form2)}.
np(Form) -->
   det(Form),
   n .
```

und modifizieren sie so, daß die zusätzlichen Informationen aus dem Lexikon zirkulieren können:

```
s(Form) -->
   np(Form,Komp1),
   vp(Form,Komp1).
vp(Form,Komp1) -->
   aux(Form/Form2,[Komp1,Komp2]),
   np(Form2,Komp2),
   vg(Form2,[Komp1,Komp2]).
```

```
vg(Form,Subkat) -->
   v(Form2,Subkat),
   aux(Form1,Subkat),
   {p(Form,Form1,Form2)}.
np(Form,Komp1) -->
   det(Form,Komp1),
   n(Komp1).
```

Über die Merkmale "Komp1" und "Komp2" für Komplement leiten wir die Nominalinformationen an alle Stellen in der Grammatik, an denen das Vollverb stehen kann. Dies ist notwendig, da die jeweilige Position des Vollverbs im voraus nicht bekannt ist. An den Verbpositionen gelangen diese Informationen in die lexikalischen Einträge der Verben und werden via Unifikation zur Übereinstimmung mit den dort festgelegten Nominalphrasenschemata gezwungen. Damit ist die Grammatik in der Lage, für jedes Verb die Anzahl und die Art seiner Komplemente zu fixieren. Da in Prolog die Unifikation der Termstrukturen nicht unter Kommutativität erfolgt, haben wir bisher jedoch nur die Reihenfolge Nominativ - Akkusativ festgelegt. Wir können also den zweiten der folgenden Sätze nicht erzeugen:

der astronom entdeckte den planeten
welchen planeten entdeckte der astronom

Dazu müssen wir innerhalb des Verbs die Aussage treffen, daß auch die umgekehrte Reihenfolge, also Akkusativ - Nominativ zulässig ist. Dies ist jedoch über ein Oder-Statement oder eine zweite Klausel leicht zu realisieren. Wir redefinieren also den Verbeintrag entsprechend. Um ihn dennoch übersichtlich zu halten, kennzeichnen wir die Reihenfolgebestimmung als Unterprogramm des Verbeintrags:

```
v(infinit,Subkat) --> [entdeckt],        {subkat(entdecken,sing,Subkat)}.
v(finit,Subkat) --> [entdeckte],         {subkat(entdecken,sing,Subkat)}.

subkat(entdecken,Numerus,Subkat):-
    Subkat = [[nom,Numerus,_],[akk,_,_]];
    Subkat = [[akk,_,_],[nom,Numerus,_]].
```

Auf diese Weise ist jede Reihenfolge der eintreffenden Komplementstrukturen zulässig. Dieses Prinzip, die genaue Konfiguration der syntaktischen Struktur eines Satzes innerhalb seines Hauptverbs zu definieren, ist als grundlegende Strategie z.B. in der HPSG (POLLARD/SAG 1987) vorhanden. Die Abfolge der Komplemente oder ihre lineare Präzedenz ist somit stark modularisiert, da die entsprechenden Definitionen vollkommen unabhängig von den eigentlichen Syntaxregeln vorgenommen werden können.

Die folgende Grammatik enthält die gerade besprochene Erweiterung der Überprüfung der Komplementstrukturen. Um diese Eigenschaft auch richtig umsetzen zu können, benötigen wir auch die entsprechenden morphologischen Varianten der jeweiligen Worte. Das heißt, daß nun jedes Wort, das in den Nominalbereich fällt, also Artikel, Nomen, Eigennamen, Relativpronomen, sowohl in seiner Nominativ-, als auch in der Akkusativform in der

Grammatik vorhanden ist. Dies gilt natürlich auch für Fälle, in denen der Kasusunterschied am Wort nicht realisiert ist, wie bei Eigennamen (en), die im Nominativ, Akkusativ und Dativ homonym sind. Eine Besonderheit stellt außerdem die Kongruenz des Relativpronomens mit dem Nomen dar, das es einleitet. Die Abhängigkeit zwischen Nomen und Relativpronomen besteht in der Wertegleichheit der Belegungen für die Merkmale Numerus und Genus. D.h. ist das Nomen im Singular und feminin, dann gilt dies auch für das Relativpronomen. Dagegen ist die Kasusbelegung vollkommen unabhängig, wie die folgenden Beispiele zeigen:

den planeten den herschel entdeckte umkreist ein mond
herschel entdeckte **den planeten der** die sonne umkreist

Die Kodierung der DCG-Regel für den Relativsatz berücksichtigt dies, indem vom Subkatmerkmal das erste Element, das den Wert für Kasus enthält, abgespalten und als anonyme Variable dargestellt wird. Der Rest des Subkatmerkmals (= Komp) enthält die Werte für Numerus und Genus, die an das Relativpronomen und an die Verbalphrase durch Unifikation weitergeleitet werden. Der Kasuswert, den das Relativpronomen einführt, wird ebenfalls an die Verbalphrase weitergegeben, innerhalb derer das gesamte Merkmalsbündel (= [K|Komp]) auf Elementschaft in einem Subkategorisierungsrahmen überprüft wird.

```
rs(Form,[_|Komp]) -->
        rp([K|Komp]),
         vp(Form,[K|Komp],_).
rs(Form,_) --> [].
```

Wir wollen die Gelegenheit der Grammatikerweiterung nutzen und eine kleine Verallgemeinerung einführen, die sowohl für die Eigenschaften der Grammatik selbst, als auch für die Semantikkomponente später nützlich ist. Alle Regeln der Grammatik sollen auf der rechten Seite in höchstens zwei Nichtterminalsymbole expandieren. Eine Grammatik mit der Einschränkung, in genau 2 Nichtterminalsymbole zu expandieren und der zusätzlichen, daß alle anderen Regeln terminale Regeln sein müssen, wird auch **Chomsky-Normalform** genannt. Um dies zu erreichen, müssen wir zwei Regeln umschreiben: die Verbalphrasenregel und die Nominalphrasenregel. Außerdem müssen wir zwei neue Symbole einführen: vp1 und n1. Somit wird aus:

```
vp(Form,Komp1,[Komp1,Komp2]) -->
    aux(Form/Form2,[Komp1,Komp2]),
    np(Form2,Komp2),
    vg(Form2,[Komp1,Komp2]).

np(Form,Komp1) -->
    det(Form,Komp1),
    n(Komp1),
    rs(rel,Komp1).
```

die revidierte Fassung:

```
vp(Form,Komp1,[Komp1,Komp2]) -->
    aux(Form/Form2,[Komp1,Komp2]),
    vp1(Form2,[Komp1,Komp2]).
vp1(Form2,[Komp1,Komp2]) -->
    np(Form2,Komp2),
    vg(Form2,[Komp1,Komp2]).

np(Form,Komp1) -->
    det(Form,Komp1),
    n1(Komp1).
n1(Komp) -->
    n(Komp),
    rs(rel,Komp).
```

Damit bekommt unsere Grammatik folgendes Aussehen:

```
s(Form,Subkat) -->
        np(Form,Komp1),
        vp(Form,Komp1,Subkat).

s(finit,_) -->
        aux(finit/Form,Subkat),
        s(Form,Subkat).

vp(Form,Komp1,[Komp1,Komp2]) -->
        aux(Form/Form2,[Komp1,Komp2]),
        vp1(Form2,[Komp1,Komp2]).

vp1(Form2,[Komp1,Komp2]) -->
        np(Form2,Komp2),
        vg(Form2,[Komp1,Komp2]).

vg(Form,Subkat) -->
        v(Form2,Subkat),
        aux(Form1,Subkat),
        {p(Form,Form1,Form2)}.

np(Form,Komp1) -->
        det(Form,Komp1),
        n1(Komp1),.
np(_,Komp1) -->
        en(Komp1).
n1(Komp) -->
        n(Komp),
        rs(rel,Komp).

en([Kasus,sing,masc]) -->
        [herschel],
        {(Kasus = nom; Kasus = akk)}.
en([Kasus,sing,masc]) -->
        [uranus],
        {(Kasus = nom; Kasus = akk)}.
rs(Form,[_|Komp]) -->
        rp([K|Komp]),
        vp(Form,[K|Komp],_).
rs(Form,_) --> [].

v(infinit,Subkat) -->
        [entdeckt],
        {subkat(entdecken,sing,Subkat)}.
v(finit,Subkat) -->
        [entdeckte],
        {subkat(entdecken,sing,Subkat)}.
v([],_) --> [].

subkat(entdecken,Numerus,Subkat):-
        Subkat = [[nom,Numerus,_],[akk,_,_]];
        Subkat = [[akk,_,_],[nom,Numerus,_]].

aux(finit/[],Subkat)  --> v(finit,Subkat).
aux(finit/infinit,_) --> [hat].
aux(X/X,_) --> [], {(X = infinit; X = []; X = rel)}.

det(finit,Komp1) --> dets(_,Komp1).
det(Form,Komp1) -->
        dets(F,Komp1),
        {(Form = infinit; Form = [];
        Form= rel),(F = def; F = indef)}.

dets(def,[nom,sing,masc]) --> [der].
dets(def,[akk,sing,masc]) --> [den].
dets(indef,[nom,sing,masc]) --> [ein].
dets(indef,[akk,sing,masc]) --> [einen].
dets(frage,[nom,sing,masc]) --> [welcher].
dets(frage,[akk,sing,masc]) --> [welchen].

n([nom,sing,masc]) --> [astronom].
n([akk,sing,masc]) --> [astronomen].

rp([nom,sing,masc]) --> [der].
rp([akk,sing,masc]) --> [den].

p(rel,finit/infinit,  infinit).
p(rel,[]/[],  finit).
p(X,[]/[], X):- X = infinit; X = []; X = finit.
```

Eine Folge der Änderungen, die wir vorgenommen haben, ist, daß die Grammatik für bestimmte Eingabeketten nicht mehr deterministisch ist. Das heißt, sie produziert mehr als eine Analyse für einen Eingabesatz. Dies ist z.B. der Fall im Bereich von Eigennamen, die sich hinsichtlich ihres Kasus morphologisch nicht unterscheiden und daher beide Merkmale (Nominativ und Akkusativ) annehmen können. Rufen wir die Grammatik mit einer Eingabe auf (wir verwenden dafür den DCG-Übersetzer aus Übungsaufgabe 8, der den syntaktischen Baum sowie die Merkmale der terminalen Elemente erzeugt), die zwei Eigennamen enthält, dann erhalten wir die folgenden Analysen:

```
?- syntaktische_analyse( [herschel,entdeckte,herschel], SyntaktischerBaum ).

SyntaktischerBaum =
        s(np(en([herschel,[nom,sing,masc]])),
          vp(aux(v([entdeckte,finit,
                    [[nom,sing,masc],
                     [akk,sing,masc]]])),
             vp1(np(en([herschel,[akk,sing,masc]])),
               vg(v([]),aux([])))));

SyntaktischerBaum =
        s(np(en([herschel,[akk,sing,masc]])),
          vp(aux(v([entdeckte,finit,
                    [[akk,sing,masc],
                     [nom,sing,masc]]])),
             vp1(np(en([herschel,[nom,sing,masc]])),
               vg(v([]),aux([])))))
```

Die erste Analyse entspricht der Reihenfolge Nominativ - Akkusativ für den Eigennamen "herschel". Dies ist auch in der Belegung der Kasusabfolge des Verbs abzulesen. Die zweite Analyse ist die Umkehrung dieser Verhältnisse: der erste Eigenamen wird als Akkusativ und der zweite als Nominativ interpretiert. Tatsächlich sind beide Sätze möglich. Der erste repräsentiert die normale Reihenfolge Subjekt-Verb-Objekt, während der zweite die Folge Objekt-Verb-Subjekt produziert. Diesen zwei verschiedenen syntaktischen Analysen des gleichen Oberflächensatzes entsprechen später in der Semantik zwei unterschiedliche Interpretationen dieses Satzes. Es gibt allerdings auch Mehrfachanalysen in der Syntax, deren Ursache rein technischer Natur ist, und die deshalb semantisch auch nur eine Interpretation haben. Diese beiden Quellen syntaktischer Mehrfachanalysen gilt es daher strikt zu unterscheiden, bzw. die Analysen, die nicht wirklich mehrdeutig sind, auszuschalten.

Extraponierte Relativsätze

Das Fragment läßt bisher keinerlei Varianten bei der Stellung von Relativsätzen zu. Die einzige Stelle einen Relativsatz einzuführen, ist die nach dem Nomen, das ihn bezüglich Numerus und Genus bestimmt. Dies ist zwar eine mögliche Position, aber nicht unbedingt die natürlichere von den folgenden beiden:

der astronom **der den planeten entdeckte** entdeckte herschel
der astronom entdeckte herschel **der den planeten entdeckte**

Der zweite Satz beschreibt eine Stellung nach dem Ende des Hauptsatzes. In der syntaktischen Feldertheorie wird diese Position deshalb auch "**Nachfeld**" genannt. Da nun aber der Relativsatz jeweils nur eine und nicht beide Positionen zugleich besetzen kann, und die Stellung beim Nomen als Grundstellung angesehen wird, bezeichnet man die Nachfeldstellung als "**Extraposition**" oder Verschiebung des Relativsatzes. Will man unsere Grammatik um diese Stellungsvariante erweitern, dann gibt es zwei Möglichkeiten. Entweder man beschreibt einen Satz als aus Satz und Nachfeld bestehend:

```
s --> s, nachfeld.
```

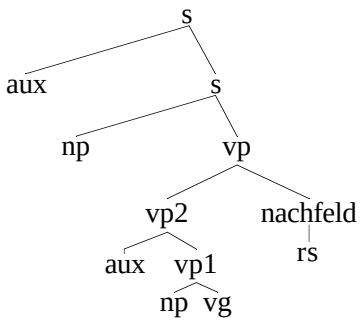
Der Nachteil dieser Definition besteht in der Ineffizienz der Abarbeitung, da wir dann 3 Satzregeln haben, die alle von dem s-Symbol der rechten Regelseite ausprobiert werden. Außerdem müssen wir uns dann, da es sich um eine linksrekursive Regel handelt, mit dem Problem der Nichtterminierung auseinandersetzen (das später behandelt wird). Aus diesem Grund wählen wir die zweite Möglichkeit, das Nachfeld als Erweiterung innerhalb der Verbalphrase (vp) mit dem Zusatzsymbol vp2 aufzufassen. Wir benennen also die bisherige vp in vp2 um, und definieren vp neu als aus vp2 und nachfeld bestehend. Diese Position wird auch **Adjunktion** an die Verbalphrase genannt:

```
vp2--> aux, vp1
vp --> vp2, nachfeld.
```

Im Prinzip genügt nun die Angabe einer Regel, die den Relativsatz als Nachfeldbesetzung beschreibt und einer weiteren, die auch die Nichtbesetzung erlaubt:

```
nachfeld--> rs .
nachfeld--> [].
```

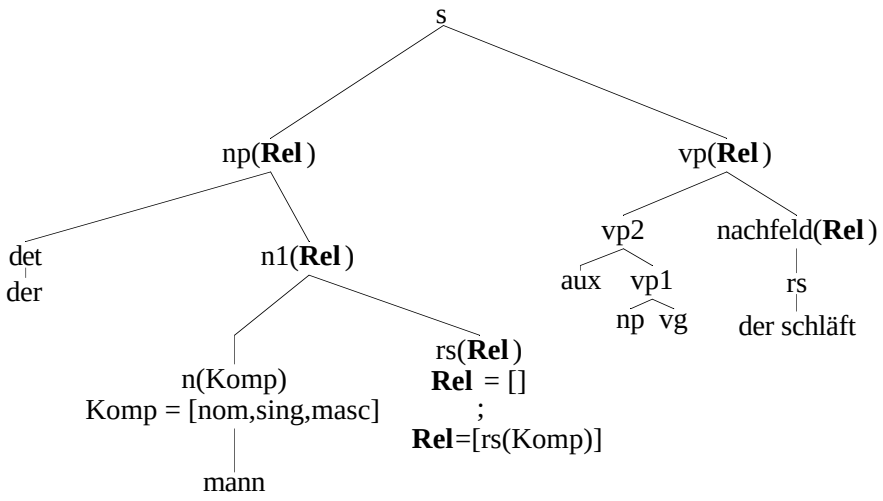
Ein allgemeines Schema der Satzstruktur unserer Grammatik ist nun das folgende:



Leider ist die Grammatik damit noch unterspezifiziert, da sie auch Relativsatzanschlüsse erlauben würde, die mit keinem der im Hauptsatz vorkommenden Nomen kongruieren. Angenommen, wir hätten ein femines Relativpronomen "die" im Lexikon, dann wäre auch der folgende Satz regelkonform (aber nicht korrekt):

der astronom entdeckte herschel **die den planeten entdeckte**

Die Kongruenzinformation (Numerus und Genus) zwischen Kopfnomen und seinem Relativpronomen muß wie bei der nichtextraponierten Stellung auch im Nachfeld verfügbar sein. Das heißt wir müssen ein zusätzliches Merkmal einführen, das wir aus der Position innerhalb der Nominalphrase in das Nachfeld weiterleiten. Der folgende Syntaxbaum zeigt den Informationsweg des Kongruenzmerkmals "Rel" des Kopfnomens eines extraponierten Relativsatzes:



"Rel" läuft entweder leer (Rel = []), dann darf im Nachfeld kein Relativsatz realisiert sein. Oder "Rel" wird mit den morphologischen Werten des Nomens belegt. In diesem Fall werden diese an das Nachfeld weitergeleitet und bilden die Kongruenzbasis des Relativpronomens des folgenden Relativsatzes.

Die Implementierung des extraponierten Relativsatzes folgt diesem Muster, das aber noch etwas verallgemeinert wird. Da wir bei transitiven Verben die Möglichkeit zweier Relativsätze und damit auch von zwei extraponierten Relativsätzen haben, müssen wir ihre Verschiebung einschränken, um stilistisch schlechte Sätze zu unterbinden.

der astronom hat einen planeten entdeckt der die sonne umkreist der uranus entdeckte

So darf jeweils nur ein Relativsatz bewegt werden. Dies wird mit der Konstruktion einer Rel-Liste erreicht, aus der im Nachfeld jeweils nur ein Element herausgegriffen (per "member") werden darf. Hier folgen nun die veränderten Regeln der Grammatik:

```
s(Form,Subkat) -->
        np(Form,Komp1,Rel),
        vp(Form,Komp1,Subkat,Rel).

vp(Form,Komp1,[Komp1,Komp2],Rel1) -->
        vp2(Form,[Komp1,Komp2], Rel2 ),
        { append(Rel1,Rel2,Rel) },
        nachfeld(Rel).

vp2(Form,Subkat,Rel) -->
        aux(Form/Form2,Subkat),
        vp1(Form2,Subkat, Rel ).

  vp1(Form,[Komp1,Komp2],Rel) -->
          np(Form,Komp2,Rel),
          vg(Form,[Komp1,Komp2]).

  np(_,Komp1,[]) --> en(Komp1).
```

```
np(Form,Komp1,Rel) -->
        det(Form,Komp1),
        n1(Komp1,Rel).

n1(Komp1,Rel) -->
        n(Komp1),
        rs(rel,Komp1,Rel).

rs(Form,[_|Komp],[]) -->
        rp([K|Komp]),
        vp(Form,[K|Komp],_,[]).

rs(Form,Komp1,[ rs(Komp1 )]) --> [].

nachfeld([]) --> [].
nachfeld(Rel) -->
        { member(rs(Komp),Rel) },
        rs(rel,Komp,_).
```

Die Extraposition des Relativsatzes ist eine Quelle möglicher syntaktischer Mehrdeutigkeit, da sich der Satz im Nachfeld auf verschiedene Antezedenten oder Vorderglieder beziehen kann. So ist der folgende Satz ambig zwischen den Lesarten Adjunktion des Relativsatzes an das Nominalsubjekt und an das Akkusativobjekt:

ein astronom hat einen astronomen entdeckt den herschel entdeckte

Besonders gut wird diese Ambiguität an der syntaktischen Struktur deutlich, wenn wir die Grammatik mit dieser Eingabe aufrufen:

```
?- syntaktische_analyse([ein,astronom,hat,einen,astronomen,
                         entdeckt,den,herschel,entdeckte],
                        SyntaktischerBaum ).


SyntaktischerBaum =

    s(np(det(dets([ein,indef,[nom,sing,masc]])),
        n1(n([astronom,[nom,sing,masc]]),rs([])),
        vp(vp2(aux([hat,finit / infinit,[nom,sing,masc],[akk,sing,masc]]]),
            vp1(np(det(dets([einen,indef,akk,sing,masc]])),
                n1(n([astronomen, [akk,sing,masc]]),rs([])),
                vg(v([entdeckt,infinit,[[nom,sing,masc],[akk,sing,masc]]]),aux([])))),
        nachfeld(rs(rp([den,[akk,sing,masc]]),
                vp(vp2(aux([]),vp1(np(en([herschel,[nom,sing,masc]])),
                        vg(v([entdeckte,finit,[[akk,sing,masc],nom,sing,masc]]]),aux([])))),
                nachfeld([])))))) ;


SyntaktischerBaum =

    s(np(det(dets([ein,indef,[nom,sing,masc]])),
        n1(n([astronom,[nom,sing,masc]]),rs([])),
        vp(vp2(aux([hat,finit / infinit,[[nom,sing,masc],[akk,sing,masc]]]),
            vp1(np(det(dets([einen,indef,akk,sing,masc]])),
                n1(n([astronomen, [akk,sing,masc]]),rs([])),
                vg(v([entdeckt,infinit,[[nom,sing,masc],[akk,sing,masc]]]),aux([])))),
        nachfeld(rs(rp([den,[akk,sing,masc]]),
                vp(vp2(aux([]),vp1(np(en([herschel,[nom,sing,masc]])),
                        vg(v([entdeckte,init,[[akk,sing,masc],[nom,sing,masc]]]),aux([])))),
                nachfeld([])))))
```

Die fett unterlegten Merkmalsbündel zeigen an, daß sich der Relativsatz in der ersten Analyse auf das Nominativsubjekt, in der zweiten auf das Akkusativobjekt bezieht.

Übungsaufgabe 11:
Die Grammatik erlaubt bislang nur zweistellige/transitive Verben wie in dem Satz

herschel hat uranus entdeckt

Überlegen Sie wie man auch dreistellige/ditransitive Verben wie "geben" in die Grammatik einführen kann, so daß auch Sätze wie

herschel hat uranus den namen gegeben

analysiert werden können.

Übungsaufgabe 12.1:

Die Art und Weise wie die Extraposition des Relativsatzes in das Nachfeld definiert wurde, ist für die beschriebenen Sätze korrekt. Dennoch hat diese Änderung eine unerwünschte Auswirkung auf unsere bisherigen Sätze ohne Relativsatz. So bekommt man für den Satz

der astronom hat den astronomen entdeckt

nunmehr zwei Analyseergebnisse anstatt wie bisher eines, die von ihrer Baumstruktur her jedoch vollkommen äquivalent sind. Versuchen Sie herauszufinden, wie es zu den beiden Analysen kommt und warum sie sich in ihrem syntaktischen Baum nicht unterscheiden. Ein Tip dazu: Der analoge Satz

herschel hat den astronomen entdeckt

bringt wie zuvor nur eine Analyse. Verwenden Sie den Debug-Modus.

Übungsaufgabe 12.2:

Die syntaktischen Bäume, die die Grammatik generiert, sind im Durchschnitt zu redundant, da jeder leere Knoten in der Struktur auftaucht. Schreiben Sie ein einfaches Übersetzungsprogramm, das alle leeren Knoten eliminiert.

Wir kommen nun zu einigen Verallgemeinerungen bzw. Korrekturen der Grammatik. So ist z.B. die Art und Weise wie die Extraposition des Relativsatzes in das Nachfeld definiert wurde, für die beschriebenen Sätze zwar korrekt. Dennoch hat diese Änderung eine unerwünschte Auswirkung auf unsere bisherigen Sätze ohne Relativsatz. Der folgende Hauptsatz

```
der astronom hat den astronomen entdeckt
```

kann nun auf zwei verschiedene Weisen analysiert werden, obwohl die resultierenden syntaktischen Bäume identisch sind. Dies liegt an der Definition der zweiten Regel "nachfeld", die den Relativsatz über "rs" aufruft. Die Definition der Regel "rs" erlaubt jedoch auch eine leere Ableitung, so daß immer wenn die Extrapositionsliste ein Element enthält, die zweite Relativsatzregel leerlaufen kann. Das heißt, daß für n Elemente auf der Extrapositionsliste n Analyseergebnisse produziert werden. Eine Möglichkeit diese unerwünschten Analysen zu vermeiden, besteht darin, den Aufruf von "rs" im Nachfeld nicht leerlaufen zu lassen, wenn sich Elemente auf der Extrapositionsliste befinden und wenn die Auswahl der Elemente per "member" geschieht. Zu diesem Zweck definieren wir die beiden Nachfeldregeln wie folgt um:

```
rs(Form,[_|Komp],[]) --> rp([K|Komp]), vp(Form,[K|Komp],_,[]).
rs(Form,Komp1, [rs(Komp1)]) --> [].

nachfeld([_|_]) --> [].
nachfeld(Rel)  --> { member(rs([X|Y]),Rel) }, rs(rel, [X|Y], []).
```

Der Aufruf von "rs" ist enthält als zweites Argument eine nicht leere Liste "[X|Y]", was verhindert, daß es in Regel "rs" Nummer 2 mit dem dritten Argument unifiziert, und als drittes Argument die leere Liste. Damit kann dieser Aufruf nur mit der nicht leeren Relativsatzregel matchen.

Die nächste Änderung betrifft die Beseitigung des expliziten "append" Aufrufs zur Verkettung von Elementen der Extrapositionsliste. Dazu konstruieren wir diese Liste direkt. Wir redefinieren die Regel der Form

```
vp(Form,Komp1,[Komp1,Komp2],Rel1) -->
        vp2(Form,[Komp1,Komp2], Rel2 ),
        { append(Rel1,Rel2,Rel) },nachfeld(Rel).
```

um in

```
vp(Form,Komp1,[Komp1,Komp2],Rel1) -->
        vp2(Form,[Komp1,Komp2], Rel2),
        nachfeld([Rel1|Rel2]).
```

Auf diese Weise erspart man sich "append", muß aber auf der anderen Seite in Kauf nehmen, daß leere Listen in die Extrapositionsliste aufgenommen werden, wenn kein Relativsatz möglich ist. Aus diesem Grunde haben wir bei der gerade vorgenommenen Definition der zweiten Nachfeldregel auch dafür gesorgt, daß das zweite Argument im rs-Aufruf keine solche leere Liste sein kann. Daher mußte auch die erste Nachfeldregel geändert werden, um ein Leerlaufen zu erlauben, auch wenn eine nur aus technischen Gründen in der Extrapositionsliste befindliche leere Liste vorhanden ist. Die Grammatik erlaubt nur die Analyse zweistelliger Verben wie "entdecken". Um auch andere Stelligkeiten, wie z.B. beim dreiwertigen Verb "geben" zu erlauben, benötigt man zwei Dinge:

- eine Subkategorisierungsliste mit beliebig vielen Elementen
- eine rekursive Kategorie für Nominalphrasen

Diese Änderungen sind im Vergleich zwischen bisherigen und neuen Regeln leicht darstellbar. Die Grammatik bisher hat (auszugsweise) das folgende Aussehen gehabt:

```
s(Form,Subkat) -->
        np(Form,Komp1,Rel),
        vp(Form,Komp1,Subkat,Rel).
vp(Form,Komp1,[Komp1,Komp2],Rel1) -->
        vp2(Form,[Komp1,Komp2], Rel2 ),
        nachfeld([Rel1|Rel2])).

vp2(Form,Subkat,Rel) -->
    aux(Form/Form2,Subkat),
    vp1(Form2,Subkat, Rel ).
vp1(Form,[Komp1,Komp2],Rel) -->
    np(Form,Komp2,Rel),
    vg(Form,[Komp1,Komp2]).
```

Die Subkategorisierungsliste war von vorneherein zweistellig und sammelte in "s" und in "vp1" die beiden Nps auf. Nach Umformulierung der Regeln ist die Subkategorisierungsliste nunmehr nach hinten offen und nimmt beliebig viele Elemente auf.

```
s(Form,[Komp1|Rest]) -->
    np(Form,Komp1,Rel),
    vp(Form,[Komp1|Rest],Rel).
vp(Form,Komp,Rel1) -->
    vp2(Form,Komp, Rel2 ),
    nachfeld([Rel1|Rel2]).

vp2(Form,Subkat,Rel) -->
    aux(Form/Form2,Subkat),
    vp1(Form2,Subkat, Rel ).
vp1(Form,Komp,Rel) -->
    nprec(Form,Komp,Rel),
    vg(Form,Komp).
```

Ausgenutzt wird diese Fähigkeit von der neuen Kategorie "nprec", die rekursiv die normale Nominalphrase aufruft, und dabei neue Subkatelemente in die Liste aufnimmt:

```
nprec(_,[],[]) --> [].
nprec(Form,[Subkat|SubkatRest],[Extrapos|ExtraposRest]) -->
        np(Form,Subkat,Extrapos),
        nprec(Form,SubkatRest,ExtraposRest).
```

Präpositionalphrasen

Der Sprachumfang der Grammatik baut noch immer auf sehr wenigen Grundstrukturen auf. Eine wichtige Konstruktion, die bisher noch nicht enthalten ist, stellt die Präpositionalphrase dar. In den folgenden Sätzen

besitzt uranus einen durchmesser **von 90000 km**
der durchmesser **von uranus** ist 90000 km
entdeckte herschel einen mond **von uranus**
entdeckte herschel den mond **von einem planeten**

sehen wir die Präpositionalphrase in ihrer Funktion als einschränkende Beschreibung des Nomens, auf das sie folgt. Semantisch ist sie deshalb dem Relativsatz verwandt. Entsprechend ordnen wir die Präpositionalphrase als Alternativkonstruktion zum Relativsatz ein, etwa in der Form, daß wir die Fortsetzung eines Nomens entweder als Relativsatz oder als Präpositionalphrase definieren:

```
n1(Komp1,Rel) -->  n(Komp1), radj( Komp1 , Rel).
radj( Komp1 , Rel) --> rs(rel,Komp1,Rel).
radj( Komp , Rel ) --> pp(Rel).
```

Die Bezeichnung "radj" steht für rechtes Adjunkt, also die rechte Expansionsmöglichkeit des Nominalknotens "n1". Das Merkmal "Rel" geben wir weiter um eine mögliche Verschiebung eines PP-internen Relativsatzes zu erlauben. Die interne syntaktische Konfiguration einer Präpositionalphrase ist nach den obigen Beispielen, die einer Kombination einer Präposition und einer Nominalphrase (wir möchten den Ausdruck "90000 km" als Spezialfall einer Nominalphrase betrachten). Die syntaktische Regel lautet also:

```
pp(Rel)  -->  prep(Komp,Form), np(Form, Komp , Rel).
```

Somit muß die Präposition nur noch im Lexikon eingetragen werden, um die Beschreibung einer Präpositionalphrase zu vervollständigen. Allerdings müssen wir darauf achten, daß Präpositionalphrasen ihren Kasus nicht von dem Nomen der darüberliegenden Nominalphrase erhalten, sondern von der Präposition selbst:

```
prep([ dat, _,_], [])  -->  [ von ].
```

Damit führen wir einen weiteren Kasus, nämlich Dativ ("dat"), in die Grammatik ein, und müssen das Lexikon dementsprechend auffüllen. Außerdem legen wir innerhalb der Präposition fest, welche Art von Artikeln die folgende Nominalphrase enthalten darf. Die Festlegung "[]" dient dazu, die ersten zwei der drei folgenden Sätze zu erlauben und den letzten zu verhindern, in der Art und Weise wie wir das schon für Fragewörter beschrieben hatten:

entdeckte herschel den mond von **einem** planeten
entdeckte herschel den mond von **dem** planeten
entdeckte herschel den mond von **welchem** planeten

Neben der Anreicherung der Grammatik ist es uns bereits wieder gelungen, eine zusätzliche Ambiguität einzuführen. Aus der Tatsache, daß sich aus der Position innerhalb der Nominalphrase der Präpositionalphrase ein Relativsatz ins Nachfeld verschieben läßt, folgt, daß die Zuordnung der extraponierten Phrase prinzipiell mehrdeutig ist:

hat herschel den mond von einem planeten entdeckt **den** ein astronom entdeckte

Sie kann sich entweder auf das Nomen der Präpositionalphrase oder auf das Nomen der Nominalphrase darüber beziehen, also

hat herschel den mond von einem planeten entdeckt den ein astronom entdeckte
hat herschel den mond von einem planeten entdeckt den ein astronom entdeckte

Da die Einbettungstiefe beliebig ist, kann das Nomen auf das der extraponierte Relativsatz Bezug nimmt, auch beliebig tief innerhalb der Abfolge von ineinander verschachtelter Präpositionalphrasen stehen. Übrigens ist diese Definition der Präpositionalphrase als Adjunktion zu einem Nomen aus den gleichen Gründen rekursiv wie schon der Relativsatz. Auch hier kann jedes Nomen selbst wieder Auslöser für eine weitere PP-Adjunktion sein, so daß unendlich viel verschachtelte Konstruktionen möglich sind:

hat herschel den mond von einem planeten von einer sonne von einem ...

Auch hier muß man beim Generieren darauf achten, diese Regel entweder herauszunehmen oder die Eingabekette hinsichtlich ihrer Länge zu beschränken.

Vergleiche (Komparativ)

Für die vollständige syntaktische Abdeckung der Satzsammlung, die wir am Anfang des Syntaxkapitels vorgestellt haben, fehlt noch eine wichtige Konstruktionsart. Wenn man die Größe zweier Himmelskörper vergleichen möchte, muß man Sätze wie die folgenden formulieren können:

ist der durchmesser von jupiter groesser als der durchmesser von uranus
welcher mond besitzt einen durchmesser der groesser als 5000 km ist

Die Realisierung solche Sätze setzt zuerst das Vorhandensein des Verbs "ist" voraus. Dies ist jedoch nur eine Frage der Erweiterung des Lexikons. Gleiches gilt für das Nomen "durchmesser" und die Np "5000 km". Weit interessanter ist die Frage, wie man die Sequenz "groesser als" in die Syntax hineinbringt, ohne die Regeln zu stark abzuändern. Zunächst definieren wir eine Kategorie, die Vergleiche ermöglicht: Die Adjektivphrase (ap). Man sieht an den folgenden beiden Sätzen, daß ein Vergleich nichts anderes ist als ein prädikatives Adjektiv:

der durchmesser von jupiter ist gross
der durchmesser von jupiter ist groesser als der durchmesser von uranus

Daher definieren wir folgende Regel, die zunächst ohne Merkmale bleibt, um sie übersichtlicher zu halten

```
ap --> [ groesser, als ], np.
```

die nun an der gleichen Stelle vorkommen können soll wie in Sätzen ohne Vergleiche die Objektnominalphrase. Das heißt wir müssen an der Stelle in vp1

```
vp1 --> nprec, vg.
```

sowohl die rekursive NP als auch die AP ermöglichen, indem wir eine neue Regel np_oder_ap einführen, die diese Alternativen berücksichtigt:

```
np_oder_ap -->  nprec.
np_oder_ap -->  [ groesser, als ], np.
```

Nun definieren wir das Terminal "groesser" noch als "komparativ"-Terminal, um es mit "kleiner" innerhalb einer Kategorialbezeichnung zu führen:

```
np_oder_ap -->  nprec.
np_oder_ap -->   komparativ , [als],np.
komparativ -->  [groesser ].
```

Das Wort "als", das den zweiten Teil des Vergleichs einleitet, möchten wir als zu der nachfolgenden Nominalphrase zugehörig betrachten, da wir weiter unten diese neue Kategorie "als_np" noch als Einheit verwenden werden. Also bekommen wir:

```
np_oder_ap -->    ap.
ap -->            komparativ ,   als_np.
als_np -->        [als],    np.
```

Wir geben nun alle Regeln mit deren Merkmalen wieder, die wir für die Komparativerweiterung benötigen:

```
vp1(Form,Subkat,Rel) -->
      nprec(Form,Subkat,Rel),
      vg(Form,Subkat).

np_oder_ap(Form2,Subkat,R) -->
      nprec(Form2,Subkat,R).
np_oder_ap(Form2,Subkat,R) -->
      ap(Form2,Subkat,R).

ap(Form,[Subkat1,Subkat2],Extrapos) -->
      komparativ([Subkat1,Subkat2]),
      als_np(Form,Subkat2,Extrapos).

als_np(Form2, Subkat,R) -->
      [als],
      np(Form2,Subkat,R).
```

```
komparativ(Subkat) -->
   [groesser],
   {subkat(groesser_als,_,Subkat)}.

v(finit,Subkat) -->
   [ist],
   {subkat(sein,sing,Subkat)}.

subkat(sein,Numerus,Subkat):-
   Subkat = [[nom,Numerus,_],[nom,_,_]].
subkat(groesser_als,Numerus,Subkat):-
   Subkat = [[nom,Numerus,_],[nom,_,_]].
```

Man muß nun noch zeigen, daß die Komparativkonstruktion nicht in anderen Satzkontexten verwendbar ist, so daß Sätze wie die folgenden entstehen könnten:

```
* der astronom entdeckte groesser als herschel
```

Die Struktur dieses Satzes ist bis auf das Verb vollkommen gleichartig zu der eines korrekten Vergleichssatzes. Daß ein solcher Satz nicht realisiert werden kann, liegt an der Definition der Komplementstruktur der Verben innerhalb des Prädikates "subkat". Hier ist für das Verb

"sein" und den prädikativen Ausdruck "groesser_als" gleichermaßen festgelegt, daß beide Komplemente den Wert "nom", also Nominativ besitzen müssen. Diese Forderung wird aber nur für das Verb "sein" erfüllt, da alle anderen Verben ein Akkusativkomplement fordern.

Wenn wir die rekursiven Kategorien "rs" und "pp" auskommentieren, können wir den Generierungsmodus unserer Grammatik verwenden, um den positiven Beweis zu führen:

```
?-  s(finit, _, Eingabe, [] ), member(groesser,Eingabe), write(Eingabe), nl, fail.
```

Dieser Aufruf generiert alle Sätze, die das Vergleichswort "groesser" enthalten (member) und schreibt sie auf die Ausgabe. Das Ergebnis zeigt, daß die Definition des Komparativs im Rahmen unserer Grammatik korrekt ist.

Der Komparativ verfügt über eine dem Relativsatz ähnliche Eigenschaft. Ein Teil seiner Struktur kann extraponiert werden:

ein planet dessen durchmesser groesser als der durchmesser von pluto ist umkreist die sonne
ein planet dessen durchmesser groesser ist als der durchmesser von pluto umkreist die sonne

Die Implementierung dieser Verschiebungsmöglichkeit ist sehr einfach, da der grundlegende Mechanismus der Extrapostion schon für den Relativsatz konstruiert wurde. Man braucht nur der "als_np" im Mittelfeld des Satzes auch die Möglichkeit zu geben, daß sie leerlaufen kann und dies ähnlich wie beim Relativsatz durch ein Merkmal (hier: [], bzw. als_np(Form,Subkat2,[])) anzeigen.

Ein Unterschied zum Relativsatz ist die Tatsache, daß der verschobene Komparativteil nicht optional ist, sondern immer realisiert werden muß. Die Initialisierung des dritten Arguments innerhalb des Terms "als_np(Form,Subkat2,[]) " als leere Liste sorgt dafür, daß beim Aufruf im Nachfeld nur die erste "als_np" Regel verwendet werden kann. Um die Abarbeitung beider verschiebbarer Kategorien, also Komparativ-PP und Relativsatz, im Nachfeld übersichtlicher zu gestalten, schreiben wir die Nachfeldregel nun als rekusrsive Prozedur auf. Diese nimmt alle Elemente aus der Extrapositionsliste und übergibt sie an die Regel "extraposition", die die Fallunterscheidungen macht.

Die Syntax unserer Grammatik betrachten wir als vorläufig abgeschlossen und wenden uns nun dem Lexikonbereich zu. Zuvor geben wir als Zusammenfassung ein Listing des Standes der Syntax wieder:

```
s(Form,[Subkat|SubkatRest]) -->
        np(Form,Subkat,Extrapos),
        vp(Form,[Subkat|SubkatRest],Extrapos).

s(finit,_)  -->
        aux(finit/Form,Subkat),
        s(Form,Subkat).

vp(Form,Subkat,Extrapos) -->
        vp2(Form,Subkat, ExtraposRest ),
        nachfeld([Extrapos|ExtraposRest]).

vp2(Form,Subkat,Extrapos) -->
        aux(Form/Form2,Subkat),
        vp1(Form2,Subkat, Extrapos ).

vp1(Form,Subkat,Extrapos) -->
        np_oder_ap(Form,Subkat,Extrapos),
        vg(Form,Subkat).

vg(Form,Subkat)  -->
        v(Form2,Subkat),
        aux(Form1,Subkat),
        { p(Form,Form1,Form2) }.

np(_,Subkat,[]) --> en(Subkat).
np(Form,Subkat,Extrapos) -->
        det(Form,Subkat),
        n1(Subkat,Extrapos).

np_oder_ap(Form,[_|Rest],Extrapos)  -->
        nprec(Form,Rest,Extrapos).
np_oder_ap(Form,Subkat,Extrapos)  -->
        ap(Form,Subkat,Extrapos).
ap(Form,[Subkat1,Subkat2],Extrapos) -->
        komparativ([Subkat1,Subkat2]),
        als_np(Form,Subkat2,Extrapos).

nprec(_,[],[]) --> [].
nprec(Form,[Subkat|SubkatRest],[Extrapos|ExtraposRest]) -->
        np(Form,Subkat,Extrapos),
        nprec(Form,SubkatRest,ExtraposRest).
```

```
n1(Subkat,Extrapos) -->
        n(Subkat),
        radj( Subkat , Extrapos).

radj( Subkat , Extrapos) --> rs(rel,Subkat,Extrapos).
radj( Subkat ,Extrapos) --> pp(Extrapos).

als_np(Form,Subkat2,[]) -->
        [als],
        np(Form,Subkat2,_).
als_np(Form,Subkat2,[als_np(Form,Subkat2,[])]) --> [].

pp(Extrapos) -->
        prep(Subkat,Form),
        np(Form, Subkat , Extrapos).

rs(Form,[_|Subkat],[]) -->
        rp([K|Subkat]),
        vp(Form,[[K|Subkat]|_],[]).
rs(_,Subkat, rs(Subkat )) --> [].

nachfeld([]) ---> [].
nachfeld([H|T]) --->
        extraposition(H),
        nachfeld(T).

extraposition([]) ---> [].
extraposition(rs(Z)) --->
        rs(rel, Z, _).
extraposition(als_np(Form,Subkat,E)) --->
        als_np(Form, Subkat,E).
```

Lexikonerweiterung

Die Anzahl der Sätze, die bislang von der Grammatik erzeugt und analysiert werden können, ist beschränkt durch die Anzahl der lexikalischen Alternativen, die für jede terminale Kategorie existieren. Dies hat zwar zu einer Reihe von unsinnigen Beispielsätzen geführt wie

```
herschel entdeckte herschel
```

hatte aber gleichzeitig den Vorteil einer kleineren Anzahl von Analyseergebnissen beim Generieren und einer besseren Übersicht über die Gesamtabdeckung der Grammatik. Bei der nun folgenden Auffüllung des Lexikons orientieren wir uns einerseits an der Sammlung von Sätzen, die diesem Kapitel vorangestellt wurde. Auf der anderen Seite wird sie beschränkt durch die syntaktischen Konstruktionsmöglichkeiten, die bislang realisiert worden sind.

Beginnen wir mit dem Verbalbereich. Außer dem Verb "entdecken" (für das noch eine Pluralvariante analog zu den folgenden Verben hinzugefügt werden muß) definieren wir noch die Verben "umkreisen" und "besitzen" für Sätze wie

```
welcher mond umkreist uranus
welchen durchmesser besitzt oberon
```

```
v(finit,Subkat) --> [umkreist],   {subkat(umkreisen,sing,Subkat)}.
v(finit,Subkat) --> [umkreisen],{subkat(umkreisen,plu,Subkat)}.

v(finit,Subkat) --> [besitzt],      {subkat(besitzen,sing,Subkat)}.
v(finit,Subkat) --> [besitzen],     {subkat(besitzen,plu,Subkat)}.
```

Beide neuen Verben verfügen jedoch über kein Partizip, da dies zu wenig plausiblen Sätzen führt. Die Definitionen der Prozedur "subkat" sind vollkommen analog zu den von "entdecken". Beide Verben sind nun auch als Pluralformen verfügbar, so daß man nun überprüfen kann, ob die Subjekt-Verb-Kongruenz für den Numerus innerhalb von "subkat" korrekt definiert worden ist.

Für die beiden Auxiliare "sein" und "haben" erfolgt die Definition der Pluralvarianten ebenfalls analog zu den obigen:

```
aux(finit/infinit,Subkat) --> [haben],    {subkat(haben,plu,Subkat)}.
aux(finit/infinit,Subkat) --> [sind],     {subkat(sein,plu,Subkat)}.
```

Im Nominalbereich müssen wir für eine größere Anzahl neuer Wörter sorgen. Hier liegt die Erweiterung ebenfalls vor allem in der Hinzufügung von Pluralvarianten, wie im folgenden Beispiel:

```
n([nom,sing,masc]) --> [astronom].
n([nom,plu,masc]) --> [astronomen].
n([akk,sing,masc]) --> [astronomen].
n([akk,plu,masc]) --> [astronomen].
n([dat,sing,masc]) --> [astronomen].
n([dat,plu,masc]) --> [astronomen].
```

Die Nomen, die wir neu einführen, folgen in ihrer Definition diesem Schema, deshalb ersparen wir uns die explizite Darstellung und zählen sie nur auf:

sonne, planet, mond, entdeckung,
durchmesser, kilometer, himmelskoerper

Das letzte Nomen soll es ermöglichen, etwas allgemeiner über Himmelskörper zu sprechen wie z.B. in dem Satz:

```
welche himmelskoerper hat herschel entdeckt
```

Damit kann man also sowohl Planeten als auch Monde benennen. Zum Nominalbereich zählen wir die Relativpronomen, die nun ebenfalls in Pluralvarianten vorkommen können:

```
rp([nom,plu,masc]) --> [die].
rp([akk,plu,masc]) --> [die].
rp([nom,plu,fem]) --> [die].
rp([akk,plu,fem]) --> [die].
```

Schließlich erweitern wir auch die Artikel entsprechend dem Pluralparadigma:

```
dets(def,[nom,plu,masc]) --> [die].
dets(def,[akk,plu,masc]) --> [die].
dets(frage,[nom,plu,masc]) --> [wieviele].
dets(frage,[nom,plu,masc]) --> [welche].
```

Außerdem definieren wir auch Wörter wie "jeder" und "jeden" als Artikel:

```
dets(all,[nom,sing,masc]) --> [jeder].
dets(all,[akk,sing,masc]) --> [jeden].
dets(all,[nom,sing,fem]) --> [jede].
```

```
dets(all,[akk,sing,fem]) --> [jede].
```

Nun fehlen noch zwei Bereiche, die etwas umständlicher zu behandeln sind. Der eine Bereich ist der der Namen. Hier besteht das Problem in der Datenmenge, denn bei der Art von Datenbank, die wir voraussetzen, enthält jede Datenrelation einen Namen. Man steht nun vor der Alternative, einfach alle Namen aus der Datenbank in das Lexikon zu überführen und sie analog zu dem vorhandenen Namen "herschel" zu definieren:

```
en([Kasus,sing,_]) --> [herschel], { (Kasus = nom; Kasus = akk ; Kasus = dat) }.
en([Kasus,sing,_]) --> [galilei], { (Kasus = nom; Kasus = akk ; Kasus = dat) }.
en([Kasus,sing,_]) --> [uranus], { (Kasus = nom; Kasus = akk ; Kasus = dat) }.
```

Dies hat den Vorteil einer sauberen Trennung der beiden Vorkommnisse von Namen und den Nachteil großer Redundanz, da es zu einer Verdopplung einer Datenstruktur führt. Die andere Möglichkeit besteht darin, die Datenbank direkt als Erweiterung des Lexikons zu verwenden:

```
en([Kasus,sing,_]) -->
    [ Name ],
    {
    db( Name_Himmelskoerper, _, _, Name_Astronom ,_),
    (Name = Name_Himmelskoerper ; Name = Name_Astronom),
    (Kasus = nom; Kasus = akk ; Kasus = dat)
    }.
```

Dies ist eleganter aber ineffizienter, da die Datenbank eben mehr Informationen als nur Namen enthält, diese aber immer mitberechnet werden müssen. Beide Definitionsarten sind äquivalent in Bezug auf ihre Progammsematik.

Der zweite noch zu behandelnde Bereich betrifft Zahlen. In dem Satz

```
hat uranus einen durchmesser von 50000 km
```

müssen wir die Zahl "50000" in irgendeiner Form als korrekten Sprachbestandteil beschreiben. Eine ganz schlechte Art, dies zu tun, wäre diejenige, versuchen zu wollen, alle Zahlen einzeln zu definieren. Plausibler ist es, Zahlen über einen Typentest zu definieren, etwa in der folgenden Form:

```
dets( def, [dat, plu, _] ) --> [ Zahl ], { integer( Zahl ) }.
```

Mit dieser Definition hat man übrigens die Verwendung von Zahlen auf Präpositionalphrasen beschränkt, da Dativkomplemente nur dort vorgesehen sind. Eine Erweiterung des Lexikons, die eigentlich syntaktischen Charakter hat, ist diejenige, die eine Variante des Relativsatzes erlaubt. Relativsätze gibt es bisher nur in der Form, daß ein Relativpronomen einen Satz einleitet:

herschel hat einen mond entdeckt der einen planeten umkreist

Nun ist es relativ einfach auch folgende relativische Einleitung zu implementieren:

uranus hat einen mond entdeckt dessen durchmesser groesser ist als der durchmesser von venus

Dazu mißbrauchen wir die terminale Regel "rpron" und geben ihr quasi syntaktischen Status, indem wir sie wie folgt definieren:

```
rp(Subkat) --> np(relpro,Subkat,_).
```

Damit kann ein Relativpronomen auch durch eine Nominalphrase vertreten werden. Das Merkmal "relpro" steuert dabei die Auswahl des Artikel "dessen" innerhalb dieser Nominalphrase :

```
det(relpro,[_,sing,masc]) --> [dessen].
det(relpro,[_,sing,neu]) --> [dessen].
```

Da alle "normalen" nicht pronominalen Artikel an dieser Stelle andere Merkmalwerte besitzen, ist dafür gesorgt, daß es keinen Satz der Art

*uranus hat einen mond entdeckt der durchmesser groesser ist als der durchmesser von venus

geben kann.

Semantik

Ist eine sprachliche Äußerung von dem System akzeptiert worden, so kann es vorkommen, daß diese mehrdeutig ist und zwar in einer Weise, die von der syntaktischen Form unabhängig ist. Entsprechen einer syntaktischen Struktur mehrere Bedeutungen, dann spricht man von einer **semantisch ambigen** sprachlichen Äußerung:

$$\text{Satz} \Rightarrow \text{Syntax}(\text{Satz}) \Rightarrow \text{Semantik}^1(\text{Satz}) \Rightarrow \text{Semantik}^2(\text{Satz}) \ldots$$

Ein schon klassisches Beispiel ist die Ambiguität, die sich aus dem folgenden Satz ergibt:

Jeder Mann liebt eine Frau

Die erste Bedeutung des Satzes besteht in der Relation zwischen allen Männern und einer bestimmten Frau, während die zweite Lesart nichts darüber aussagt, ob es sich immer um diesselbe Frau oder um verschiedene Frauen handelt.

Auch der umgekehrte Fall ist möglich, daß es mehrere Sätze gibt, die vom System akzeptiert werden, und die alle die gleiche Bedeutung haben, obwohl sie sich in ihrer syntaktischen Struktur unterscheiden. Man spricht dann von **Paraphrasen** einer Satzsemantik:

$$\text{Satz}^1, \text{Satz}^2, \ldots \; \text{Syntax}^1(\text{Satz}) \Rightarrow \text{Syntax}^2(\text{Satz}) \Rightarrow \text{Semantik}(\text{Satz}) \ldots$$

Die Beziehung zwischen einem Aktivaussage und seiner Passivvariante ist hierfür ein typisches Beispiel:

Peter liebt eine Frau = eine Frau wird von Peter geliebt

Die Erkennung und Behandlung von Ambiguitäten und Paraphrasen ist eine der hauptsächlichen Motivationen für die **kompositionale Semantik**, die als separate Strukturebene angesehen werden muß. Als nicht-ambige Darstellungsform verwendet sie verschiedene logische Sprachen wie z.B. die Prädikatenlogiken n-ter Stufe. Die Verwendung von logischen Kalkülen als Repräsentationsebene legt außerdem eine einfache Erkennung von widersprüchlichen Benutzereingaben nahe. So kann die folgende Aussage

Peter hat den Wagen genommen und Peter nahm den Wagen nicht

vom System leichter als Inkonsistenz identifiziert werden, wenn die Sätze in ihrer logischen Form (also etwa "A und non A") vorliegen. In dieser Weise läßt sich Semantik als ein der Syntaxanalyse nachgeordneter Filter auffassen, der die Menge der akzeptierbaren Sätze weiter reduziert.

Die Idee der kompositionalen Semantik besteht darin, die Gesamtbedeutung eines Satzes aus den Teilbedeutungen, also den Semantiken der Phrasen und Wörter aufzubauen. Dieser Aufbau darf jedoch nicht zu frei erfolgen, da die Bedeutung eines Wortes oft die Bedeutungen der anderen Wörter, mit denen es in einem Teilsatz oder Satz vorkommen kann, einschränkt. Chomskys klassisches Beispiel für einen syntaktisch korrekten aber semantisch mehrfach fehlerhaften Satz ist das folgende:

Colorless green ideas sleep furiuosly

Die Angabe, daß z.B. das Verb "sleep" nur mit belebten Subjekten kombinierbar ist, ist Teil der **Selektionsrestriktionen**, die im Lexikon des betreffenden Wortes vermerkt sein müssen. Da neben diesen Restriktionen noch eine Vielzahl anderer Informationen den Lexikoneintrag eines Wortes ausmachen, spricht man auch von einem eigenen Feld der **lexikalischen Semantik**.

Satzsemantik

Das eher heterogene Bild, das man in der Syntax findet, nämlich, daß es nicht *die* syntaktische Theorie gibt, sondern eine Vielzahl konkurrierender Ansätze, trifft für die Semantik nicht im selben Maße zu. Hier gibt es eine beherrschende Richtung und das ist die wahrheitsfunktionale oder modelltheoretischeSemantik. Dieser Ansatz definiert die Bedeutung eines (Aussage-) Satzes letztlich durch dessen Wahrheit oder Falschheit. So würde der Satz

herschel hat einen planeten entdeckt

in unserem Modell mit "wahr" bewertet werden. Die Bedeutung des Satzes würde uns aber auch dann klar sein, wenn er mit falsch bewertet werden würde. Es genügt in diesen Fällen die Wahrheitsbedingungen eines Satzes zu kennen:

Wenn das Subjekt des Satzes ("herschel") die erste Argumentstelle des Prädikates ("entdeckt") belegt und das Objekt ("ein planet") die zweite Argumentstelle des Prädikates und das Prädikat mit diesen Belegungen in der Datenbank vorhanden ist, dann muß der Satz wahr sein.

An dieser Stelle haben wir bereits Begriffe wie "Prädikat" und "Argument" verwendet, die Teil der Sprache sind, in der die wahrheitsfunktionale Semantik dargestellt wird: die Prädikatenlogik. Die Aufgabenstellung innerhalb des Moduls "grammatikalische_analyse"

soll ja wie schon zuvor dargestellt lauten, eine Übersetzung von syntaktischen zu semantischen Repräsentationen zu finden, also:

```
grammatikalische_analyse( Eingabe, Repräsentation):-
        morphologische_analyse(Grundformen, Vollformen),
        lexikalische_analyse(Eingabe, Getaggte_Eingabe),
        syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
        semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
        pragmatische_analyse(SemantischeRepräsentation, Repräsentation).
```

wobei wir Prädikatenlogik als semantische Repräsentationsform gewählt haben. Abgesehen davon, daß es noch einen weiteren Bearbeitungsschritt (pragmatische_analyse) innerhalb des Grammatikmoduls gibt, ist mit der Übersetzung in Prädikatenlogik dann also erst eine Hälfte des Semantikproblems gelöst. Der modelltheoretische Teil, also die Berechnung der Wahrheitswerte von Sätzen aus prädikatenlogischen Formeln, geschieht außerhalb des Grammatikmoduls im Modul "evaluierung". Wenn wir für die Zwecke der Darstellung nun zunächst davon ausgehen, daß die pragmatische Analyse keine Änderung der semantischen Repräsentation bewirkt, also diese Regel eine identische Abbildung definiert

```
pragmatische_analyse(SemantischeRepräsentation, SemantischeRepräsentation).
```

dann kann der modelltheoretische Teil als derjenige definiert werden, der die semantischen Repräsentationen, also die prädikatenlogischen Ausdrücke in dem gegebenen Modell evaluiert. Dazu nehmen wir noch einmal die Definition der obersten Regel her:

```
computerlinguistisches_problem(Eingabe, Ausgabe):-
        grammatikalische_analyse( Eingabe, LogischeRepräsentation),
        anwendung( LogischeRepräsentation, Ausgabe).
```

Da die Anwendung in diesem Fall aus der Datenbankabfrage besteht, definieren wir die Prozedur "anwendung" als Evaluierung der logischen Repräsentationen (Formeln) in der jeweiligen Datenbank:

```
anwendung( LogischeRepräsentation, Ausgabe):-
        evaluierung(LogischeRepräsentation, Ausgabe).
```

Somit läßt sich die Aufgabe der Semantik auf die Definition der beiden Ziele "semantische_analyse" und "evaluierung" reduzieren. Auf das Teilproblem, das wir nun übersprungen haben, nämlich "pragmatische_analyse" werden wir später zurückkommen und widmen uns nun der Implementierung der semantischen Bereiche:

```
semantische_analyse(SyntaktischerBaum, PraedikatenlogischeAusdruecke)

evaluierung(PraedikatenlogischeAusdruecke, Ausgabe)
```

Bevor wir die Teilaufgabe der Übersetzung von syntaktischen Bäumen zu Ausdrücken der Prädikatenlogik angehen, müssen wir zunächst auf die Syntax der Prädikatenlogik und auf die speziellen Strukturen, die für die Übertragung natürlichsprachlicher Ausdrücke gebraucht werden, eingehen.

Syntax der Prädikatenlogik

Die kanonische Darstellungsform der prädikatenlogischen Ausdrücke verwendet Quantorenzeichen wie "∀" , und Junktorenzeichen wie "∧". Außerdem werden Junktorenzeichen als Infixoperatoren dargestellt wie in "a ∧ b". Da wir uns auf Prolog (Syntax) als alleiniger Beschreibungssprache beschränken wollen, wandeln wir diese Konvention so ab, daß wir Quantoren lexikalisch umschreiben (z.B.: "für_alle" statt "∀") und die Infixoperatoren zu Prologoperatoren machen (z.B.: "p & q" statt "p ∧ q"). Die folgende kontextfreie DCG beschreibt zunächst die Syntax der Aussagenlogik mit diesen Änderungen.

Aussagenlogik:

```
wohl_geformte_formel --> atomare_formel.
wohl_geformte_formel -->
                        junktor2,
                ['('],  wohl_geformte_formel,  [','],
                        wohl_geformte_formel,  [')'].
wohl_geformte_formel -->
                        junktor1,
                ['('],  wohl_geformte_formel,  [')'].

junktor2 --> [&].              % Konjunktion
junktor2 --> [v].              % Disjunktion
junktor2 --> [=>].             % Implikation
junktor2 --> [<->].            % Bikonditional
junktor1 --> [¬].              % Negation

atomare_formel --> praedikat.
praedikat --> [X], { atom(X) , not(element(X, [[],&,v,=>,<->,¬]))}
```

Eine wohlgeformte Formel der Aussagenlogik besteht somit entweder aus einer atomaren Formel wie

```
max_mag_musik
```

Oder sie ist komplex und verbindet Einzelformeln mithilfe zweiwertiger Junktoren bzw. dem einwertigen Junktor zu Formeln wie

```
max_mag_musik  v  ¬ max_mag_musik
```

Von der Aussagenlogik gelangt man zur Prädikatenlogik, indem man die Möglichkeit der Quantifizierung hinzunimmt. Dazu definiert man den Existenzquantor und den Allquantor jeweils mit einer Variablen und erweitert die Definition der Prädikate von null-stelligen zu n-stelligen Ausdrücken (mit n >= 0), wobei die einzelnen Stellen, die Argumente nun selbst von der Form Variable bzw. 0-stelliges Prädikat (komplexe Argumente lassen wir außer acht) sein sollen:

Prädikatenlogik:

```
wohl_geformte_formel  -->
                    quantor,
                    ['('],
                    variable,
                    [','],
                    wohl_geformte_formel,
                    [')'].

quantor --> [fuer_alle].                % Allquantor
quantor --> [existiert].                % Existenzquantor

atomare_formel --> praedikat,   ['('], argumente, [')'].

argumente --> term.
argumente --> term, argumente.

term --> [X], {atomic(X)}.
term --> variable.

praedikat --> [X], { atom(X) , not(element(X, [[],&,v,=>,<->,¬, fuer_alle, existiert]))}.
variable --> [X], { var(X) }.
```

Mit diesen Zusatzdefinitionen erhält man eine komplette Beschreibung prädikatenlogischer Ausdrücke (in Termschreibweise). Die Einführung von Variablen bringt einige Eigenschaften hinzu, die Formeln betrifft: Wenn in einer Formel jede Argumentvariable zuvor durch einen Quantor eingeführt wurde und in dessen Bereich (Skopus) liegt, so bezeichnet man die Formel als geschlossen und die Variablen als gebunden. Die Benennung der Variablen ist zwar frei, aber man sollte bei der Wahl der Namen darauf achten, daß keine Kollisionen entstehen. Diese erreicht man einfach dadurch, daß jeder Quantor einen Variablennamen einführt, der bisher noch nicht vorkam. Wenn man sich daran hält, dann tritt eine Schwierigkeit nicht auf, die mit der Wahl von Prolog als Darstellungssprache der Prädikatenlogik zusammenhängt. So ist der Bereich einer Variablen die Klausel, in der sie

auftritt, unabhängig von dem Quantorensymbol, das sie einführt. Mit der obigen Selbstbeschränkung ist dies allerdings unproblematisch.

Die Übersetzung natürlichsprachlicher Ausdrücke in Ausdrücke der Prädikatenlogik geschieht auf zwei Ebenen:

1.) Die lexikalische Ebene

Prädikate:

Die Prädikatsnamen entsprechen den Namen der Verben, Nomen und Präpositionen. Um die natürlichsprachliche Form von der prädikatenlogischen zu trennen versieht man das Prädikat mit einem Operator, bzw. benennt es in einen abgewandelten Namen um. So könnte man beispielsweise das Verb "entdeckte" in dem Satz "herschel entdeckte einen planeten" als das Prädikat "entdecken" übersetzen, das sich aus dem Namen des infiniten Verbs ergibt.
Die Stelligkeit des Prädikats richtet sich ebenfalls nach der Anzahl der Argumente des entsprechenden Verbs, bzw. Nomens oder Präposition. So wird das 2-stellige Verb "entdecken" in das 2-stellige Prädikat "entdecken(X,Y)" übersetzt.

das Verb "entdeckte" wird prädikatenlogisch zu entdecken(X,Y)

Beispielsübersetzungen für Nomen und Präpositionen sind:

das Nomen "planet" wird prädikatenlogisch zu "planet(X)".
die Präposition "von" wird prädikatenlogisch zu "von(X,Y)".

Namen und Zahlen:
Ein Name wie "herschel" bleibt unverändert. Seine Übersetzung in die Prädikatenlogik ist der identische Ausdruck "herschel". Das gleiche gilt für Zahlen.

Artikel:
Artikel stellen die komplexesten natürlichsprachlichen Ausdrücke dar. So wird der Artikel "ein" in "herschel entdeckte einen planeten" als Quantor "existiert" übersetzt, der - siehe syntaktische Definition - eine Variable einführt und einen Bereich - Skopus genannt - in dem sie gültig ist:

der Artikel "ein" wird zu existiert(X, S)

Der Skopus ist im Falle des Beispielsatzes "herschel entdeckte einen planeten" der ganze Satz.

Andere Artikelformen können ganz andere Übersetzungen in die Prädikatenlogik erhalten. So wird der Artikel "jeder" wie in "jeder astronom entdeckte einen planeten" als Allquantor übersetzt, der sich hinsichtlich der internen Struktur seines Skopus von dem Quantor "existiert" unterscheidet:

der Artikel "jeder" wird zu fuer_alle(X, S)

Auch der definite Artikel erhält eine gesonderte Definition (s.u.), da er eine andere Interpretation als die All- und Existenzquantoren erhalten soll.

2.) Die syntaktische Ebene

Hier wird festgelegt, in welcher Weise die terminalen und nichterminalen Kategorien miteinander kombinieren dürfen, um eine wohlgeformte Formel zu bilden, die eine korrekte Übersetzung eines natürlichsprachlichen Ausdruckes darstellt.

So läßt sich z.B. der folgende abgewandelte Syllogismus von Aristoteles

(1) jeder planet ist groesser als 1000 km.
(2) uranus ist ein planet.
---
(3) also ist uranus groesser als 1000 km.

in prädikatenlogischer Termschreibweise wie folgt wiedergeben:

```
fuer_alle(X,
        planet_(X) => groesser(X, 1000)
        &
        existiert(Y),
                planet_(Y)
                &
                gleich_(Y, uranus )
        =>
        groesser(uranus, 1000)
```

Eine gute Übung für das Umgehen mit prädikatenlogischen Formeln wie für das Programmieren in Prolog besteht nun darin, ein Programm anzugeben, das eine prädikatenlogische Formel als Eingabe nimmt und entscheidet, ob es sich dabei um eine geschlossene Formel handelt oder nicht. Hierbei kommt es also darauf an sicherzustellen, daß keine Variable in einer Formel auftaucht, die nicht im Skopus eines Quantors liegt. Die Programmlösung muß also einerseits wie unsere DCG für die Syntax der Prädikatenlogik die Struktur einer Formel durchgehen können. Zudem muß ein Stack angelegt werden, der jede von einem Quantor eingeführte Variable aufnimmt und sie an jede Stelle unterhalb des Quantors führt. Schließlich muß an den Stellen, an denen Variablen als Prädikatsargumente auftauchen, eine Überprüfung stattfinden, ob jedes variable Prädikatsargument auch im Stack vorkommt. Das folgende Programm implementiert diese Vorgehensweise:

```
geschlossene_formel(Formel) :- wohl_geformte_formel(Formel, []).

wohl_geformte_formel(W,X) :- atomare_formel(W,X).
wohl_geformte_formel(WGF,X):-
                    WGF =.. [J,W1,W2],
                    junktor2(J),
                    wohl_geformte_formel(W1,X),
                    wohl_geformte_formel(W2,X).
wohl_geformte_formel(WGF,X) :-
                    WGF =.. [J,W],
                    junktor1(J),
                    wohl_geformte_formel(W,X).
wohl_geformte_formel(WGF,X) :-
                    WGF =.. [Q,V,W],
                    quantor(Q),
                    variable(V),
                    wohl_geformte_formel(W,[V|X]).


atomare_formel(T,X):- T =.. [P|A], praedikat(P),   argumente(A,X).
atomare_formel(P,_):- praedikat(P).

argumente([T],X) :- term(T,X).
argumente([T|R],X) :- term(T,X), argumente(R,X).

term(T,_) :- atomic(T).
term(T,X) :- variable(T), element(T,X).
```

Der syntaktische Teil des Programms zerlegt die Formel sukzessive und führt eine Liste (den Variablenstack) mit, die am Beginn mit "[]" initialisiert wurde. Nur an der Quantorenstelle wird eine Variable in diese Liste aufgenommen und in die Subformeln weitergereicht. Ist eine Argumentvariable identifiziert, findet eine Elementüberprüfung statt. Die Definition von "element" geschieht dabei über den Operator "==", der die Identität zweier Variablen überprüft.

```
junktor2(&).
junktor2(v).
junktor2(=>).
junktor2(<->).
junktor1(¬).

quantor(fuer_alle).
quantor(existiert).

praedikat(X) :- atom(X), not(X = []; quantor(X); junktor2(X); junktor1(X)).

variable(X) :- var(X).

element(X, [S|T]):- X == S.
element(X, [S|T]):- element(X,T).
```

Es gibt für dieses Problem eine alternative Lösungsmöglichkeit, die auf das Konzept eines Variablenstacks verzichtet. Stattdessen macht man sich dabei den Unifikationsmechanismus von Prolog zunutze und erhält die Möglichkeit, damit Variablen für die Laufzeit des Programms zu belegen, d.h. mit einem festen Wert zu unifizieren. So kann man an Quantorenstelle der jeweiligen Variablen einen beliebigen konstanten Wert zuweisen, so daß eine korrekte Formel kein variables Argument mehr enthält. Dann nimmt man die Definition für ein variables Argument (innerhalb von "term") aus dem Programm. Damit kann keine Formel mehr abgeleitet werden, die ein Argument enthält, das nicht von einem Quantor eingeführt wurde. Der nachteilige Seiteneffekt dieser Programmvariante besteht in der Bindung aller Variablen an einen festen Wert. Da man die Variablen jedoch für die Berechnung der Semantik benötigt, muß man die ursprüngliche Formel zur Verfügung haben. Dies ist in Prolog leicht zu erhalten, indem man den Aufruf des Programms in die doppelte Negation einbettet, die dafür sorgt, daß das Programm als Test abläuft, ohne eine Bindung von Variablen an konstante Terme vorzunehmen.

Semantik der Prädikatenlogik

Die Semantik der Prädikatenlogik ist modelltheoretisch und wahrheitsfunktional definiert. Das heißt, daß es einen aus der Aussagenlogik bekannten Teil der Semantik gibt, der über die Angabe von Wahrheitswerten für Aussagenjunktoren definiert ist:

```
wahr(¬ Formel):-
    nicht_erfuellbar(wahr(Formel)).
wahr( Formel & Formel2)):-
   wahr(Formel), wahr(Formel2).
wahr(Formel  => Formel2)):-
  wahr(¬Formel v Formel2).
wahr(Formel v Formel2):-
   wahr(¬(¬Formel &¬Formel2)).
wahr(X):- erfuellbar(X).

erfuellbar(F):- call(F), !.

nicht_erfuellbar(F):- call(F), !, fail.
nicht_erfuellbar(_).
```

So ist eine Konjunktion von Aussagen wie

```
true & true v  true
```

wahr, wenn jede Teilformel als wahr bewiesen werden konnte.

Mit Quantoren und Variablen kommen weitere Bedingungen hinzu. So muß zunächst die Eigenschaft atomar zu sein durch die Eigenschaft n-stelliges Prädikat zu sein ersetzt werden (diese Definition ersetzt also die letzte "wahr"-Definition oben):

```
wahr(Formel):-
    atomar(Formel),  erfuellbar(Formel).
```

Dies geschieht über die Angabe zweier Wertebereiche, einen für Individuen und einen für Prädikate:

```
atomar(F):- F =.. [R,X], relationsbezeichnung(R), individuum(X).
atomar(F):- F =.. [R,X,Y], relationsbezeichnung(R), individuum(X), individuum(Y).
atomar(F):- F =.. [R,X,Y,Z], relationsbezeichnung(R), individuum(X), individuum(Y),
individuum(Z).

individuum(X):-
        member(X,[ariel,pluto,callisto,jupiter,uranus,lassell,tombaugh,
        galilei,sonne,3000,9000,4848,51800,142800,1392000]).

relationsbezeichnung(R):-
        member(R,[entdecken,umkreisen,astronom,durchmesser,
        groesser,kleiner,gleich,sonne,mond,planet,himmelskoerper]).
```

Nun lassen sich schließlich die Quantoren definieren, die über den Wertebereich von Individuen gehen:

```
wahr(existiert(X,Formel)):-
        individuum(X),  wahr(Formel).
wahr(fuer_alle(X,Formel)):-
        wahr(¬ existiert(X,¬ Formel)).
```

Die Auswertung einer Existenzaussage geschieht dabei immer folgendermaßen: Es wird ein beliebiges Element aus dem Individuenbereich ausgewählt und versucht, den Rest der Formel mit dieser Belegung zu beweisen. Bei der Definition des Allquantors macht man sich die Äquivalenz mit der doppelt negierten Existenzaussage zunutze. Für die Auswertung der Prädikate mit konkreten Belegungen fehlt nun noch die Angabe der n-stelligen (hier 1 bis 3-stelligen) Prädikate:

```
entdecken(lassell,ariel).
entdecken(tombaugh,pluto).
entdecken(galilei,callisto).
umkreisen(pluto,  sonne).
umkreisen(ariel,  uranus).
umkreisen(callisto,  jupiter).
durchmesser(pluto, 3000).
durchmesser(ariel, 9000).
durchmesser(callisto, 4848).
durchmesser(uranus, 51800).
durchmesser(jupiter, 142800).
durchmesser(sonne, 1392000).
astronom(tombaugh).
astronom(lassell).
planet(pluto).
planet(jupiter).
planet(uranus).
mond(ariel).
mond(callisto).
sonne(sonne).
himmelskoerper(X) :- planet(X).
himmelskoerper(X) :- mond(X).
himmelskoerper(X) :- sonne(X).
groesser(X,Y):- X > Y.
kleiner(X,Y):- X < Y.
gleich(X,X).
astronom(galilei).
```

Anhand des Syllogismusbeispiels läßt sich zeigen, daß man bei der Übersetzung darauf achten muß, daß die Formel die natürlichsprachliche Bedeutung korrekt wiedergibt. So wurde die Nominalphrase "jeder planet" als Implikation übersetzt

```
fuer_alle(X, planet(X) => groesser(X, 1000) ),
```

und nicht etwa als Konjunktion, also

```
fuer_alle(X, planet(X) & groesser(X, 1000) ),
```

Letzterer Fall hätte bei der Interpretation dazu geführt, daß die Teilformel nur dann als wahr ausgewertet worden wäre, wenn der Individuenbereich ausschließlich aus Planeten bestanden hätte. Die Deutung als Implikation ist insofern weniger fordernd, als daß das Konsequens nur dann wahr sein muß, wenn die Variable X mit einem Objekt belegt wird, auf das das Prädikat "menschlich" zutrifft. Nominalphrasen mit einem Artikel wie "alle" oder "jeder" werden daher normalerweise als Implikation gedeutet, während Artikel wie "ein" oder "mancher" zu einer konjunktiven Interpretation führen. Die spezielle konjunktive, bzw. implikative Darstellung der beiden Ausdrücke kennzeichnet sog. "generalisierte Quantoren", deren Elemente über die Aufteilung ihrer obersten binären Verknüpfung ("&", "=>") in eine linke Hälfte "Restriktion" und eine rechte Hälfte "Skopus" , verfügen. Mit diesem in Prolog definierten prädikatenlogischen System lassen sich nun Abfragen der folgenden Art machen:

```
hat ein astronom einen planeten entdeckt
```

was zu der Formel führt:

```
wahr( existiert(X,astronom(X) & existiert(Y, planet(Y) & entdecken(X,Y)))).
```

Nominalphrasen mit indefinitem bzw. unbestimmtem Artikel (ein, einen) erhalten also eine Formalisierung, die aus einem Existenzquantor besteht, der Skopus über eine Konjunktion von Eigenschaften hat:

```
existiert(X, P & Q)
```

im Gegensatz dazu führen Artikel der Form jeder, jeden etc. zu Ausdrücken mit Allquantor und Implikation (s.o.):

```
fuer_alle(X, P => Q)
```

Die Metavariable P steht dabei jeweils für die aus der Nominalphrase kommenden Beschreibungen des Individuums, die Variable Q für die aus der Verbalphrase stammenden Beschreibungen.

Ein dritter Typ von Artikeln, den definiten wie in der Nominalphrase

der astronom

ordnen wir eine Semantik zu, die aus einer Kombination von Existenz- und Allquantoren besteht:

existiert(X, P & fuer_alle(Y, P <-> gleich(X,Y)) & Q)

Im Gegensatz zu "ein" und drückt "der" nicht nur die Forderung nach der Existenz eines Individuums mit bestimmten Eigenschaften aus, sondern zusätzlich auch die Eigenschaft der Einzigartigkeit des Individuums mit den spezifizierten Eigenschaften. Diese Formalisierung entspricht der Definition definiter Beschreibungen, wie sie Russell vorgeschlagen hat. Ein Satz wie

der astronom entdeckte einen planeten

erhält damit die folgende Repräsentation:

existiert(X, astronom (X) & fuer_alle(Y, astronom (Y) <-> gleich(X,Y))
& existiert(Z, planet(Z) & entdecken(X,Z)))

Aus dieser Semantik für den definiten Artikel folgt, daß Sätze wie

gibt es **die** sonne

in unserem System zu wahr ausgewertet werden, da die Nominalphrase "die sonne" eindeutig auf ein und nur ein Individuum angewandt werden kann, während Sätze der Form

gibt es **den** astronomen

zu falsch ausgewertet werden, da das Prädikat "astronom(X)" auf mehr als nur ein Individuum zutrifft, also nicht mehr eindeutig ist.

Bei der Auswertung von Formeln, wie sie gerade beschrieben wurden, belegt Prolog als Seiteneffekt die Variablen, so daß man dies auch als Implementierung von Fragen nach Werten verstehen kann. Die Mächtigkeit dieses Systems ist beträchtlich, kann man doch

mithilfe primitiver sprachlicher Mittel auch Operationen wie Aggregierungen ausdrücken. So kann man einen Superlativ wie "der größte Durchmesser" in dem Satz

welcher planet hat den groessten durchmesser

umformulieren in eine Paraphrase wie z.B.:

gibt es es einen himmelskoerper mit einem durchmesser so dass fuer alle anderen himmelskoerper ungleich diesem himmelskoerper gilt dass sie einen durchmesser haben der kleiner ist

Diese Paraphrase ist nun leicht als prädikatenlogische Formel in unserem System ausdrückbar:

```
existiert(X,
   himmelskoerper(X)
   &
   existiert(Y,
         durchmesser(Y) &  besitzen(X,Y)
         &
         fuer_alle(Z,
               himmelskoerper(Z)  &  ¬gleich(Z,X)
               =>
               existiert(U,
                     durchmesser(U)  & besitzen(Z,U)
                     &
                     kleiner(U,Y))))).
```

Die Abfrage unter Prolog innerhalb des Prädikates "wahr" berechnet das Resultat, das als korrekte Antwort auf die ursprüngliche Frage angesehen werden kann. Der Vorteil der Prädikatenlogik als Sprache der semantischen Repräsentation eines natürlichsprachlichen Satzes liegt also auch darin, daß prädikatenlogische Ausdrücke gleichzeitig Grundausdrücke eines logischen Systems sind, das vielfältige Manipulationen erlaubt. So können beispielsweise paraphrastische natürlichsprachliche Ausdrücke wie

ein planet den herschel entdeckte besitzt einen durchmesser von 1000 km
ein planet der einen durchmesser von 1000 km besitzt ist eine entdeckung von herschel

in ihre jeweilige prädikatenlogische Form gebracht werden

$$\exists x \exists y(\mathit{planet}(x) \wedge \mathit{entdecken}(\mathit{herschel}, x) \wedge \mathit{durchmesser}(x, y) \wedge \mathit{von}(y, 1000)$$
$$\exists x \exists y(\mathit{planet}(x) \wedge \mathit{durchmesser}(x, y) \wedge \mathit{von}(y, 1000) \wedge \mathit{entdecken}(\mathit{herschel}, x))$$

und mittels eines allgemeinen Gesetzes wie z.B. $p \wedge q \leftrightarrow q \wedge p$ als äquivalente Formeln bewiesen werden. Im Generierungsmodus können so aus einer der beiden oberen Formeln verschiedene natürlichsprachliche Sätze erzeugt werden.

Theorembeweisen

Die gerade erwähnte Gesetzmäßigkeit gehört zur Klasse der Tautologien. Das sind logische Theoreme oder Formeln, die unter jeder Wahrheitswertbelegung zu 1 ausgewertet werden. Ein automatisches Beweisverfahren für die Frage, ob es sich bei einer gegebenen Formel um eine Tautologie handelt, nennt man einen Theorembeweiser. Theorembeweisen gehört zwar nicht zum Kernbereich der Computerlinguistik, doch als Teilgebiet der wahrheitsfunktionalen Semantik und als Prologanwendung (siehe Einleitung) erscheint es uns instruktiv genug, um uns eine prototypische Implementierung eines Theorembeweisers anzusehen[2]. Da eine Implementierung für die Prädikatenlogik wesentlich umfangreicher wäre und es uns nur um eine Darstellung der grundlegenden Prinzipien beim Theorembeweisen geht, wird das Beweisverfahren nur für aussagenlogische Formeln expliziert. Das folgende Programm inkorporiert zwei wesentliche Strategien beim Theorembeweisen. Die erste und klassische Vorgehensweise, Formeln zu beweisen, besteht in der primären Verwendung von **Wahrheitstafeln**. Die Prozedur, die diese Strategie implementiert, nennen wir "theorem_wahrheitswert". Das alternative Verfahren wird in konventioneller Terminologie als **Tableauverfahren** bezeichnet und ist als Prozedur "theorem_tableau" in unserem Programm realisiert. Das Programm besteht aus drei Teilen. Das Modul "vars" berechnet zu einer Formel deren Aussagenvariablen:

```
vars( Ausdruck , Variablen) :-
      Ausdruck =.. [_, Formel1, Formel2 ],
     !,
      vars( Formel1, Var1),
      vars( Formel2, Var2),
       vereinigung(Var1, Var2, Variablen).
vars( ¬  Formel , Variable) :-
     !,
      vars( Formel, Variable).
vars(V, [V] ).
```

Die Berechnung von[3]

```
?- vars((q & p <-> p & q), Var).
```

ergibt daher: Var = [p.q]

Die Beispielsformel stellt die Übersetzung der Formel $p \wedge q \leftrightarrow q \wedge p$ in Prolognotation dar (Die Äquivalenz beider Ausdrücke beruht wiederum auf einer Tautologie). Das zweite

[2] Außerdem verwenden wir Teile des Theorembeweisers im Kapitel über aussagenlogische Merkmalskodierung.
[3] Die logischen Junktoren =>, <->, &, v, ¬ sind als Prologoperatoren definiert, siehe Anhang.

Modul "belegung" sucht eine Belegung zu den Aussagenvariablen. Dies geschieht über eine Zuordnung der beiden Wahrheitswerte "1" und "0":

```
belegung([],[]).
belegung([V|Rest], [[V,0]|Reste]):- belegung(Rest,Reste).
belegung([V|Rest], [[V,1]|Reste]):- belegung(Rest,Reste).
```

Der Aufruf

```
?- belegung([p,q], Wert).
```

ergibt so die Antwort (eine von vier möglichen):

```
Wert = [ [ p, 1 ], [ q, 1 ] ].
```

Das dritte und umfangreichste Modul "tabelle" nimmt die Formel F und einen Wert für deren Aussagenvariablen und berechnet einen Wahrheitswert für die gesamte Formel. Um diese Aufgabe erfüllen zu können, muß das Programm über Regeln verfügen, die die Wahrheitsbedingungen für jeden aussagenlogischen Junktor beschreiben. Diese Regeln stellen also die klassischen Wahrheitstafeln dar:

```
tabelle(  P & Q , B, 1):-    tabelle(P, B, 1),  tabelle(Q, B, 1).
tabelle(  P &  _ , B, 0):-   tabelle(P, B, 0).
tabelle(  _ &  Q , B, 0):-   tabelle(Q, B, 0).

tabelle(  P v Q , B, 0):-    tabelle(P, B, 0),  tabelle(Q, B, 0).
tabelle(  P v _ , B, 1):-    tabelle(P, B, 1).
tabelle(  _ v Q , B, 1):-    tabelle(Q, B, 1).

tabelle(  P => Q , B, 0):-   tabelle(P, B, 1), tabelle(Q, B, 0).
tabelle(  P => _ , B, 1):-   tabelle(P, B, 0).
tabelle(  _ => Q , B, 1):-   tabelle(Q, B, 1).

tabelle(  P <-> Q , B, 1):-  tabelle(P, B, 1),  tabelle(Q, B, 1).
tabelle(  P <-> Q , B, 1):-  tabelle(P, B, 0),  tabelle(Q, B, 0).
tabelle(  P <-> Q , B, 0):-  tabelle(P, B, 1),  tabelle(Q, B, 0).
tabelle(  P <-> Q , B, 0):-  tabelle(P, B, 0),  tabelle(Q, B, 1).

tabelle( ¬ F , B, 0):-       tabelle(F, B, 1).
tabelle( ¬ F , B, 1):-        tabelle(F, B, 0).

tabelle( F , B, Wert):-      atom(F),   member([F,Wert],B),   !.
```

Der Aufruf von "tabelle" erfolgt z.B. mit:

```
?- tabelle( p & q, [ [ p, 1 ], [ q, 1 ] ], Wahrheitswert ).
```

und ergibt so die Zuordnung

```
Wahrheitswert = 1
```

Der Vorteil einer Implementierung in Prolog besteht, wie man weiß, in der prinzipiellen Reversibilität der Programme. So kann man "wahr" auch ohne eine Belegung für die Werte der aussagenlogischen Variablen aufrufen und sich dabei Werte berechnen lassen, für die die Formel konsistent ist.
Tautologien sind diejenigen Formeln, die unter jeder Wahrheitswertbelegung zu 1 ausgewertet werden. In Prolog läßt sich dies nicht direkt überprüfen. Man fragt stattdessen nach allen Beweisen der negierten Formel, also, ob es wenigstens eine Belegung gibt, unter der die Formel zu 0 ausgewertet werden kann. Ist sichergestellt, daß diese Fragen immer entweder erfolgreich sind, oder aber die Berechnung nach endlich vielen Schritten abbricht, so braucht man das Ergebnis nur noch in der umgekehrten Weise zu deuten:

```
theorem_wahrheitswert(Formel):-
        vars(Formel,Variablen),
         belegung(Variablen, VariablenWahrheitswerte),
        tabelle( Formel, VariablenWahrheitswerte, 0),
      !,
       fail.
theorem_wahrheitswert(_).
```

Das Programm berechnet also zu einer Formel zunächst deren Aussagenvariablen. Dann wird diesen Variablen eine Belegung zugeordnet und schießlich überprüft, ob diese Belegung die Formel zu 0 auswertet.

Das Tableau-Verfahren läßt sich darauf aufbauend realisieren, indem man die drei Teilprogramme in anderer Weise anordnet:

```
theorem_tableau(Formel):-
        vars(Formel,Variablen),
        tabelle( Formel, VariablenWahrheitswerte, 0),
         belegung(Variablen, VariablenWahrheitswerte),
      !,
       fail.
theorem_tableau(_).
```

Nun werden für eine Formel zunächst Belegungen ihrer Aussagenvariablen gesucht, die diese erfüllen. Danach wird überprüft, ob dabei ein Widerspruch auftrat. Dieses zweite Verfahren ist wesentlich effizienter als das Wahrheitstafelverfahren. Dies liegt daran, daß das Wahrheitstafelverfahren blind Belegungen für die Aussagenvariablen generiert und die Formel damit testet. Dagegen berechnet das Tableauverfahren alle Belegungen für einen vollständigen Beweis als Seiteneffekt und muß diese nur noch auf Widersprüchlichkeit hin überprüfen. Dies geschieht im Programm über den kleinen Trick, daß "belegung" zwei Listen gleicher Länge als Argumente benötigt, um erfolgreich zu terminieren. Liegt aber ein Widerspruch vor, so ist die Liste der Variablenbelegungen länger als die der Variablen, da dann zumindest eine Aussagenvariable mit 0 *und* 1 bewertet worden ist.
Versuchen wir den Unterschied zwischen den beiden Verfahren etwas präziser zu beschreiben: Sei die Zuordnung

p = 1, q = 1

eine mögliche Belegung der Formel

p & q <-> q & p

Die Idee hinter dem Tableauverfahren ist nun, widersprüchliche Belegungen zu finden, unter denen eine Formel wahr ist, bzw. in der Prologübersetzung, Belegungen zu finden, die die negierte Formel falsifizieren. Im Programmablauf entspricht der folgende Aufruf der Prozedur "Tabelle" dieser Konzeption:

T:

```
Call: tabelle(p&q<->q&p,X,0) ?
Exit: tabelle(p&q<->q&p,[[p,1],[q,1],[q,0]|_2931],0) ?
```

Das Resultat besteht hier in einer Belegung der Aussagenvariablen, die die Formel offensichtlich nicht falsch machen kann, da eine Aussagenvariable zwei verschiedene Wahrheitswerte zugeordnet bekommen hat. Dies ist aber in der Standardlogik (Gesetz vom ausgeschlossenen Dritten) nicht erlaubt. Es gilt: q oder nicht q. Genau das wird im Tableauverfahren durch die nachfolgende Bedingung "belegung" geprüft. Im Gegensatz dazu startet das Wahrheitstafelverfahren mit einer beliebigen Struktur und versucht damit die Formel zu falsifizieren, also z.B.:

W:

```
Call: tabelle(p&q<->q&p,[[q,0],[p,0]],0) ?
Fail: tabelle(p&q<->q&p,[[q,0],[p,0]],0) ?
```

Dieser Versuch mißlingt direkt, da die Belegungen schon vorhanden sind. Allerdings ist der Nachweis aufwendiger, da nun erst alle Wahrheitswertzuweisungen der Junktoren der Formel durchgespielt werden müssen.

Der Unterschied zwischen den beiden Strategien besteht also darin, daß im klassischen Wahrheitstafelverfahren alle Wahrheitswerte von Atomen und komplexen Ausdrücken (im Beispiel W die Belegung für p, q, <-> und &) bekannt sein müssen, während für das Tableauverfahren gerade zwei (widersprüchliche) Belegungen einen atomaren Formel ausreichen (im Beispiel T die beiden Belegungen für q) um abzubrechen.

Natürlich läßt sich das Tableauverfahren weiter optimieren, etwa indem man die weitere Berechnung einer Belegung sofort abbricht, wenn sie zwei unvereinbare Paare enthält und nicht bis zum Aufruf der Prozedur "belegung" wartet. Das Auffinden widersprüchlicher Belegungen einer Aussagenvariablen findet in der Liste statt, die im Programm "VariablenWahrheitswerte" genannt wurde. Die Stelle, an der diese Überprüfung frühestens vorgenommen werden kann, ist beim Aufruf von "member" unterhalb der letzten Definition von "tabelle". Um die Überprüfung zu bewerkstelligen, mache man sich nun zunächst klar, daß die Feststellung, die Liste möge keine widersprüchlichen Belegungen von Aussagenvariablen enthalten, äquivalent ist zu der Forderung, daß in der Liste keine Aussagenvariable mehr als einmal vorkommt. Dann ist die weitere Vorgehensweise sofort einsichtig. Als erstes muß die Vorbedingung für die Forderung geschaffen werden - die Liste als Struktur mit endlicher Länge. Das erhält man über die folgende Erweiterung der Prozedur "vars", die nun "vars0" heißt und zusätzlich eine Liste von Variablen (EndlicheListe) vorgeneriert, deren Länge der Anzahl der in der Formel vorkommenden Aussagenvariablen entspricht:

```
vars( Ausdruck , Variablen, EndlicheListe) :-
      vars0(Ausdruck , Variablen),
    !,
      length(Variablen,Laenge),
      length(EndlicheListe,Laenge).
```

Über diese Liste läßt sich nun die Verschiedenheits-Forderung wie folgt definieren:

```
not_member(_,[]).
not_member(X,[[H,_]|T]):-
     dif(X,H),
      not_member(X,T).
```

```
alle_verschieden([]).
alle_verschieden([[X,_]|T]):-
      not_member(X,T),
       alle_verschieden(T).
```

Die Prozedur "not_member" legt für eine Liste aus Paaren (Aussagenvariable und dazugehöriger Wahrheitswert) fest, daß das erste Argument eines jeden Paares in der Liste verschieden sein muß von einem gegebenen Term X. Die Forderung der Verschiedenheit wird über die Verwendung von "dif" laufzeitunabhängig gemacht. Die Prozedur "alle_verschieden" definiert diese Verschiedenheits-Forderung für alle möglichen Paar-Paar-Beziehungen der Liste. Nun müssen diese Änderungen nur noch in das Gesamtkonzept des Tableauverfahrens eingebracht werden:

```
theorem_tableau(Formel):-
        vars(Formel,Variablen,VariablenWahrheitswerte),
        alle_verschieden(VariablenWahrheitswerte),
       tabelle( Formel, VariablenWahrheitswerte, 0),
        belegung(Variablen, VariablenWahrheitswerte), % redundant
     !,
      fail.
theorem_tableau(_).
```

Dabei ist zu beachten, daß für die effizienteste Überprüfung auf Widersprüchlichkeit der Aufruf von "alle_verschieden" vor dem Aufruf von "tabelle" zu erfolgen hat. Die Ausführung von "dif" wird ja ein zweites Mal aktiviert, wenn dessen Argumente zu nichtvariablen Termen instantiiert werden und das ist der Fall, wenn versucht wird, unterhalb der letzten Definition von "tabelle" (per "member") ein Element der Liste hinzuzufügen. Schließlich ist anzumerken, daß durch diese Modifikationen der Aufruf von "belegung" innerhalb von "theorem_tableau" redundant geworden ist und aus der Definition entfernt werden kann.

Übersetzung Syntax-Semantik

Mit der Prädikatenlogik kennen wir nun die Zielsprache, in die wir natürlichsprachliche Sätze überführen wollen. Was noch fehlt ist nun natürlich eine Beschreibung, wie diese Übertragung geschehen soll. Die wohl bekannteste Aussage über die zu wählende Strategie stammt von Frege, wonach die Bedeutung eines Satzes sich aus den Bedeutungen seiner Teilausdrücke ermitteln läßt. Diese Vorgehensweise wird auch als **Kompositionalitätsprinzip** bezeichnet. Man versteht darunter, daß die syntaktischen und semantischen Teilausdrücke in einer parallelen Weise zusammengesetzt werden, so daß es immer eine 1-zu-1-Entsprechung zwischen einer syntaktischen und einer semantischen Regel gibt. Wir wollen anhand des folgenden Satzes und seiner Übersetzung dies versuchen:

jeder astronom entdeckte einen planeten

fuer_alle(X,astronom(X) => existiert(Y, planet(Y) & entdecken(X,Y)))

Wir gehen den Satz von links nach rechts durch und übersetzen die Worte anhand der Festlegungen des letzten Kapitels.

Lexikonregeln der Semantik:

1. "jeder" wird als fuer_alle(A, R => S) übersetzt.
   "einen" wird als existiert(E, P & Q) übersetzt.
2. "astronom" wird als astronom(B) übersetzt.
   "planet" wird als planet(U) übersetzt
3. "entdeckte" wird als entdecken(C,D) übersetzt.

Damit haben wir auf semantisch-lexikalischer Ebene eine Annäherung an die Zielformel bekommen. Nun müssen wir sagen, wie wir die einzelnen Wortübersetzungen so kombinieren, daß wir die gesamte Formel in der obigen Form erhalten.

Syntaxregeln der Semantik:

I. Das einstellige Prädikat nach Lexikonregel 2 wird in den Quantor nach Regel 1 eingesetzt, indem wir es durch die linke Variable innerhalb der binären Verknüpfung substituieren.

II. Ein Verb wie in Lexikonregel 3 wird mit Ausdrücken nach Syntaxregel I kombiniert, indem man das Verb durch die rechte Variable innerhalb der binären Verknüpfung eines Quantors substituiert.

III. Zwei Quantoren werden kombiniert, indem man den einen Quantor durch die rechte Variable innerhalb der binären Verknüpfung des anderen Quantors substituiert.

Die Anwendung von I ergibt:

fuer_alle(A, astronom(B) => S)
existiert(E, planet(U) & Q)

Die Anwendung von II ergibt:

existiert(E, planet(U) & entdecken(C,D))

Die Anwendung von III ergibt:

fuer_alle(A, astronom(B) => existiert(E, planet(U) & entdecken(C,D)))

Wir stellen folgende Analogien fest:

Syntaxregel I entspricht np --> det , n.
Syntaxregel II entspricht vp --> v, np.
Syntaxregel III entspricht s --> np, vp.

Wir können also analog zu den Operationen der natürlichsprachlichen Syntax semantische Operationen aufbauen. Die Formel, die wir auf diese Weise konstruiert haben, entspricht der Vorgabe bis auf die Benennung der Variablen. Bei den Einsetzungen nach den Syntaxregeln I - III muß also zusätzlich festgelegt werden, welche Variablen gleichen Namens sein sollen. Um diese Variablenumbenennungen in einer allgemeinen Weise zu berechnen, bedient man sich in der Semantik eines bestimmten Verfahrens, das eine ähnliche Funktion wie Prologunifikation besitzt: Lambda-Abstraktion und Reduktion.
Ein Ausdruck wie "astronom(X)" wird über seine Variable abstrahiert:

$\lambda X.astronom(X)$

Dies erlaubt einen direkten Zugriff auf die Variable im Inneren des Lambdaterms. Will man der Variablen einen bestimmten Wert zuweisen, so wendet man ihn auf den Term an,

indem man den Wert dahinter schreibt. Der Term wird dann Funktor, der Wert Argument genannt:

$$(\lambda X.astronom(X))(p)$$

Die Regel der Beta-Reduktion

$$(\lambda X.\phi)p \Rightarrow [\phi]\{X = p\}$$

sorgt dann für die "Unifikation" von "p" mit allen Vorkommnissen von "X" in der Formel, so daß die Formel

```
astronom(p)
```

resultiert. Dieses Verfahren ist nun so allgemein, daß wir es für die Kombination der semantischen Ausdrücke verwenden wollen. Da wir im Prolograhmen arbeiten, wählen wir wie bei den prädikatenlogischen Ausdrücken eine Termrepräsentation der Lambdaausdrücke:

```
X ^ astronom(X))
```

Außerdem benützen wir für die Anwendung eines Lambdaterms auf einen anderen Term eine Regel, die Prologunifikation für die Substitutionsoperation verwendet:

```
beta_reduktion( X ^ P, X, P).
```

Mit diesen Festlegungen können wir nun ein simples Programm angeben, das einen syntaktischen Baum als Eingabe nimmt und anhand der eben skizzierten Vorgehensweise unter Verwendung von Lambdaausdrücken prädikatenlogische Formeln konstruiert.

```
semantische_analyse(SyntaktischerBaum, Praedikatenlogik):-
        syn_sem(SyntaktischerBaum, Praedikatenlogik).


syn_sem(v([Wort,_,_]), Logik):-
    Logik = (X^V^NP)^(Y^NP ),
    praedikate(Wort, X^Y^V).

syn_sem(n([Nomen,_]), Logik ):-
    praedikate(Nomen, Logik ).

syn_sem(det(dets([_,Typ|_])), Logik ):-
    Logik = ((X^N)^((X^VP)^Det)),
    quantoren(Typ, X^N^VP^Det).

syn_sem(SynBaum, Sem):-
    SynBaum =.. [_, Links, Rechts],
    semantische_analyse(Links, LSem),
    semantische_analyse(Rechts, RSem),
    beta_reduktion(LSem, RSem, Sem).
syn_sem(SynBaum, Sem):-
    SynBaum =.. [_, Tochter],
    semantische_analyse(Tochter, Sem).
syn_sem([], X^X).


beta_reduktion(P^Q, P, Q).
beta_reduktion(P,P^Q, Q).


praedikate(astronom, X^astronom(X)).
praedikate(astronomen, X^astronom(X)).
praedikate(planet, X^planet(X)).
praedikate(planeten, X^planet(X)).
praedikate(entdeckte, X^Y^entdecken(X,Y)).

quantoren(indef, X^R^S^existiert(X,R&S)).
quantoren(all, X^R^S^fuer_alle(X,R=>S)).
quantoren(frage,X^ R^S^frage(X,R&S)).
```

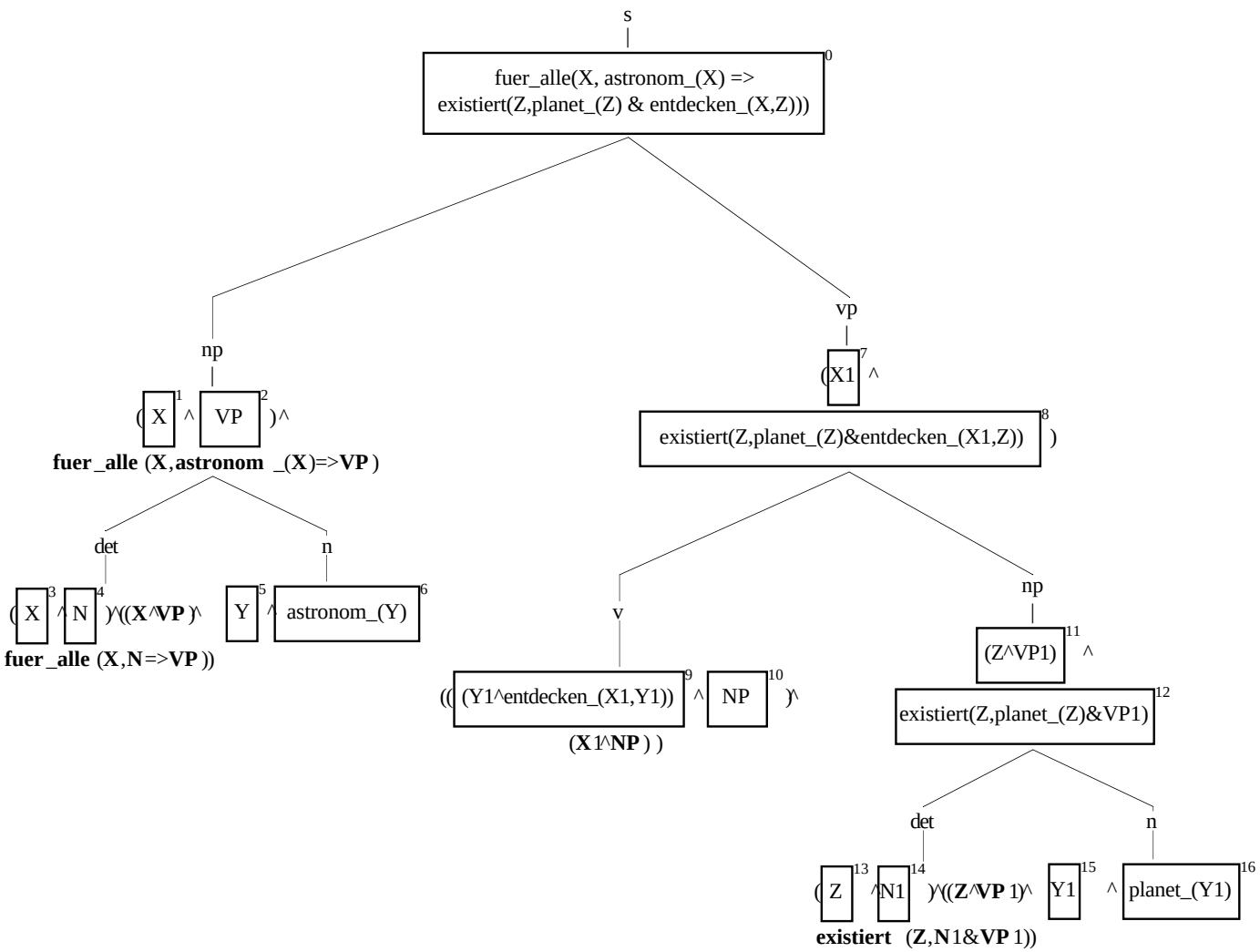
Gegeben den Satz "jeder astronom entdeckte einen planeten", erhalten wir als Ergebnis der Syntaxanalyse:

```
s(np(det(dets([jeder,all,[nom,sing,masc]])),
     n1(n([astronom,[nom,sing,masc]]),
       radj(rs([])))),
  vp(vp2(aux(v([entdeckte,finit,[[nom,sing,masc],[akk,sing,masc]]])),
         vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),
                                 n1(n([planeten,[akk,sing,masc]]),
                                   radj(rs([])))),
                                nprec([]))),
           vg(v([]),aux([])))),
    nachfeld([])))
```

Dieser Term wird mit dem ersten Argument der Relation "semantische_analyse" unifiziert und rekursiv abgearbeitet. Der Vorgang beginnt am obersten Knoten des Baumes und zerlegt diesen in eine linke Tochter "np" und eine rechte Tochter "vp". Da beide Töchter selbst komplex sind werden diese erneut der Zerlegung unterworfen. Die Betareduktion semantischer Ausdrücke, die aus komplexen syntaktischen Bäumen resultieren, findet erst statt, nachdem die einfacheren Teilbäume reduziert worden sind. Dies liegt daran, daß im Programm der Reduktionsaufruf nach den rekursiven Aufrufen von "semantische_analyse" kommt. Wenn durch die Zerlegung eines Knotens zwei terminale Töchter entstehen, werden deren lamdaabstrahierte Terme mittels der Prozedur "beta_reduktion" direkt aufeinander angewendet. Man benötigt also nicht für jeden syntaktischen Knoten eine separate Regel, sondern kann eine rekursive Prozedur benutzen wie die drittletzte "syn_sem"-Regel. Alle Syntaxregeln, die leer laufen, werden über die letzte Definition von "syn_sem" abgefangen und sorgen bei der Betareduktion für eine identische Abbildung des nichtleeren Ausdrucks.

Das nächste Schaubild illustriert ein Beispiel für die syntaktisch-semantische Übersetzung. Der Unifikationsweg spielt sich dabei wie folgt ab (die fett unterlegten Ausdrücke stellen die Resultate der Lambdareduktion dar):

- Ausdruck^{3} unifiziert mit Ausdruck^{5}
- Ausdruck^{4} unifiziert mit Ausdruck^{6}
- Ausdruck^{13} unifiziert mit Ausdruck^{15}
- Ausdruck^{14} unifiziert mit Ausdruck^{16}
- Ausdruck^{9} unifiziert mit Ausdruck^{11}
- Ausdruck^{10} unifiziert mit Ausdruck^{12}
- Ausdruck^{1} unifiziert mit Ausdruck^{7}
- Ausdruck^{2} unifiziert mit Ausdruck^{8}



Die Abbildung zeigt die Konstruktion einer prädikatenlogischen Formel mithilfe von Lambdaausdrücken anhand eines einfachen Satzes wie "jeder astronom entdeckte einen planeten". Der syntaktische Baum wurde aus Darstellungsgründen gekürzt und zeigt nur die nicht-leeren Knoten und die Knotenbenennungen.

Weil die Verteilung von Funktor und Argument nicht auf eine Abfolge (z.B. Links-Rechts) festgelegt ist, braucht man zwei Regeln für die Betareduktion. Dies läßt nun aber Beta-Reduktionen zu, die nicht gewollt sind, wenn der Funktor als Argument und das Argument als Funktor interpretiert wird. So kann aus den beiden Teilausdrücken

```
Y^astronom(Y)
```

und

```
((X^N)^((X^VP)^fuer_alle(X,N=>VP)))
```

statt

```
((X^VP)^fuer_alle(X,astronom(X)=>VP))
```

also leider auch

```
astronom(((X^N)^((X^VP)^fuer_alle(X,N=>VP))))
```

entstehen.

Eine Möglichkeit nur die korrekten Reduktionen zu erhalten, besteht darin, die Lambda-Ausdrücke zu typisieren und dabei nur die korrekten Funktor-Argument-Anwendungen als kompatible Typanwendung zu definieren.

Bei der Typisierung unserer semantischen Ausdrücke orientieren wir uns im wesentlichen an einem System nach Montague. Danach ergeben sich die folgenden Zuordnungen:

Typ	Bezeichnung bei Montague	Syntaktische Kategorie	Beispiel
t	t	s	herschel entdeckte uranus
(e,t)	CN	n	astronom
((e,t), ((e,t),t))	T/CN	dets	der
(((e,t),t), (e,t))	TV	v, komparativ	entdeckte, groesser
((e,t),t)	T	np	herschel, ein astronom
(((e,t),t), ((e,t),(e,t)))	IAV/T	prep	von
((e,t), ((e,t),(e,t)))		rp	den
(E,E)		aux	hat, ist, als, es

Der Typ (E,E) in der letzten Zeile der Tabelle ist speziell auf unser Programm zugeschnitten. Er erlaubt über die Verwendung von Prologvariablen eine identische Abbildung bei einer Betareduktion, so daß ein Argument dieses Ausdrucks unverändert aus der Reduktion hervorgeht. Dies gestattet uns, Wörtern wie "hat", "als" und "es" eine leere semantische Funktion zuzuordnen. Die Programmerweiterung für die Miteinbeziehung der obigen Typen

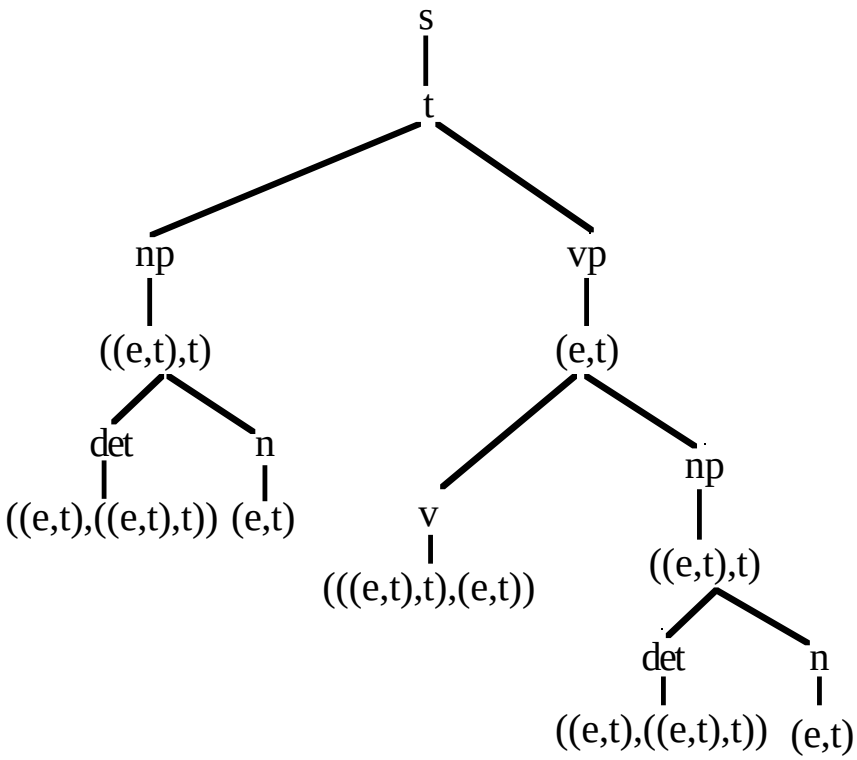
ist trivial, da man nur das semantische Lexikon so erweitern muß, daß die lambdaabstrahierten Variablen mit der Typinformation angereichert werden. So entsteht z.B. aus der Definition der Artikelsemantik ohne Typinformation:

```
syn_sem(det(dets([_,Typ|_])), Logik ):-
    Logik = ((X^N)^((X^VP)^Det)),
    quantoren(Typ, X^N^VP^Det).
```

die getypte Variante, indem man den zweiten Parameter von syn_sem als Liste kodiert, die nun sowohl Repräsentationsinformation wie den prädikatenlogischen Ausdruck als auch Typinformation enthält:

```
syn_sem(Syntax, [Logik, LogikTyp]):-
      Syntax = det(dets([Name,Typ,_])),
      Logik = (X^N)^((X^VP)^ Det),
      LogikTyp =  ((e,t) ,((e,t),t)),
       quantoren(Name,Typ, X^N^VP^Det).
```

Der folgende Ableitungsbaum zeigt unter Auslassung der leeren syntaktischen Knoten die Typenkombination für einen Satz wie "jeder astronom entdeckte einen planeten".



Umseitig folgt das entsprechend modifizierte Programm für die semantische Übersetzung.

```
semantische_analyse(Syntax,  Logik):-
            syn_sem(Syntax, Logik).

syn_sem(Syntax,  [Logik,  LogikTyp]):-
        Syntax = v([Wort,_,[[nom|_],[_|_]]]),
        Logik = ((Y^V)^NP)^(X^NP),
        LogikTyp = (((e,t),t) , (e,t)),
        praedikate(Wort, X^Y^V).

syn_sem(Syntax,  [Logik,  LogikTyp]):-
        Syntax = n([Wort,_]),
        Logik = X^N,
        LogikTyp =  (e,t),
        praedikate(Wort, X^N).

syn_sem(Syntax,  [Logik,  LogikTyp]):-
        Syntax = det(dets([Name,Typ,_])),
        Logik = (X^N)^((X^VP)^ Det),
        LogikTyp =  ((e,t) ,((e,t),t)),
        quantoren(Name,Typ, X^N^VP^Det).

syn_sem(Syntax,  [Logik,  LogikTyp]):-
        Syntax = prep([Wort|_]),
         Logik = ((X^V)^NP) ^ ((Y^N)^(Y^(N&NP))),
        LogikTyp = (((e,t),t) , ((e,t),(e,t))),
        praedikate(Wort, X^Y^V).

syn_sem(Syntax,  [Logik,  LogikTyp]):-
        Syntax = rp([_,_,_]),
         Logik = (Y^VP) ^ ((X^N)^(X^(N&VP))),
        LogikTyp =  ((e,t) , ((e,t),(e,t))),
       X=Y.

syn_sem(Syntax,  [Logik,  LogikTyp]):-
        Syntax = en([Name,_]),
        Logik = (Name^VP)^VP,
        LogikTyp =  ((e,t),t),
        typ_name(Name).

syn_sem(komparativ([Wort,Subkat]),  Logik ):-
        syn_sem(v([Wort,_,Subkat]), Logik ).

syn_sem(en([es,_]),  [X^X,(E,E),C/C]).

syn_sem(SynBaum,  Sem):-
        SynBaum =.. [_, Links, Rechts],
         semantische_analyse(Links, LSem),
         semantische_analyse(Rechts, RSem),
        beta_reduktion(LSem, RSem, Sem).
syn_sem(SynBaum,  Sem):-
        SynBaum =.. [_, Tochter],
         semantische_analyse(Tochter, Sem).
syn_sem([],  [X^X,(E,E),C/C]).
syn_sem([Wort|_],  [X^X,(E,E),C/C]):-
        member(Wort,[hat,haben,ist,sind,als]).

beta_reduktion( [P,E], [P^Q,(E,T)],  [Q,T]).
beta_reduktion( [P^Q,(E,T)],  [P,E], [Q,T]).

praedikate(von,  X^Y^gleich(X,Y)).
praedikate(von,  X^Y^besitzen(X,Y)).
praedikate(groesser,  X^Y^groesser(X,Y)).
praedikate(kleiner,  X^Y^kleiner(X,Y)).
praedikate(astronom,  X^astronom(X)).
praedikate(astronomen,  X^astronom(X)).
praedikate(planet,  X^planet(X)).
praedikate(planeten,  X^planet(X)).
praedikate(entdeckte,  X^Y^entdecken(X,Y)).
praedikate(entdeckten,  X^Y^entdecken(X,Y)).

quantoren(_,indef,  X^N^VP^existiert(X,N&VP)).
quantoren(_,all,  X^N^VP^fuer_alle(X,N=>VP)).

typ_name(uranus).
typ_name(herschel).
```

Übungsaufgabe 13:
Wir haben zuvor gesehen, wie man rein semantisch den Superlativ in Prädikatenlogik ausdrücken kann. Implementieren Sie eine superlativische Konstruktion wie z.B.

```
welcher durchmesser ist am groessten
```

syntaktisch und semantisch.

Übungsaufgabe 14:

Syntaktisch ist es möglich, die relativische Konstruktion mit "dessen" zu verwenden:

```
welchen planeten dessen durchmesser groesser als 3000 kilometer ist hat herschel entdeckt
```

Implementieren sie dessen (!) semantische Übersetzung.

Übungsaufgabe 15:

Die extraponierten Konstruktionen Relativsatz und Komparativ sind nur syntaktisch implementiert. Konstruieren Sie ihre semantische Übersetzung. Gehen Sie dabei von einem normalisierten Syntaxbaum für extraponierte und nicht-extraponierte Strukturen aus.

Übungsaufgabe 16:

Zeigen Sie, daß der zuvor diskutierte nicht wohlgeformte Ausdruck

```
astronom(((X^N)^((X^VP)^fuer_alle(X,N=>VP))))
```

nach der Typisierung der semantischen Ausdrücke nicht mehr konstruiert werden kann.

Syntaktische Tiefenstruktur und Semantik

Die syntaktischen Strukturen unserer Grammatik reflektieren die Reihenfolge und Zusammensetzung der Konstituenten und der terminalen Ausdrücke. Insofern kann man hier von einer direkten Entsprechung von Syntax und Wortkette reden. Diese Syntax enthält als nicht-wortkettenorientierte Elemente nur die leeren terminalen Knoten, deren Einführung eine rein technische Bequemlichkeit darstellt. Wie man gleich sehen wird, gibt es technische Probleme, die es nahelegen, sich weiter von der oberflächenorientierten Syntax zu entfernen. Der Grundgedanke einer tiefenstrukturellen Syntax besteht letztlich darin, aus einer kompakten Menge möglichst allgemeiner Regeln mithilfe einiger Umformungen (Transformationen) zu der vollen Anzahl von Strukturbeschreibungen zu gelangen. Dieser Gedanke ist z.B. in der generativen Grammatik Chomskys enthalten, aber auch in neueren Formalismen wie z.B. der GPSG, in der Metaregeln die Funktion von Transformationen übernehmen.

Über die kompakte Darstellung hinaus bietet eine Syntax, die zwischen Oberflächen- und Tiefenstruktur unterscheidet, Vorteile für die Übersetzung in eine semantische Repräsentation. Diese kann auf einer kleinen Menge verschiedener Strukturen erfolgen und muß nicht die volle Anzahl möglicher Oberflächenformen vorsehen. Anhand von zwei Beispielen aus unserem Fragment läßt sich dies demonstrieren. Da es notwendig wird, die syntaktische Repräsentation zu modifizieren, erfolgen die Änderungen nicht auf dem Format einer DCG, sondern auf der kompilierten Regelebene, auf der außer den DCG-Konstrukten auch die Differenzliste und die Baumpositionen vorhanden sind:

- Extraponierte Relativsätze und Komparativphrasen

Relativsätze entstehen als Ergänzung eines Nomens. Ihre Stellung ist variabel: Sie werden entweder postnominal realisiert oder wandern ins Nachfeld des Satzes, dessen Bestandteil das Nomen ist. In der postnominalen Stellung läßt sich der Relativsatz semantisch direkt konstruieren: Zusammen mit den anderen Attributen des Nomens bildet er eine Konjunktion von prädikatenlogischen Formeln. Besetzt der Relativsatz das Nachfeld, befindet er sich nicht mehr im Bereich seines Nomens und kann nur noch indirekt damit in Zusammenhang gebracht werden. Dazu muß

- das Kopfnomen des verschobenen Relativsatzes im Syntaxbaum gefunden werden, was einen zusätzlichen Aufwand darstellt.

- eine spezielle Interpretationsprozedur für die Konstruktion des semantischen Ausdrucks geschrieben werden, was die Semantik unnötig kompliziert.

Die Nachteile, die mit einer solchen Lösung verbunden sind, lassen sich vermeiden, wenn man den Relativsatz auf tiefenstruktureller Ebene als nicht-verschoben betrachtet und diese

Ebene als Eingabe für die Berechnung der semantischen Repräsentation dient. In unserem Ansatz ist es der syntaktische Baum, der diese Ebene darstellt. Die Aufgabe reduziert sich also darauf, den extraponierten Relativsatz im syntaktischen Baum an seinem Ursprungsort zu belassen. Somit erhält man als "syntaktische" Tiefenstruktur für den extraponierten Relativsatz dieselbe Baumposition wie für den unverschobenen Relativsatz.

Die ursprüngliche Regel unseres Fragments sah vor, anstelle des (verschobenen) Relativsatzes einen leeren postnominalen Eintrag im syntaktischen Baum vorzunehmen:

```
rs(rel, A, A, rs([]), B, B).
```

Die revidierte Regel sieht dagegen eine Baumstruktur vor, die einem nicht-verschobenen Relativsatz entspricht: ein binär verzweigender rs-Knoten (in der Regel fett abgesetzt):

```
rs(rel, A, rs(X,Y,A), rs(X,Y), B, B).
```

Die Instantierung der Töchter dieses Teilbaumes mit der tatsächlichen Struktur des verschobenen Relativsatzes erfolgt über die Weitergabe identischer Variablen (in der Regel unterstrichen) in die Extrapositionsliste.

Für verschobene Teile von Komparativphrasen können wir den gleichen Algorithmus wie für Relativsätze verwenden. Damit erhält die Regel, die die als-Phrase leerlaufen läßt, eine syntaktische Repräsentation (fett abgesetzt), die über die Extrapositionsliste aus dem Nachfeld instantiert wird:

```
als_np(A, B, [als_np([als,A,B,[]],E)], als_np([als,A,B,[]],E), C, C).
```

Da wir extraponierbare Relativsätze, die aus der Nominalphrase der als-Phrase stammen, nicht in die globale Extrapositionsliste aufnehmen, müssen wir dafür Sorge tragen, daß deren Strukturvariablen (fett abgesetzt), mit Konstanten (leeren Listen) unifiziert werden, die verhindern, daß die semantischen Algorithmen auf Variablen im syntaktischen Baum operieren:

```
als_np(A, B, [], als_np([als,A,B,[]],E), [als|D], C) :-
        np(A, B, Q, E, D, C),
        (Q = []; Q =.. [_,[],[],_]).
```

Die Änderungen, die wir vorgenommen haben, müssen nun noch auf die Prozeduren im Nachfeld abgestimmt werden. Wir geben die Regeln unkommentiert wieder, da die Anpassungen direkt aus dem zuvor gezeigten folgen:

```
nachfeld([], nachfeld([]), B, B).
nachfeld([H|T], nachfeld([]), B, C) :-
        extraposition(H, B, D),
        nachfeld(T, nachfeld([]), D, C).

extraposition([], B, B).
extraposition(rs(X,Y,Z), B, C):-
        rs(rel, Z, [], rs(X,Y), B, C).
extraposition(rs([],[],_), B, B).
extraposition(als_np([als,X,Y,[]],Z), B, C):-
        als_np(X, Y, [], als_np([als,X,Y,[]],Z), B, C).
```

- Verbstellung

Die Kombination von Nominalphrasen und Verb zu einer wohlgeformten prädikatenlogischen Formel ist über die Definition der entsprechenden semantischen Grundausdrücke einigen Restriktionen unterworfen. So kann das semantische Äquivalent einer Nominalphrase nicht direkt mit einer weiteren Nominalphrase zu einem neuen Ausdruck kombinieren, sondern muß zunächst mit dem semantischen Äquivalent eines Vollverbs einen komplexen Ausdruck bilden. Dieser letzte Fall tritt in syntaktischen Bäumen auf, in denen das Vollverb entweder Verb-Zweit-Stellung oder Verb-Letzt-Stellung im Satz einnimmt, also in Bäumen mit der folgenden obersten Struktur:

```
s(Np,Vp)
```

Man erinnere sich dabei daran, daß die Zusammensetzung der semantischen Teilausdrücke zu komplexen Ausdrücken von unten nach oben erfolgt, da erst über die lexikalischen Definitionen die für die Kombination notwendige Typinformation verfügbar wird. D.h., daß erst die Synthese der gesamten Verbalphrasenstruktur (im oberen Beispiel) abgeschlossen sein muß, bevor die oberste Nominalphrase damit kombiniert werden kann. Der kritische Fall tritt in unserem Fragment dann ein, wenn das Vollverb an erster Stelle im Satz steht:

```
s(V,s(Np,Vp))
```

Ist diese syntaktische Struktur vorgeben, dann scheitert das semantische Konstruktionsverfahren daran, daß der rechte Teilbaum "s(Np,Vp)" semantisch nicht übersetzt werden kann, da das Prädikat, das dem Vollverb entspricht, nicht in diesem Baum erscheint und damit zwei Nominalphrasen direkt zu kombinieren wären.

Wie im Falle des extraponierten Relativsatzes verzichten wir aus obengenannten Gründen auf eine Anpassung der semantischen Algorithmen und bringen stattdessen den syntak-

tischen Baum auf eine (Tiefen-) Struktur, die eine Konstruktion der Semantik mit den vorhandenen Mitteln erlaubt. Dazu reicht es aus, die syntaktische Repräsentation des Vollverbs im Baum an eine Stelle zu verschieben, die zwischen den beiden Nominalphrasen liegt. Eine mögliche Lösung ist die folgende:

```
s(Np,s(V,Vp))
```

Wir vertauschen also einfach die Nominalphrase mit dem Verb (bzw. Auxiliar) und erhalten die gewünschte Kompatibilität mit dem semantischen Konstruktionsverfahren.

Auf Ebene der Regeln genügt es, die zweite s-Regel entsprechend zu modifizieren, um dieses Resultat zu erzielen. So wird aus der oberflächennahen Baumstruktur:

```
s(finit, _, s(F,S), A, B) :-
        aux(finit/C, D, F, A, E),
        s(C, D, S, E, B).
```

eine tiefenstrukturelle Darstellung, die den Erfordernissen der Semantik entspricht:

```
s(finit, _, s(Np,s(F,Vp)), A, B) :-
        aux(finit/C, D, F, A, E),
        s(C, D, s(Np,Vp), E, B).
```

Die gerade beschriebenen Modifikationen der Grammatik erfolgen auf dem syntaktischen Baum. Es folgt daher, daß etwaige andere Teilprogramme, die diesen Baum ebenfalls als Informationsstruktur verwenden, dadurch in ihrer Funktionalität nicht eingeschränkt werden dürfen. Das Modul "pragmatische_analyse" etwa greift auf den syntaktischen Baum zu, um die Satzart, also Frage- oder Aussagesatz, zu bestimmen. Die eben erfolgte Modifikation hat jedoch darauf keinen Einfluß.

Semantische Merkmale und Sinnrelationen

Die bisherigen Beschreibungsmittel, die wir benutzt haben, waren allesamt rein syntaktischer Natur: Phrasenstrukturregeln, Merkmale für Numerus, Kasus, Genus, für den Relativsatz, für dessen Verschiebung, Merkmale für die Verbstellung. Mit diesen Mitteln lassen sich zwar eine ganze Menge inkorrekter Wortketten eliminieren. Dennoch bleibt eine Reihe von Sätzen übrig, die damit nicht in den Griff zu bekommen sind. So erzeugt unsere Grammatik z.B. auch den Satz:

herschel hat herschel entdeckt

bzw. mit der Hinzufügung des Verbs "umkreisen" und des Namens "uranus":

uranus hat einen astronomen entdeckt der herschel umkreist

Da wir diese Sätze in der Auswertung nicht interpretieren können, wollen wir sie schon früher ausschließen, indem wir die freie Kombinationsfähigkeit der Verben mit ihren Komplementen beschränken. So wollen wir für den ersten Satz eine Bedingung einführen, die für unser Modell aussagt:

Das Verb entdecken hat als Subjekt einen Astronomen und als Objekt.einen Himmelskoerper.

bzw. für den letzteren Satz:

Das Verb umkreist hat als Subjekt und als Objekt.einen Himmelskoerpere

Mit diesen Einschränkungen wären die beiden oberen Sätze unmöglich, sehr wohl aber noch die folgenden:

herschel hat uranus entdeckt
ein astronom hat einen mond entdeckt der uranus umkreist

Wie man richtig bemerkt, sind diese Einschränkungen keine allgemeinen Axiome. In anderen Kontexten kann z.B. der erste Satz "herschel entdeckt herschel" sehr wohl einen Sinn haben, bzw. eine Interpretation erfahren. Aus diesem Grund sollten wir diese semantischen Einschränkungen so vornehmen, daß sie modularen Charakter haben und leicht gegen andere austauschbar sind, wenn der Kontext sich geändert hat. Auf diese Weise

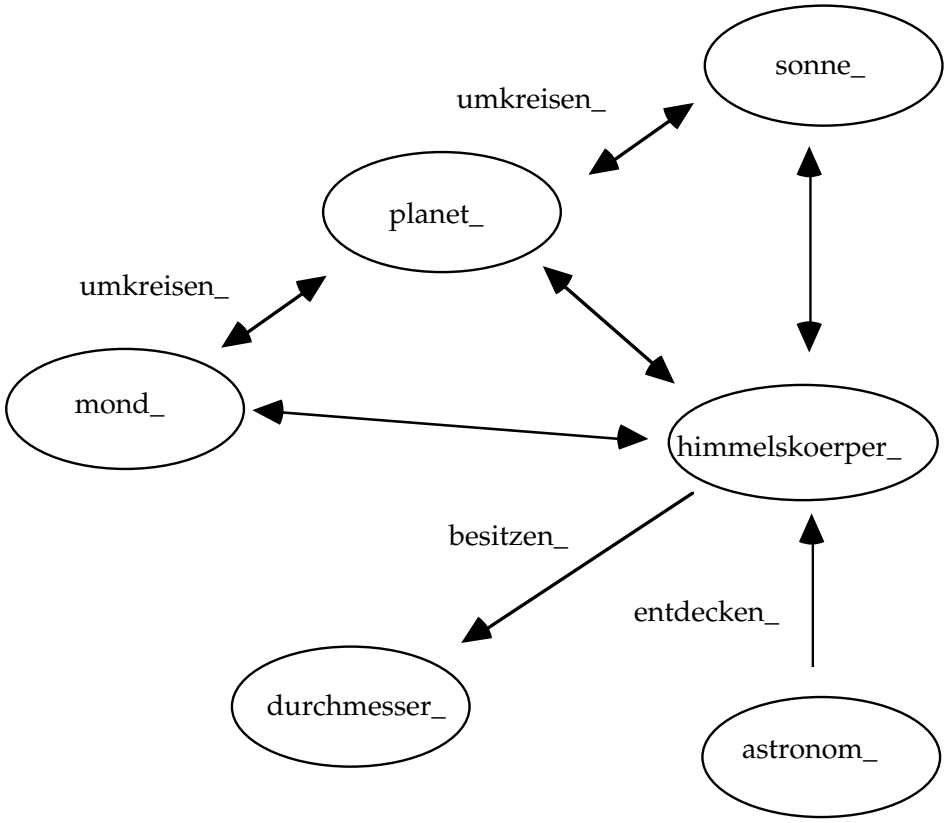
bleiben die reine Syntax uns die repräsentationelle Semantik autonom und stabil über die verschiedenen Anwendungen hinweg. Die Arbeit die man dabei zu leisten hat, wird von Datenbankingenieuren als **Entity-Relationship Modell** oder als **konzeptuelles Schema** bezeichnet. Damit ist gemeint, daß die Spalten einer Datenbank in bestimmten **Sinnrelationen** miteinander stehen und in gewisser Weise miteinander verknüpfbar sind. So kann man z.B. die beiden Entitäten "Himmelskörper" und "Astronom" als durch die Relation "entdecken" miteinander verbunden sehen. Um die konzeptuelle Relation "entdecken" von dem zweistelligen Prädikat "entdecken" zu unterscheiden, fügen wir an die konzeptuellen Relationsbezeichner einen Unterstrich ("_") an.

```
entdecken_(astronom_, himmelskoerper_)
```

Ebenso läßt sich "Himmelskörper" als durch die Relation "besitzen" mit der Entität "Durchmesser" verbunden begreifen. Keine Relation würde aber z.B. die beiden Entitäten "Durchmesser" und "Astronom" miteinander in Beziehung setzen, da diese Angabe sich nur auf den Spalteneintrag "Himmelskörper" bezieht. Genausowenig läßt sich die Ordnung der Entitäten einer Relation als beliebig ansehen. Die Umkehrung der obigen Reihenfolge:

```
entdecken_(himmelskoerper_, astronom_)
```

ist inkorrekt weil nicht definiert. Auf diese Weise wird natürlich die grammatikalische Subjekt-Objekt-Unterscheidung für den konzeptuellen Bereich simuliert. Das folgende Schaubild zeigt beispielhaft wie ein konzeptuelles Schema für unser Programm visualisiert werden kann. Die Pfeile zeigen dabei die Richtung der jeweiligen Beziehung zwischen zwei konzeptuellen Typen an. Pfeile, die nicht durch eine konzeptuelle Relation benannt werden wie z.B. die Beziehung zwischen "himmelskoerper_" und "mond_", werden als Beziehung zwischen Ober- und Unterbegriff in unserem Programm über die Struktur eines Terms im weiter unten erklärten Sinn definiert. Herkömmlich wird eine solche Beziehung als "is_a"-Relation bezeichnet.



Die Integration der Beschränkungen, die über die Definition von Sinnrelationen zustande kommen, in unser bisheriges Programm geschieht auf die gleiche Weise wie die Typisierung der Lambdaausdrücke. Das bedeutet, daß wir eine bestimmte Beschränkung wie "astronom_" so wie das Typmerkmal "(e,t)" als zusätzlichen Parameter definieren. Anhand des vorherigen Beispiels läßt sich zeigen, wie das semantische Lexikon entsprechend anzureichern ist:

```
praedikate(astronom, X^astronom(X),astronom_).
praedikate(entdeckte, X^Y^entdecken(X,Y),astronom_,himmelskoerper_(_)).
quantoren(_,indef, X^N^VP^existiert(X,N&VP)).
quantoren(_,all, X^N^VP^fuer_alle(X,N=>VP)).
```

Außerdem erweitern wir den zweiten komplexen Parameter von "syn_sem" um eine weitere Stelle, die die Information des konzeptuellen Typs repräsentiert. Die beiden Definitionen der Artikel- und der Verbsemantik dienen als Beispiel dafür:

```
syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
        Syntax = det(dets([Name,Typ,_])),
        Logik = (X^N)^((X^VP)^ Det),
        LogikTyp =  ((e,t) ,((e,t),t)),
        ConceptTyp = (C/C) /((C/C)/C),
        quantoren(Name,Typ, X^N^VP^Det).
syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
        Syntax = v([Wort,_,[[nom|_],[_|_]]]),
        Logik = ((Y^V)^NP)^(X^NP),
        LogikTyp = (((e,t),t) , (e,t)),
        ConceptTyp = ((O/O)/O) / (S/S),
        praedikate(Wort, X^Y^V,S,O).
```

Damit die konzeptuellen Typen in der gleichen Weise bei der Betareduktion verwendet werden können wie die logischen Typen, übernehmen wir deren Struktur und füllen sie mit der konzeptuellen Information. Somit muß nur noch die Relation "beta_reduktion" um einen weiteren Parameter ergänzt werden:

```
beta_reduktion( [P,E,A], [P^Q,(E,T),(A/B)],  [Q,T,B]).
beta_reduktion( [P^Q,(E,T),(A/B)],  [P,E,A], [Q,T,B]).
```

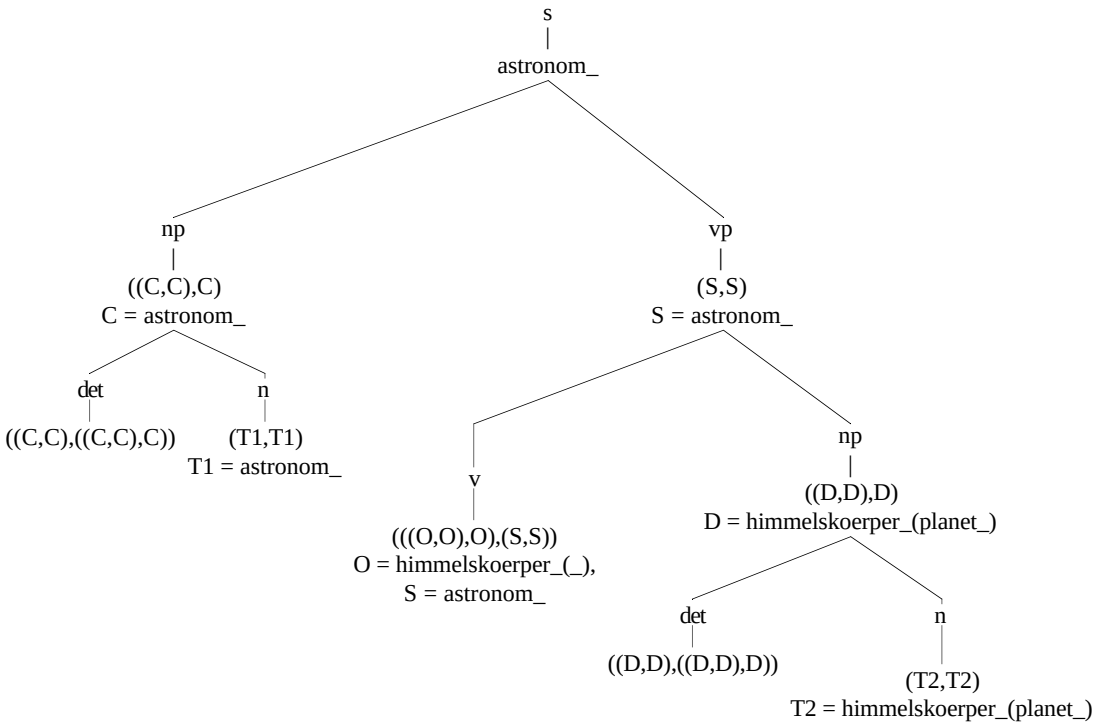
Die konzeptuellen Restriktionen werden somit bei jeder Betareduktion überprüft. Bestimmte Konzepte wie Himmelskörper sind Oberbegriffe, die durch eine Menge von hyponymen Elementen definiert sind, wie "Planet" oder "Mond". Die Benutzung solcher Oberbegriffe ist auch auf konzeptueller Ebene sinnvoll. Will man z.B. sagen, daß Astronomen sowohl Planeten als auch Monde entdecken, dann läßt sich dies in Prolog effektiv so darstellen:

```
entdecken_(astronom_,  himmelskoerper_(_))
```

Die Definition der Entitäten muß dann nur noch das entsprechende Format erhalten:

```
himmelskoerper_(planet_))
himmelskoerper_(mond_))
```

Nun kann bei der Betareduktion entdecken_ mit beiden Konzepten unifizieren. Der folgende Ableitungsbaum zeigt wiederum an unserem Beispielssatz "jeder astronom entdeckte einen planeten" wie die konzeptuelle Typinformation zu einem konzeptuell wohlgeformten Ausdruck kombiniert wird. Um den konzeptuellen Bereich zu fokussieren sind alle anderen syntaktischen und semantischen Information eines kompletten Übersetzungsvorganges ausgespart worden.



Die Angabe von semantischen Merkmalen kann man als Klassifizierung des Modells betrachten. Ihr Sinn besteht darin Sätze von denen klar ist, daß sie keine Interpretation im Modell haben werden, schon vor ihrer Evaluierung scheitern zu lassen. Daher könnte man semantische Merkmale auch als Resultat einer Präevaluierung des Modells ansehen. Da das Modell alle sinnlosen Sätze letztlich aber sowieso falsifiziert, ist die Verwendung semantischer Merkmale in Systemen mit Modellauswertung in gewisser Weise redundant.

Skopusambiguitäten

Auf den letzten Seiten wurde ein Verfahren definiert, das aus syntaktischen Strukturen prädikatenlogische Formeln aufbaut. Diese Formeln sollen die Bedeutung eines natürlich-sprachlichen Satzes erfassen, um sie (im Modell) berechenbar zu machen. Die Tatsache, daß das Verfahren erfolgreich eine Formel als Zuordnung findet garantiert aber nicht, daß diese Formel auch tatsächlich die Bedeutung des Satzes vollständig und korrekt erfaßt. So erhält z.B. der Satz

einen himmelskoerper hat jeder astronom entdeckt

in unserem System die folgende Formel als Repräsentation

```
existiert(X,
          himmelskoerper(X)
          &
          fuer_alle(Y,
                    astronom(Y)
                    =>
                    entdecken(Y, X)))
```

Aber ist dies auch tatsächlich die korrekte Übersetzung des Satzes ? Wenn wir den Satz in unserem Modell auswerten, sehen wir, daß er mit falsch bewertet wird. Wir wissen aber, daß nur Astronomen im Modell vorhanden sind, die auch einen Himmelskörper entdeckt haben, so daß die Auswertung eigentlich den Satz wahr machen müßte. Eine andere Erklärung besteht darin, daß die Formel unserer Intuition bzgl. der Bedeutung des Satzes nicht entspricht. Sehen wir uns die Formel genau an: Zunächst wird gefordert, daß ein X gefunden werden kann, das vom Typ Himmelskörper ist und für das gilt, daß alle Astronomen es entdeckt haben, Das heißt also, alle Astronomen haben ein und denselben Himmelskörper entdeckt. Dies war aber nicht die oder zumindest nicht die einzige Lesart des Satzes. Intuitiv sollte der Satz bedeuten, daß alle Astronomen mindestens einen Himmelskörper entdeckt haben, und daß es sich dabei um verschiedene Objekte handeln kann. Diese letztere Lesart wird aber von einer anderen Formel wiedergegeben:

```
fuer_alle(Y,
          astronom(Y)
          =>
          existiert(X,
                    himmelskoerper(X)
                    &
                    entdecken(Y, X)))
```

Diese Formel unterscheidet sich von der ersten durch die Reihenfolge der Quantoren. Im ersten Beispiel befand sich der Allquantor im Skopus des Existenzquantors, man spricht dann vom weiten Skopus des Existenzquantors. Diese Lesart wird auch starke Lesart genannt. Bei der schwachen Lesart ist der Existenzquantor im Skopus des Allquantors und hat selbst nur engen Skopus. Tatsächlich handelt es sich bei den beiden Formeln um mögliche Bedeutungen des natürlichsprachlichen Satzes, wenn auch die letztere Formel in unserem Modell die viel wahrscheinlichere ist. Andere Modelle können jedoch auch oder nur die erste Formel verifizieren. Wenn man die Modellabhängigkeit außer acht läßt, muß man also im allgemeinen davon ausgehen, daß manche Sätze semantisch ambig sind, was die Reihenfolge der Verschachtelung ihrer Quantoren angeht. Ein natürlichsprachliches System sollte in der Lage sein, diese Ambiguitäten auch zu erzeugen. Das heißt also, es muß ein Algorithmus gefunden werden, der die Reihenfolge der Quantoren unabhängig von ihrer Reihenfolge im Satz permutiert und so die verschiedenen Lesarten erzeugt. Ein Beispiel für einen derartigen Algorithmus stammt von Pereira/Shieber (PEREIRA/SHIEBER 1987), die auch eine Prologimplementierung vorstellen. Diese sieht eine Nachbehandlung einer Formel vor, derart, daß diese zerlegt und in einer neuen Quantorenreihenfolge zusammengesetzt wird. Das ist natürlich ein relativ aufwendiges Verfahren - schöner wäre eine Lösung, die schon bei der Konstruktion der Formel aus der Syntax alternative Reihenfolgen berücksichtigt. Prinzipiell gibt es aber eine Menge von technischen Lösungsmöglichkeiten für dieses Problem. Allerdings ist mit so einer Lösung noch nicht die Frage beantwortet, wie die verschiedenen Lesarten dem Benutzer eines natürlichsprachlichen Systems verständlich gemacht werden sollen. Wir stellen hier ein Programm vor, das die verschiedenen Lesarten erzeugt, indem es den syntaktischen Baum, der Input für die Übersetzung in die Semantik ist, modifiziert. Dabei werden die Positionen von Nominalphrasen, die Quantoren enthalten, im Baum permutiert. Dieses Programm binden wir in unser System ein, indem wir es der Übersetzungsrelation "syn_sem", die aus syntaktischen Bäumen prädikatenlogische Formeln erzeugt, vorschalten:

```
semantische_analyse(SyntaktischerBaum, Praedikatenlogik):-
        skopus_analyse(SyntaktischerBaum, SyntaktischerBaum2),
        syn_sem(SyntaktischerBaum, Praedikatenlogik).
```

Die Erzeugung verschiedener Lesarten bzw. Quantorenstellungen wird vom folgenden Programm erreicht, indem eine Hilfsliste verwendet wird, die die quantifizierten Nominalphrasen eines Baumes aufnimmt und sie dann zu neuen Bäumen zusammensetzt:

```
skopus_analyse(SynBaum,SynBaumi):-
        scope(SynBaum,SynBaumi,Vars,[],Nps,[]),
        einsetzen(Nps,Vars).

scope(B,Bi,V1,V3,F1,F3):-
       dif(M,np),
       B =.. [M,L,R],
       Bi =.. [M,L1,R1],
       scope(L,L1,V1,V2,F1,F2),
       scope(R,R1,V2,V3,F2,F3).

scope(B,Bi,V1,V2,F1,F2):-
       B =.. [M,L],
       Bi =.. [M,L1],
       scope(L,L1,V1,V2,F1,F2).

scope(np(L,R),X,[X|V],V,[np(L,R)|F],F).
scope(B,B,V,V,F,F):-
       B =.. [M,[_|_]]; B =.. [M,[]].

einsetzen([],[]).
einsetzen([X|Y],Vars):-
       delete(X,Vars,RestVars),
        einsetzen(Y,RestVars).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|Z1]):- dif(X,Y), delete(X,Z,Z1).
```

Die zentrale Prozedur "scope" startet mit dem syntaktischen Baum "SynBaum" und berechnet eine Kopie "SynBaumi" dieses Baumes als Resultat. Die Kopie enthält anstelle der quantifizierten Nominalphrasen Variablen, die in der Liste "Vars" aufgesammelt werden. In der Liste "Nps" erhält man die aus dem Eingabebaum extrahierten Nominalphrasen. Somit besteht die Aufgabe der Prozedur "einsetzen" nur noch im (nicht-deterministischen) Zuweisen von Nominalphrasenen zu Variablen. Das Ergebnis von "skopus_analyse" sind Baumkopien, die jeweils andere Füllungen der Nominalphrasenpositionen enthalten.

Die Anwendung dieses Programms auf den Beispielsatz

einen himmelskoerper hat jeder astronom entdeckt

ergibt die zweite Lesart. Für andere Sätze unserer Grammatik ist dieses Programm jedoch nicht korrekt. So werden z.B. bei Quantoren innerhalb von Präpositionalphrasen Ausdrücke erzeugt, die ungebundene Variablen enthalten, also Formeln, die nicht geschlossen sind. Um dies zu vermeiden, kann man entweder das schon vorgestellte Programm zur Überprüfung geschlossener Formeln als Filter nachschalten:

```
semantische_analyse(SyntaktischerBaum, Praedikatenlogik):-
        skopus_analyse(SyntaktischerBaum, SyntaktischerBaum2),
        syn_sem(SyntaktischerBaum2, Praedikatenlogik).
        geschlossene_formel(Praedikatenlogik).
```

Dann wird die Weitergabe inkorrekter prädikatenlogischer Formeln an die Auswertung verhindert, weil diese Ausdrücke diesen Test nicht bestehen. Die effizientere Lösung besteht in der Integration des Tests in das Modul, das die verschiedenen Lesarten erzeugt, in "skopus_analyse". Ein Implementierung, die nach diesem Konzept vorgeht, findet man bei (HOBBS/SHIEBER 1987). Außerdem behandelt unser Skopusalgorithmus nur Nominalphrasen, die ihrerseits nicht in eine Nominalphrase eingebettet sind. Eine Erweiterung des Programms, die auch diese Ausdrücke erfaßt, ist jedoch mit dem vorgestellten syntaktischen Verfahren nicht realisierbar (warum ?).

Übungsaufgabe 17:
Die Skopuskomponente erzeugt die verschiedenen Quantorenabfolgen, ohne den Typ der Quantoren zu beachten. Sinnvolle Lesartenunterschiede ergeben sich jedoch nur, wenn Quantoren ihre Position gegenüber Quantoren anderen Typs verändern. So ergibt sich aus der Umkehrung der Abfolge Allquantor-Existenzquantor die starke Lesart aus der schwachen. Vertauscht man jedoch zwei Existenz- oder zwei Allquantoren, so ergibt sich kein Unterschied in der Interpretation. Modifizieren Sie daher die Skopuskomponente derart, daß sie nur Lesartenunterschiede für Quantorenabfolgen erzeugt, die aus Quantoren verschiedenen Typs bestehen.

Pragmatik

Einer verbreiteten Einschätzung zufolge ist die Pragmatik das Auffangbecken aller von der Syntax und der Semantik nicht gelösten Probleme. Noch weiter geht die Extremposition (KALISH 1967), daß es Semantik nicht gibt, sondern nur eine syntaktische und eine pragmatische Ebene. Hier gehen wir von der eher traditionellen Sicht aus, daß Pragmatik die Sprache mit ihren abstrakten syntakto-semantischen Ebenen in einen konkreten funktionellen (aber nicht wahrheitsfunktionalen) Zusammenhang rückt. Auf pragmatischer Ebene fragen wir also nicht nach der syntaktischen Korrektheit oder der Bedeutung eines Satzes, sondern nach seiner Funktion in einem kommunikativen Kontext. So haben wir bisher die folgenden Sätze auf semantischer Ebene nicht unterschieden:

herschel entdeckte einen planeten
hat herschel einen planeten entdeckt
welchen planeten hat herschel entdeckt

Der prädikatenlogische Gehalt, der sich aus den drei Sätzen ergibt, ist und sollte auch nur ein einziger sein, da wir in allen Sätzen die gleichen Objekte bzw. Relationen beschreiben. Aus pragmatischer Sicht gesehen dienen diese Sätze aber natürlich drei verschiedenen kommunikativen Zwecken. So könnte man im Kontext einer natürlichsprachlichen Datenbankschnittstelle folgende Funktionen der Sätze definieren:

- *Hinzufügung einer neuen Information in die Datenbank*
- *Abfrage des Zutreffens der gesamten Information*
- *Retrieval eines bestimmten Informationsteils*

Die Unterscheidung dieser verschiedenen kommunikativen Absichten erlaubt natürlich eine spezifischeren Umgang mit dem Kommunikationspartner (hier der Datenbank). Zu diesem Zweck muß die prädikatenlogische Formel, die diesen Sätzen entspricht, jeweils so modifiziert werden, daß sie die pragmatische Funktion eines Satzes kodiert. So könnte man die pragmatische Funktion in die Kodierung der prädikatenlogischen Repräsentation integrieren, indem man der Formel

existiert(X, planet(X) & entdecken(herschel, X))

verschiedene Operatoren mitgibt:

aussage(existiert(X, planet(X) & entdecken(herschel, X)))
ja_nein_frage(existiert(X, planet(X) & entdecken(herschel, X))))
wert_frage(existiert(X, planet(X) & entdecken(herschel, X))))

Mit dieser unterschiedlichen Typisierung kann man auf der Ebene der Evaluierung einer Formel entsprechende Reaktionen des Systems vorsehen. Wie die Typisierungen zustande kommen, kann uns nur die Syntax sagen, da es auf ihrer Ebene entschieden wird, ob ein Satz ein Aussage- oder ein Fragesatz ist. Dazu müssen wir die ursprüngliche Konzeption

```
grammatikalische_analyse( Eingabe, Repräsentation):-
        morphologische_analyse(Grundformen, Vollformen),
        lexikalische_analyse(Eingabe, Getaggte_Eingabe),
        syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
        semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
        pragmatische_analyse(SemantischeRepräsentation, Repräsentation).
```

so abändern, daß das pragmatische Modul auch Informationen über den syntaktischen Status eines Satzes erhält:

```
grammatikalische_analyse( Eingabe, Repräsentation):-
        morphologische_analyse(Grundformen, Vollformen),
        lexikalische_analyse(Eingabe, Getaggte_Eingabe),
        syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
        semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
        pragmatische_analyse(SyntaktischerBaum,SemantischeRepräsentation, Repräsentation).
```

Die Klassifizierung des jeweiligen Typs eines Satzes ist nun leicht vozunehmen, da eine Ja-Nein-Frage immer die Struktur "s(_, s(_,_))" besitzt, wobei die erste anonyme Variable ein beliebiges Verb oder Hilfsverb beschreibt. Aussagesätze und Fragen nach Werten dagegen haben immer die Baumrepräsentation mit einer rechten Vp-Tochter "s(_,vp(_,_))". Diese Struktur muß nun noch differenziert werden, nach Wert-Frage oder Aussagesatz. Dafür müssen wir die linke Np-Tochter bis in den Artikel verfolgen um festzustellen, ob ein Frageartikel wie "welcher" vorlag: s(np(det(dets([_,frage|_])),_), _). Mit diesen Vorgaben können wir die pragmatische Analyse leicht definieren:

```
pragmatische_analyse(s( _, s(_,_)), Repräsentation, ja_nein_frage(Repräsentation)).

pragmatische_analyse(SynBaum ,Repräsentation, wert_frage(Repräsentation)):-
        SynBaum = s(np(det(dets([_,frage|_])),_), _) , !.

pragmatische_analyse(s(_,vp(_,_),),Repräsentation, aussage(Repräsentation)).
```

Eine weitere sinnvolle Unterscheidung ist die zwischen Wert-Fragen nach einem Wert und die nach mehreren Werten. Dafür reicht es aus den Numerus des Frageartikels zu bestimmen und danach die Frage zu typisieren:

```
pragmatische_analyse(SynBaum ,Repraesentation, wert_frage_ein(Repraesentation)):-
   SynBaum = s(np(det(dets([_,frage,[_,sing,_]])),_), _) , !.
pragmatische_analyse(SynBaum ,Repraesentation, wert_frage_mehr(Repraesentation)):-
   SynBaum = s(np(det(dets([_,frage,[_,plu,_]])),_), _).
```

Natürlich ist mit der Unterscheidung der Satzfunktionen die pragmatische Seite nicht erschöpfend behandelt. So müssen auch Probleme der folgenden Art unter diesen Begriff fallen. Stellt ein Benutzer die Frage

```
welcher astronom hat einen mond entdeckt
```

dann bekommt er als Reaktion nur den Namen des Astronomen nicht aber den des Mondes. Ebenso wird in der Frage

```
wieviele astronomen haben einen mond entdeckt
```

evtl. mehr als nur die reine Zahlenangabe erwartet. Hier könnte aus pragmatischen Erwägungen heraus auch die Angabe der Namen der Astronomen (oder sogar der Monde) erwartet werden. Dies könnte man z.B. so angehen, daß man Formeln, die als "wert_frage" typisiert wurden, prinzipiell so interpretiert, daß man alle Spalten ihrer Hauptrelation (hier: entdecken_) mit ausgibt. Die genaue Festlegung der Aufgaben einer pragmatischen Komponente ist jedoch außerordentlich schwierig und oft anwendungsabhängig (Stichwort Benutzermodellierung). Die vorliegenden Definitionen sind daher sehr oberflächlich und beschreiben nur einen Ausschnitt der Möglichkeiten. So könnte man eine Komponente, die dem Benutzer klar macht, daß bestimmte Eingaben mehrere Bedeutungen haben können und wie die Antworten darauf zu verstehen sind, auch als Teil der pragmatischen Ebene betrachten.

Evaluierung

Mit der Beschreibung der pragmatischen Komponente ist das Modul grammatikalische_analyse vollständig definiert. Daher kommen wir nun zur Definition von "anwendung", der Komponente also, die das grammatikalische Modul mit der Datenbank verbindet. Die Konzeption, die wir am Anfang vorgestellt hatten, war die folgende:

```
computerlinguistisches_problem(Eingabe, Ausgabe):-
        grammatikalische_analyse( Eingabe, LogischeRepräsentation),
        anwendung( LogischeRepräsentation, Ausgabe).
```

Da die hier interessierende Anwendung die Datenbankabfrage ist, definieren wir das Prädikat "anwendung" entsprechend als Evaluierung:

```
anwendung(Eingabe, Ausgabe):-
        evaluierung( Eingabe, Ausgabe).
```

Somit reduziert sich die Definition von "evaluierung" auf die Anbindung des Ausgabeparameters der pragmatischen Komponente mit dem Modul, das wir vorher unter dem Titel "Semantik der Prädikatenlogik" vorgestellt hatten, also mit dem Programm "wahr". Den Aufruf von "wahr" parametrisieren wir nun in Abhängigkeit der Satzart der Eingabe. Handelt es sich dabei um eine Ja-Nein-Frage, soll "wahr", wenn es erfolgreich ist, "ja" zurückgeben werden und im Falle des Scheiterns "nein".

```
evaluierung( ja_nein_frage(Logik), ja ):-
    wahr( Logik), ! .
evaluierung( ja_nein_frage(_), nein ).
```

Handelt es sich bei der Eingabe um eine Wertfrage, die einen Wert erwartet, dann schreiben wir diesen Wert heraus:

```
evaluierung( wert_frage_ein(Logik), Wert ):-
    Logik =.. [_, Wert, _],
    wahr( Logik) ,
    !,
    schreibe(Wert).
evaluierung( wert_frage_ein(_), keine_werte_berechnet ).
```

Die Antworten auf Wertfragen nach mehreren Werten erzeugen wir, indem wir "findall" aufrufen und per "sort" mehrfache Vorkommen von Werten eliminieren:

```
evaluierung( wert_frage_mehr(Logik), Wert ):-
      Logik =.. [_, Wert, _],
       findall(Wert,wahr( Logik),Werte),
      sort(Werte,W),
      schreiben(W),
     !.
evaluierung( wert_frage_mehr(_), keine_werte_berechnet ).
```

Schließlich definieren wir noch einen Aussagesatz als Anweisung, eine Relation in die Datenbank einzufügen:

```
evaluierung( aussage(Logik), relation_abgespeichert(Logik) ):-
   variablenfrei(Logik),
   assert(Logik), ! .
evaluierung( aussage(_), relation_nicht_gespeichert ).
```

Da wir hiervon so wenig informative Sätze wie

```
ein astronom entdeckte einen planeten
```

ausnehmen wollen, verlangen wir, daß es sich dabei nur um Sätze handeln darf, die feste Werte angeben, die m.a.W. also quantorenfrei und damit auch variablenfrei sind. Zusätzlich prüfen wir durch einen Test, ob die Datenbank einen derartigen Eintrag schon enthält, was natürlich nicht der Fall sein soll:

```
variablenfrei( Logik ) :-
   Logik =..  [ Relation | T],
   not( quantor( Relation ) ),
   not( junktor1( Relation ) ),
   not( junktor2( Relation ) ),
   not((member(X,T),  var(X))),
   not(call(Logik)).
```

Natürlich genügen diese Festlegungen nicht, um eine Aussage bezüglich der Konsistenz der Datenbankeinträge treffen zu können. So kann man den Satz "herschel entdeckte ariel" einfügen, auch wenn ein dem widersprechender Eintrag wie z.B. "tombaugh entdeckte ariel" bereits in der Datenbank vorhanden ist. Ein korrekter Konsistenzchecker müßte dafür überprüfen, ob die Restriktion erfüllt bleibt, daß ein Himmelskörper nur von einem einzigen Astronomen entdeckt wurde, wenn eine neue Relation in die Datenbank aufgenommen wird. Derartige Überprüfungen sind leicht zu formulieren, daher ersparen wir uns ihre Definition an dieser Stelle.

Diese Erläuterungen beschliessen das Thema Evaluierung. Ein letzter Punkt betrifft deren Effizienz. Mit der Definition der Auswertung als Beweis einer prädikatenlogischen Formel in einem Modell haben wir eine korrekte Semantik für die Prädikatenlogik erhalten, die jedoch durch die Art der Interpretation sehr rechenzeitaufwendig ist. So wird bei einer Existenzaussage zunächst ein beliebiges Individuum aus dem Bereich eingesetzt, um dann die Formel mit dieser Belegung zu beweisen. Dies entspricht der "generate and test" Vorgehensweise, die in Prolog immer ineffizienter als die umgekehrte Strategie ist, zuerst einen Test auszuführen, um dann die Belegung zu überprüfen. Geht man den Weg der direkten Auswertung der Formel, dann kann man sich deren Abarbeitung innerhalb von "wahr" ersparen, da die Junktoren in Prolog direkt implementierbar sind:

```
wahr(Formel):- call(Formel).

P &Q :- call((P,Q)).
P v Q :- call(P) ; call(Q).
P => Q :- call(¬(P & ¬Q)).
¬ P :- not(P).
```

Auf diese Art und Weise übergibt man die Formeln dem Prolog-Compiler direkt zur Auswertung. Die Korrektheit der Interpretation der Quantoren ist gegeben, da eine existenzquantifizierte Aussage korrekt ist, wenn man die Formel zumindest einmal beweisen kann.

```
existiert(_,Formel):- call(Formel).
```

Etwas trickreicher verhält es sich bei Allaussagen, da man hier sicherstellen muß, daß wirklich alle betroffenen Relationen überprüft werden. Doch kann man auch hier deren Korrektheitsüberprüfung dem Implikationsteil der Formel überlassen und den quantifizierenden Teil leerlaufen lassen, wie schon beim Existenzquantor:

```
fuer_alle(_,Formel):- call(Formel).
```

Diese Interpretationen sind jedoch nur für generalisierte Quantoren korrekt, da dort eine Unterscheidung zwischen Restriktion und Skopus syntaktisch getroffen wird. Allgemeine Allaussagen wie z.B.

```
fuer_alle(X, astronom(X) )
```

sind offensichtlich damit nicht korrekt behandelbar. Bei der geraden geführten Diskussion sollte man beachten, daß beide Definitionen von "wahr" eine korrekte Semantik für unsere prädikatenlogischen Formeln ergeben und daß die Frage, welche man auswählt, sich nur aus Effizienzgesichtspunkten stellt. Die allgemeinere und langsamere Variante ist die erstere Definition, während die zweite schneller aber auch spezieller ist und nur für die Strukturen unserer Formeln verwendbar ist.

Auch mit der zweiten Definition von "wahr" lassen sich jedoch nicht für alle Anfragen akzeptable Reaktionszeiten erzielen. Dies liegt an der Art und Weise wie die Formeln aufgebaut sind, die Sätze repräsentieren. Nehmen wir als Beispiel den folgenden Satz:

```
welche monde besitzen einen durchmesser der groesser ist
   als der durchmesser von einem planeten
```

Ihm wird als semantische Repräsentation die prädikatenlogische Formel zugeordnet:

```
existiert(_5650,
      mond(_5650)
      &
       existiert(_6249,
            durchmesser(_6249)
            &
              existiert(_7271,
                   durchmesser(_7271)
                   &
                     existiert(_7933,
                          planet(_7933)
                          &
                            besitzen(_7933,_7271))
                  &
                    groesser(_6249,_7271))
           &
             besitzen(_5650,_6249)))
```

Die Auswertung dieser Formel führt zu einer kombinatorischen Explosion, da viele Werte zu oft berechnet werden. So wird bei einer Auswertung der Formel von Links nach Rechts zuerst das Prädikat "mond(X)" berechnet, indem der Parameter X zu einem beliebigen Wert innerhalb der Domäne von Monden instantiert wird. Wenn ganz am Ende der Formel das Prädikat "besitzen(X,Y)" ausgewertet wird, das den Parameter X mit dem Prädikat "mond" teilt und scheitert, dann führt backtracking zu einer vollständigen Neuberechnung der gesamten Formel anfangend von einer Neubelegung des ersten Prädikats. Die Funktion des einstelligen Prädikats besteht dabei im Grunde nur aus der Bereichsfestlegung für X, also für die Fixierung des Wertes für X auf die Klasse von Monden. Wenn es gelänge, diese Festlegung lokal an die Variable innerhalb der Relation "besitzen" zu binden, dann ließe sich das Prädikat "mond" aus der Formel eliminieren, ohne deren Bedeutung zu verändern. Die folgende Formel erfüllt z.B. diese Funktion durch eine geeignete Typfestlegung der Argumentvariablen des Prädikats "besitzen":

```
besitzen((_7933,planet_),(_7271,durchmesser_))
                &
                  groesser(_6249,_7271)
          &
            besitzen((_5650,planet_),(_6249,durchmesser_))
```

Natürlich setzen derartig modifizierte Formeln auch entsprechend veränderte Interpretationsalgorithmen für die Evaluierung voraus. Das entsprechende Evaluierungsprädikat lautet:

```
besitzen((Himmelskoerper,planet_),(Durchmesser,durchmesser_)):-
      db(Himmelskoerper,planet,E,_,_),
   nonvar(E),
   E = Durchmesser.
```

Für die Implementierung eines solchen Ansatzes bedarf es also lediglich der Abänderung der Prädikats- (Modul Semantik2) und Interpretationsdefinitionen (Modul Evaluierung-Variante3).
Diese Feststellung kann man auf alle einstelligen Prädikate (im Beispiel fett) unseres Systems ausweiten, da sie alle nur die Funktion der Bereichseinschränkung haben (abgesehen von ihrem Vorkommen mit dem Prädikat es_gibt, auf das wir noch eingehen). Somit könnte man in der Beispielsformel 4 von 7 Prädikaten löschen, was die kombinatorischen Probleme bei der Auswertung verringern würde. Natürlich lassen sich mit dieser Methode nicht für alle Formeln Verkürzungen erreichen. In Formeln, die eine Bereichsaufzählung für die korrekte Auswertung erfordern, ist dies nicht (so einfach) möglich. Dazu zählen in unserem System vor allem Allaussagen wie die folgende:

```
entdeckte jeder astronom einen planeten ?
```

Die diesem Satz zugeordnete Formel

```
fuer_alle(_2410,
        astronom(_2410)
       =>
        existiert(_2960,
              planet(_2960)
             &
              entdecken(_2410,
                   _2960)))
```

wird wie vorher beschrieben schließlich wie folgt als

```
¬ ( astronom(_2410) & ¬ ( planet(_2960) & entdecken(_2410,_2960)))
```

an Prolog übergeben.

Bei einer Eliminierung der einstelligen Prädikate bleibt nur das doppelt negierte zweistellige Prädikat "entdecken(_2410,_2960)" übrig. Dessen Auswertung führt natürlich zu einem inkorrekten Ergebnis, da die Semantik des Ausgangssatzes nicht erhalten geblieben ist.

Integration versus Koroutinierung

Die Definition der Programme ist möglichst stringent in Abhängigkeit der verschiedenen unterscheidbaren Aufgaben erfolgt. So gibt es z.B. ein syntaktisches Modul und ein semantisches Modul, die vollkommen autonom definiert wurden und nur über eine Schnittstelle (syntaktischer Baum) miteinander in Verbindung stehen. Dies hat die üblichen Vorteile modularer Programmierung wie bessere Handhabbarkeit der einzelnen Module, Wiederverwertbarkeit einzelner Module, größere Übersichtlichkeit etc. Der Preis für diese Vorgehensweise besteht in einer geringeren Effizienz. So muß ein Satz, der syntaktisch korrekt ist, wie

ein planet entdeckte einen astronomen

erst noch in das semantische Modul übernommen werden, um schließlich zurückgewiesen zu werden, weil semantische Bedingungen nicht erfüllt werden konnten. Bei einer Parsingstrategie aber, die von links nach rechts vorgeht, könnte man sich die Bearbeitung der Objektsnominalphrase "einen astronomen" eigentlich sparen, wenn die Unvereinbarkeit von Verb und Subjektsnominalphrase bekannt ist. Das heißt, eine Verzahnung von Aufgaben dieser beiden Module wäre für eine möglichst schnelle Reaktion von Vorteil. Eine offensichtliche Lösung für dieses Problem besteht darin, nicht unterschiedliche Module zu entwerfen, sondern ein einziges Programm zu konstruieren, das alle Aufgaben quasi gleichzeitig angeht. Diese integrierte Lösung ist auch die Standardlösung für die meisten Sprachverarbeitungsprogramme in Prolog. Ein typisches Beispiel für die Vermischung der Aufgaben des Parsing, der Syntaxanalyse und der Konstruktion einer semantischen Repräsentation ist eine DCG wie die folgende:

```
s(Q) -->    np(P^Q), vp(P).
np(Q) --> det(P^Q), n(P).
vp(Q) -->  v(P^Q), np(P).
v( ((X^entdecken(Y,X))^NP)^(Y^NP) ) -->          [entdeckt].
det( ((X^N)^((X^VP)^existiert(X,N&VP)))) -->     [eine].
det( ((X^N)^((X^VP)^fuer_alle(X,N=>VP)))) -->    [jede].
n(X^astronomin(X)) -->                           [astronomin].
```

Syntax und Semantik gehen hier Hand in Hand. So wird die Betakonversion der Lambdaausdrücke in diesem Ansatz direkt erreicht, indem man mithilfe der Unifikation die jeweiligen Funktor- und Argumentausdrücke ineinander setzt. Dies hat Effizienzvorteile, da zum Teil bestimmte Strukturen schon partiell vorbelegt werden.

Möchte man den Modulansatz jedoch nicht aufgeben, so hat man (mindestens) eine weitere Option. Man koroutiniert die verschiedenen Module miteinander. Koroutinierung heißt,

daß Routinen, also Programme in Abhängigkeit bestimmter Programmkontexte aufgerufen werden. Für unsere Grammatik läßt sich die Koroutinierungsaufgabe so formulieren:

> *Das semantische Modul soll in Abhängigkeit von dem Bekanntwerden des Standes des syntaktischen Verarbeitung aktiv werden.*

Das heißt, daß die Prozedur "semantische_analyse" die Baumbearbeitung von z.B. der SubjektNp im obigen Beispiel nach ihrer syntaktischen Behandlung abgeschlossen haben soll. Der semantische und der syntaktische Teil der Bearbeitung sollen dabei Hand in Hand gehen, d.h. sobald ein neuer Teil des syntaktischen Baumes von der Syntaxanalyse konstruiert werden kann, muß die semantische Analyse diesen Teil weiterverarbeiten, um dann wieder anzuhalten, um auf neue syntaktische Informationen zu warten. Ein ideales Instrument für die Implementierung einer derartigen verzahnten Arbeitsweise bieten Prologversionen vom Typ II (z.B. Sicstus Prolog, Prolog II). Sie stellen ein Prädikat "freeze" zur Verfügung, daß den Aufruf einer Prozedur, nennen wir sie "p" abhängig macht von der Belegung einer (oder mehrerer) Variablen, nennen wir sie "X":

```
freeze( X, p( X ) )
```

Ruft man diese Funktion unter dem Prologprompt auf,

```
?- freeze( X, p( X ) ).
```

so wartet die Prozedur mit ihrem Start darauf, daß X eine (nichtvariable) Belegung bekommt, die wir hier so simulieren wollen:

```
?- freeze( X, p( X ) ), X = k.
```

Die Bearbeitung von "p" wird also erst mit der nachfolgenden Belegung von "k" für X begonnen. Dies ist genau die Eigenschaft, die wir für die Verzahnung von z.B. Syntax und Semantik brauchen. Wir nehmen die ursprüngliche Reihenfolge innerhalb von "grammatikalische_analyse" her

```
grammatikalische_analyse( Eingabe, Repräsentation):-
    morphologische_analyse(Grundformen, Vollformen),
    lexikalische_analyse(Eingabe, Getaggte_Eingabe),
    syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
    semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
    pragmatische_analyse(SemantischeRepräsentation, Repräsentation).
```

setzen die Semantik vor die Syntax

```
grammatikalische_analyse( Eingabe, Repräsentation):-
    morphologische_analyse(Grundformen, Vollformen),
    lexikalische_analyse(Eingabe, Getaggte_Eingabe),
    semantische_analyse(SyntaktischerBaum, SemantischeRepräsentation),
    syntaktische_analyse(Getaggte_Eingabe, SyntaktischerBaum),
    pragmatische_analyse(SemantischeRepräsentation, Repräsentation).
```

und frieren den Aufruf "syn_sem " in "semantische_analyse" ein in Abhängigkeit des Bekanntwerdens des syntaktischen Baumes:

```
semantische_analyse(SyntaktischerBaum, Lesart):-
        freeze(SyntaktischerBaum, syn_sem(SyntaktischerBaum, Praedikatenlogik)).
```

Damit wird der erste Prozeß des "Auftauens" der Prozedur "syn_sem" eingeleitet mit der Unifikation der obersten Satzregel, die den Wert der Variablen "SyntaktischenBaum" auf "s(NP, VP)" setzt. Daß der weitere Fortgang der semantischen Analyse nun wieder gestoppt wird, bis "NP" und "VP" eine Zuweisung erhalten, sehen wir an der rekursiven Definition von "syn_sem":

```
syn_sem(SynBaum, Sem):-
    SynBaum =.. [_, Links, Rechts],
    semantische_analyse(Links, LSem),
    semantische_analyse(Rechts, RSem),
    beta_reduktion(LSem, RSem, Sem).
```

Diese Prozedur ruft nach der Zerlegung eines binären Baumes "semantische_analyse" wieder auf für die Verarbeitung der Töchter, d.h. diese Aufrufe werden nach unserer Definition von "semantische_analyse" wieder eingefroren, bis weitere Belegungen für die Töchterknoten bekannt sind. Die einzige Modifikation, die man an dieser Prozedur vornehmen muß, ist das Einfrieren des Prozeßes der Beta-Reduktion, die erst starten soll, wenn die semantischen Analysen der Töchter bekannt geworden sind:

```
syn_sem(SynBaum, Sem):-
    SynBaum =.. [_, Links, Rechts],
    semantische_analyse(Links, LSem),
    semantische_analyse(Rechts, RSem),
    freeze(LSem, freeze(RSem, beta_reduktion(LSem, RSem, Sem))).
```

Die Koroutinierung von Syntax und Semantik ist damit bereits vollendet und dies bei einer nur minimalen Änderung des Programmtextes! Die Auswirkung dieser Änderung ist natürlich nur in einer schnelleren Reaktion auf inkorrekte semantische Eingaben ablesbar. Die genaue Nachverfolgung der Prozeße des Einfrierens und Auftauens von Prozeduraufrufen ist per Debugger zwar machbar, bei einem so großen Programm aber nur wenig instruktiv. Man macht sich daher die Bedeutung der Prozedur "freeze" besser an einem kleineren Beispiel klar. Das folgende Programm "liste_aus_einsen" berechnet eine im Prinzip beliebig lange Liste aus Einsen. Wie man sieht, führt die alleinige Ausführung dieses Programms im Generierungsmodus zu einer Endlosschleife. Das Programm "laenge" dient der Fixierung der Eingabeliste von liste_aus_einsen auf eine vorgegebene Länge N.

```
laenge(L,N):- freeze(L, laenge0(L,N) ).

laenge0([],0).
laenge0([_|T],N):-
      dif(N,0),
      N0 is N - 1,
       laenge(T,N0).

liste_aus_einsen([1|X]):- liste_aus_einsen(X).
liste_aus_einsen([]).
```

Verbindet man die beiden Programme z.B. durch den folgenden Aufruf

```
?- laenge(L, 5), liste_aus_einsen(L).
```

dann terminiert die Berechnung mit einer Liste aus 5 Einsen. Die Ausführung der beiden Teilprogramme erfolgt dabei so, daß jeweils ein Schritt in jedem berechnet wird und im letzten Schritt "laenge" eine leere Liste an "liste_aus_einsen" übergibt, was zur Terminierung führt.

Selbstverständlich ist die Technik der Koroutinierung im Prinzip auf alle Module unseres Systems übertragbar, vorausgesetzt wird nur eine möglichst homogene Schnittstelle, wie sie unsere Baumstrukturen darstellen. So kann man sich auch die Koroutinierung des Moduls "evaluierung" vorstellen, das beispielsweise die Relation, die einem Verb entspricht, berechnet, noch bevor die syntaktische Analyse den Satzteil nach dem Verb abgeschlossen hat.

Parsing

Wir haben nun mit der Definition einer Grammatik die eine Seite der Sprachverarbeitung kennengelernt. Die Funktion der Grammatik ist dabei die eines Filters. Sie soll nur diejenigen Sätze durchlassen, die man als korrekt ansieht. Das impliziert natürlich, daß auch inkorrekte Sätze vor dem Filter erscheinen. Da das System aber nur die Grammatik als Filter zur Verfügung hat, werden diese Eingabesätze diesem Filter zur Beurteilung übergeben. Der Vorgang, der sich dabei abspielt ist ein eigentlich banaler: Wort für Wort wird der Eingabsatz mit der Grammatik verglichen. Sind alle Wörter des Eingabesatzes in der Anordnung und Zusammenstellung laut Grammatik korrekt, ist die Analyse erfolgreich abgeschlossen und der Satz hat den Filter passiert. Im anderen Fall scheitert die Analyse und der Benutzer muß sein Glück erneut versuchen. Dieser Vorgang des Vergleichens zwischen Benutzereingabe und Grammatikerwartung wird in der Computerlinguistik als Parsing beschrieben. Das Parsing ist deshalb ein eigenständiges Gebiet, weil es auch hier auf eine möglichst effektive Strategie ankommt. So gibt es alle möglichen Arten von Parserarchitekturen: Solche, die den Satz von links nach rechts oder von rechts nach links zerlegen. Andere, die die Grammatik hierfür vom Lexikon zur Syntax aufwärts, bzw. von der Syntax ins Lexikon abwärts durchsuchen. Da es beim Durchsuchen von großen Grammatiken vorkommt, daß viele Schritte sich wiederholen, gibt es auch Parser, die hierfür einen separaten Speicher (Tabelle) anlegen. Schließlich gibt es auch eine Anzahl von Mischtypen unter den Parsern, die mehrere Strategien inkorporieren. Nicht erwähnt haben wir dabei spezielle Probleme, die beim Parsing auftreten können, wie das einer Endlosschleife, die je nach Parser unterschiedlich gelöst werden müssen. Die wesentliche Funktion eines Parsers ist jedoch die des Abgleichens, wie es oben beschrieben wurde.

Wir haben bei der Definition des Parsers bislang den bequemsten Weg gewählt, indem wir keine Fragen bezüglich der besten Vorgehensweise gestellt haben und stattdessen einfach den Prologmechanismus als Parsingmethode übernommen haben. Diese spezielle Methode wollen wir jetzt etwas genauer ansehen und sie dann mit einer alternativen Parsingstrategie kontrastieren.

Die Abarbeitung von Prologregeln erfolgt von oben nach unten. Dieser Strategie wird bei Parsern als Top-Down Methode bezeichnet. Wir wollen dies an einem kleinem Syntaxbeispiel nachvollziehen. Die folgende Minigrammatik in Form einer DCG dient dabei als Modell:

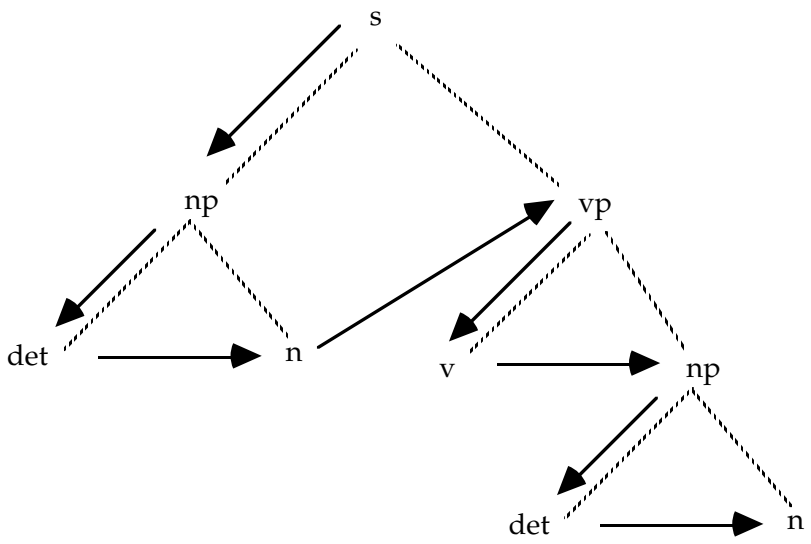
```
s --> np, vp.
np --> det, n.
vp --> v, adj.
det --> [alle].
n --> [astronomen].
v --> [sind ].
adj --> [kurzsichtig].
```

Wir haben die Regeln in einer beliebigen Folge angegeben und demonstrieren ihre Abarbeitungsreihenfolge bei einem **Top-Down-Parser** wie Prolog, indem wir sie nun in der Reihenfolge ihrer Bearbeitung durch Prolog aufführen:

```
s --> np, vp.
np --> det, n.
det --> [alle].
n --> [astronomen].
vp --> v, adj.
v --> [sind ].
adj --> [kurzsichtig].
```

Die folgende Graphik illustriert das Top-Down-Verfahren am Beispiel dieser Grammatik:

Top-Down-Parsing



Diese Reihenfolge kommt durch die rückwärtsverkettende Strategie Prologs zustande, also der Abarbeitung der Literale einer Regel von links nach rechts. Dieser Methode wollen wir als Alternative nun den Bottom-Up-Ansatz gegenüberstellen. Bottom-Up-Parser versuchen die Regeln abzuarbeiten, indem sie den Baum von unten nach oben konstruieren und erst die terminalen Symbole ersetzen, bevor nicht-terminale Strukturen erzeugt werden. Ein solcher **Bottom-Up-Parser** würde die Regeln in der folgenden Reihenfolge (von oben nach unten gelesen) abarbeiten, die Wortkette wird dabei wie beim obigen Top-Down-Parser von links nach rechts durchgegangen:

```
det --> [alle].
n --> [astronomen].
np --> det, n.
v --> [sind ].
adj --> [kurzsichtig].
vp --> v, adj.
s --> np, vp.
```

Der Unterschied von Bottom-Up zu Top-Down-Verfahren besteht also darin, daß Top-Down-Parser mehr Annahmen machen, bevor sie zu den Fakten, die diese Annahmen bestätigen oder negieren können, kommen. So besteht die Annahme in unserem Beispiel darin, daß zuerst die "s" und die "np"-Regel bearbeitet werden, noch bevor das erste Wort - der Artikel - nachgeschaut wird. Im Gegensatz dazu holt sich der Bottom-Up-Parser die ersten zwei Worte und versucht diese zu einer Nominalphrase zusammenzufügen, bevor er weitermacht. Die Konstruktion komplexerer Strukturen gleicht daher eher einem induktiven Verfahren, das so wenig wie möglich noch unbewiesene Annahmen voraussetzt. Ein spezielles Bottom-Up-Verfahren implementieren sogenannte Bottom-Up-Shift-Reduce-Parser. Das folgende Programm ist ein Beispiel für einen derartigen Parser:

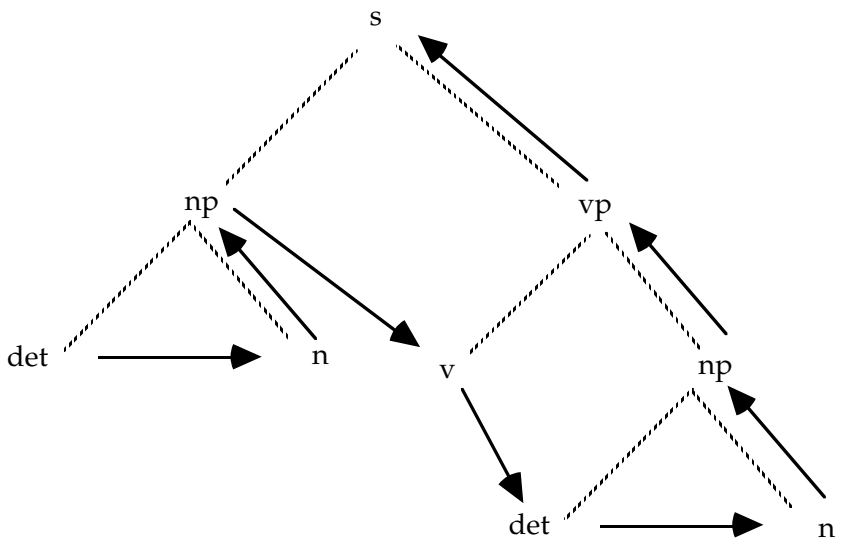
```
start([ Wort | Restworte ], SynBaum) :-
      parse( Restworte, [ Wort ], SynBaum).

parse([], Stapel, S):-
      reduktion( Stapel, [S]).
parse([ Wort | Restworte ], Stapel, SynBaum):-
      reduktion( Stapel, ReduzierterStapel),
      parse( Restworte, [ Wort | ReduzierterStapel ], SynBaum).
```

```
reduktion( X, Z ):-
      reduktion1( X, Y ),
      reduktion( Y, Z ).
reduktion( X, X ).

reduktion1( [Wort|Y], [Baum|Y] ):-
      (Cat ---> [Wort]),
      Cat =.. [C|_],
      Baum =..[C,Wort].
reduktion1( [B,A|Y], [Baum|Y] ):-
      B =.. [B1|_],
      A =.. [A1|_],
      (Cat ---> (A1,B1)),
      Cat =.. [C|_],
      Baum =..[C,A,B].

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

det ---> [eine].
n  ---> [astronomin].
v ---> [entdeckte].
```

Dieser Parser nimmt eine Eingabekette und konstruiert mithilfe eines Stapels einen Syntaxbaum als Ausgabe. Die Vorgehensweise besteht darin, ein Wort der Eingabekette auf den Stapel zu legen (zu shiften) und zu reduzieren, wobei reduzieren bedeutet, dem Wort eine Kategoriebezeichnung zuzuordnen (also dem Wort "eine" die Kategorie "det"). Die Kategoriezuordnung ist rekursiv definiert, so daß einem Wort auch nacheinander komplexere Strukturen zugeordnet werden können, bzw. mehrere Wörter zu einer komplexeren Struktur (wie "det(eine)" und "n(astronomin)" zu "np(det(eine),n(astronomin))") reduziert werden können. Wenn eine Struktur nicht mehr weiter reduziert werden kann, ist sie maximal. Die letzte maximale Struktur ist daher in unserem Beispiel die Satzstruktur unter dem Knoten "s". Zu beachten ist, daß im Programm der Aufruf "reduktion" in der ersten Definition von "parse" notwendig ist, da das Vorhandensein der leeren Eingabekette ([]) nicht impliziert, daß alle Knoten, die sich zu diesem Zeitpunkt auf dem Stapel befinden, schon maximal reduziert sind.

Die folgende Graphik demonstriert die Bottom-Up Strategie am Beispiel eines einfachen Satzbaumes (gestrichelte Linien):

Bottom-Up-Parsing



Der Top-Down-Parser beginnt im Baum mit dem S-Knoten und terminiert mit dem rechten n-Knoten. Dagegen startet der Bottom-Up-Parser mit dem linken det-Knoten und endet mit dem Satzknoten.

Ein Nachteil von Bottom-Up-Parsern liegt in der Tatsache, daß sie leere Ableitungen (wie "a --> []") nicht befriedigend behandeln können (warum ?). Der Vorteil eines solchen Bottom-Up-Verfahrens gegenüber Top-Down-Parsern besteht aber in einem grösseren Determinismus, da Entscheidungen immer so früh wie möglich gefällt werden. Ein weiterer Vorteil besteht in der Tatsache, daß Bottom-Up-Left-Right-Parser linksrekursive Regeln terminierend behandeln können, während Top-Down-Left-Right-Parser dies nicht in jedem Fall vermögen. Sehen wir uns dazu einige Beispiele an.

Wir definieren eine Np mit einer rekursiven Adjektivphrasenregel als DCG:

```
np --> det, ap,n.
ap -->  ap, a.
ap --> [].
det --> [der].
n --> [astronom].
a --> [kurzsichtige].
```

Deklarativ gesehen ist dies eine korrekte Definition der Regeln. Leider führt jedoch der np-Aufruf, egal ob die Eingabe korrekt oder inkorrekt ist, zur Nichtterminierung. Dies liegt

daran, daß beim Aufruf der ap-Regel, der nächste Aufruf auf diese Regel zurückverweist, da Prolog immer die oberste Regel einer Gruppe von Regeln zuerst ausprobiert. Dieses Verhalten läßt sich abändern, indem man die rekursive und die nicht-rekursive Regel vertauscht:

```
np --> det, ap,n.
ap --> [].
ap -->  ap, a.
det --> [der].
n --> [astronom].
a --> [kurzsichtige].
```

Nun funktioniert der Parser bei korrekten Eingabesequenzen wie gewünscht, d.h. er terminiert. Für nichtkorrekte Eingaben allerdings bleibt das Problem der Nichtterminierung bestehen, da Prolog im Backtracking wieder auf die rekursive Regel zurückgreift. Um auch dieses Problem in den Griff zu bekommen, muß man die linksrekursive Struktur der Regel aufgeben und dafür sorgen, daß der Parser nur dann die Rekursion aufnehmen kann, wenn jeweils vor dem nächsten rekursiven Aufruf ein Terminal gefunden wurde:

```
np --> det, ap,n.
ap -->  a, ap.
ap --> [].
det --> [der].
n --> [astronom].
a --> [kurzsichtige].
```

Diese rechtsrekursive Variante der Grammatik erfüllt nun den gewünschten Zweck, d.h. sie terminiert für jede Eingabe. Will man auch für die Generierung Terminierung, d.h. Konstruktion von Ketten haben, muß man zusätzlich noch die rekursive mit der nicht-rekursiven Regel vertauschen.

Das Verfahren, das wir gerade demonstriert haben um die (nicht-terminierende) Linksrekursion zu vermeiden, läßt sich auf das folgende Schema verallgemeinern:

Gegeben seien linksrekursive Regeln des Typs:

nt --> nt, x.
nt --> t.

Dabei stehen nt, t und x für beliebige Kategorien. Zunächst wird eine Hilfskategorie "aux" eingeführt, die in die nichtrekursive Regel eingesetzt wird. Dann definiert man die Hilfskategorie selbst so, daß sie den nichtrekursiven Bestandteil der linksrekursiven Regeln in einer rechtsrekursiven Weise herleitet:

nt --> t, aux.

aux --> [].
aux --> x, aux.

Das Schema angewandt auf die Adjektivregel der Ausgangsbeispielgrammatik

```
ap -->  ap, a.
ap --> [].
```

ergibt:

```
ap --> aux.
aux --> [].
aux --> a, aux.
```

Diese Verfahren der Eliminierung der (nicht-terminierenden) Linksrekursion ist für Top-Down-Left-Right-Verfahren verallgemeinerbar. Bottom-Up-Parser wie der, den wir zuvor definiert haben, benötigen diese Umformung nicht, da sie sowieso nur wortgeleitet vorgehen. Erweitern wir die Grammatik des Bottom-Up-Parsers um einige Regeln, die rekursive Adjektivphrasen zulassen:

```
s ---> np, vp.
np ---> det, n1.
vp ---> v, np.
v ---> [entdeckte].
det ---> [eine].
n ---> [astronomin].
n1 ---> [astronomin].
n1 ---> ap, n.
a ---> [kurzsichtige].
ap ---> ap, a.
ap ---> [kurzsichtige].
```

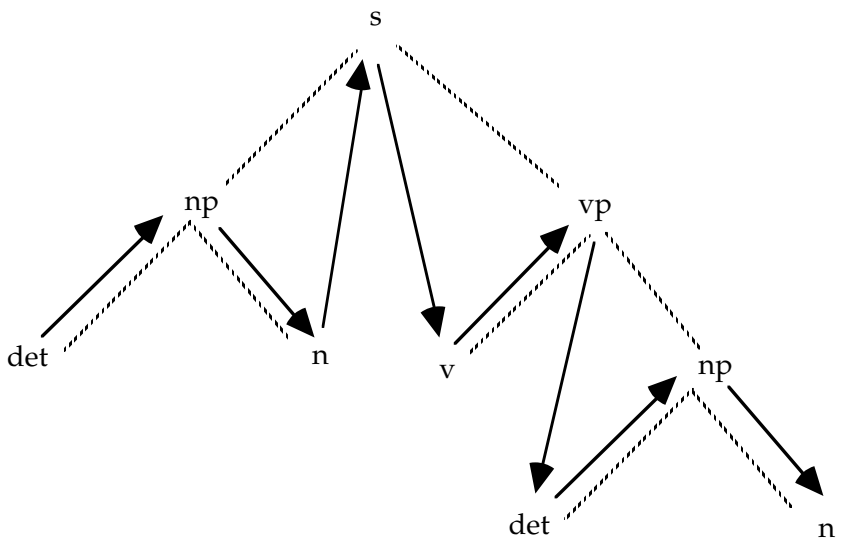
Die vorletzte Regel entspricht der linksrekursiven Variante unseres Top-Down-Parsers. Sie erlaubt im Gegensatz zu dessen linksrekursiver Regel die Kombination einer komplexen Adjektivstruktur zu einer noch komplexeren nur dann, wenn ein Adjektivterminal als nächstes auf dem Stapel liegt. Damit werden nur die rekursiven Strukturen erzeugt, die eine Verkürzung der Wortkette bewirken.

Abschließend betrachten wir eine Parsing-Strategie, die eine Kombination von Bottom-Up und Top-Down darstellt: **Left-Corner-Parsing**. In einem Left-Corner-Ansatz wird zunächst versucht, die linke Ecke (einer Phrasenstrukturregel) zu erkennen. Ist dies gelungen, wird der Rest der Regel als Vorgabe für das weitere Parsing genommen. Die folgende Prozedur implementiert einen solchen Left-Corner-Parser in Prolog:

```
p(S,X,Z):-
      blatt(Cat,X,Y),
      lc(Cat,S,Y,Z).

blatt(Cat, [Wort|C], C):- (Cat ---> [Wort]).

lc(X,X,Y,Y).
lc(A,B,X,Z):-
      (Cat --->(A,C1)),
      rechts(C1,X,Y),
      lc(Cat,B,Y,Z).

rechts((A),X,Z):-  p(A,X,Z).
rechts((A,B), X, Z):-
      p(A,X,Y),
      rechts(B,Y,Z).
```

Dieser Parser leitet z.B. dieselbe Grammatik ab wie der gerade beschriebene Bottom-Up-Shift-Reduce-Parser. Die Hauptprozedur "lc" hat 4 Parameter. Die letzten beiden repräsentieren die Differenzliste, die für das Abarbeiten der Eingabekette sorgt. Der erste Parameter enthält die zuletzt bottom-up gefundene Kategorie, der zweite Parameter die top-down-Vorgabe. Die folgende Graphik zeigt, in welcher Weise der einem Satz entsprechende Baum von dem Left-Corner-Parser aufgebaut wird:

Left-Corner-Parsing



Gehen wir diesen Baum von links nach rechts durch, dann sehen wir, daß zuerst ein "det" gefunden wird, der die linke Ecke der Regel "np ---> det, n." darstellt. Das heißt, damit ist als top-Knoten auch "np" bekannt. Sobald der Rest der Regel, also "n" bekannt ist, kann "np" als nächste linke Ecke einer Regel gesucht werden. Dies trifft auf die Regel "s ---> np, vp." zu. Die nächste Kategorie, ist damit "vp". Deren linke Ecke, ist "v". Der Rest von "vp", also "np", wird wie die erste Np analysiert.

Diese Vorgehensweise des Left-Corner-Parsers läßt sich als reine Bottom-Up-Strategie bezeichnen. Durch die folgende Modifizierung der Regeln "p" und "lc" und der Hinzufügung einer Linktabelle führen wir ein Top-Down-Parsing-Element in den Parser ein:

```
p(S,X,Z):-
      blatt(Cat,X,Y),
      link(Cat, S),
      lc(Cat,S,Y,Z).

lc(A,B,X,Z):-
      (Cat --->(A,C1)),
      link(Cat, B),
      rechts(C1,X,Y),
      lc(Cat,B,Y,Z).
```

```
link(np, s).
link(det,np).
link(ap,n1).
link(v,vp).
link(X,X).
```

Die Linktabelle für die zuletzt beschriebene Beispielsgrammatik gibt an, welche Knoten linke Ecken eines Mutterknotens darstellen. Durch die link-Aufrufe in den Parser wird diese Information jeweils überprüft, bevor das Verfahren fortgesetzt wird. Damit lassen sich aussichtslose Versuche, Knoten als linke Ecken zu etablieren, von vorneherein unterbinden. So wird der Parser mit der link-Information nicht versuchen, "vp" als linke Ecke eines Knotens zu parsen. Linktabellen lassen sich übrigens in einem Vorberechnungsprozeß auch automatisch erstellen.

Welche Parsing-Strategie die richtige ist, läßt sich nicht allgemein entscheiden. Dies ist stark abhängig von der Grammatik, die damit behandelt werden soll. Ein gravierender Einwand betrifft die Top-Down-Vorgehensweise: linksrekursive Regeln können nicht verwendet werden, da sie zur Nichtterminierung führen. Regeln, die sich direkt rekursiv aufrufen, können selbstverständlich z.B. durch das zuvor beschriebene Verfahren durch rechts-rekursive ersetzt werden. Anders verhält es sich mit indirekt rekursiven Aufrufen, die nicht mehr so leicht direkt aus der Grammatik ablesbar sind. Für ihre Erkennung benötigt man im allgemeinen Fall ein Programm. Doch die Eliminierung dieser indirekt linksrekursiven Aufrufe durch rechtsrekursive kann zu einer völlig veränderten und nicht mehr sehr intuitiven Darstellung der Grammatik führen. Bottom-up-basierte Parser wie die letzten beiden, die wir vorgestellt haben, umgehen dieses Problem elegant. Deren Nachteil beruht jedoch darin, daß die Verwendung von leeren (Epsilon-) Regeln zur Nichtterminierung führt. Schreibt man eine komplexere Grammatik (z.B. für eine natürliche Sprache), merkt man jedoch ziehmlich schnell, daß es ohne Epsilonregeln, sehr schwer wird, die Regeln redundanzfrei zu halten. Eine Grammatik ohne Epsilonregeln ist also im allgemeinen umfangreicher und redundanter.

Aussagenlogische Merkmalskodierung

Die Beschreibung der morphologischen und syntaktischen Eigenschaften eines gegebenen natürlichsprachlichen Ausdrucks erfolgte bisher mit den Mitteln einer um Merkmale angereicherten Phrasenstrukturgrammatik (= DCG). Für die Zwecke der Manipulation dieser Merkmale, z.B. um Kongruenz zwischen zwei Ausdrücken herzustellen, wurde ausschließlich das Mittel der Unifikation benutzt. In diesem Kapitel soll ein alternativer Ansatz vorgestellt werden: An der Stelle von Merkmalen werden aussagenlogische Formeln verwendet und statt Unifikation wird die Konsistenz einer gegebenen aussagenlogischen Formel überprüft. Diese alternative Kodierung linguistischer Systeme hat keinen Korrektheitsaspekt, da beide Verfahren zu äquivalenten Ergebnissen führen. Vielmehr handelt es sich dabei darum, eine kompaktere Kodierung zu erhalten, die u.U. zu effizienteren Programmen führt.

Betrachten wir folgende DCG, um uns den herkömmlichen nur auf Unifikation und Backtracking beruhenden Ansatz noch einmal klarzumachen:

```
np --> det, n.


det --> [der].
det --> [die].
n --> [astronom].
n --> [sonne].
```

Diese einfache Phrasenstrukturgrammatik ist nicht geeignet, die inkorrekten Ableitungen wie "die astronom" zu verhindern. Daher reichern wir das Lexikon um die Merkmale Kasus, Genus und Numerus an und die syntaktische Regel erhält einen variablen Parameter (P), der dazu dient, die Kongruenz zwischen den Teilbäumen zu erzwingen:

```
np2(P) --> det2(P), n2(P).


n2([nominativ,maskulin,singular]) --> [astronom].
n2([nominativ,feminin,singular]) --> [sonne].
n2([akkusativ,feminin,singular]) --> [sonne].
n2([dativ,feminin,singular]) --> [sonne].
n2([genitiv,feminin,singular]) --> [sonne].
det2([nominativ,feminin,singular]) --> [die].
det2([akkusativ,feminin,singular]) --> [die].
det2([nominativ,feminin,plural]) --> [die].
det2([akkusativ,feminin,plural]) --> [die].
det2([nominativ,maskulin,plural]) --> [die].
det2([akkusativ,maskulin,plural]) --> [die].
```

Die vollständige Angabe aller kombinatorischen Möglichkeiten führt, wie man sieht, schon bei drei unterschiedlichen Merkmalen zu einer großen Anzahl lexikalischer Einträge. Die Einträge, die auf dem (natürlichsprachlichen) Wort übereinstimmen, also homonym sind, verursachen beim Parsen Backtracking. Das Ziel besteht also darin, homonyme Lexikoneinträge möglichst völlig zu eliminieren. Natürlich kann man versuchen, nicht voll ausspezifizierte Merkmalsangaben, sondern Variablen zu verwenden. So kann man z.B. die 6 Einträge für "die" auf 4 reduzieren:

```
det2([nominativ,feminin,_]) --> [die].
det2([akkusativ,feminin,_]) --> [die].
det2([nominativ,maskulin,plural]) --> [die].
det2([akkusativ,maskulin,plural]) --> [die].
```

Weitere Reduktionen lassen sich mit dieser Methode jedoch nicht erzielen. Eine Verbesserungsmöglichkeit besteht in der Verwendung von laufzeitunabhängigen Ungleichungen über Variablen. Diese Ungleichungen schließen aus, daß eine gegebene Variable einen spezifizierten Wert annehmen kann. Derartige Aussagen lassen sich in Prolog mit dem Prädikat "dif" ausdrücken. Mithilfe von "dif" kann man die Anzahl der Einträge für "die" nochmals halbieren:

```
det2([K,feminin,_]) -->            {dif(K,dativ), dif(K,genitiv)},        [die].
det2([K,maskulin,plural]) -->      {dif(K,dativ), dif(K,genitiv)},        [die].
```

Schließlich kann man auch die letzte Homonymie eliminieren, wenn man "dif" auf mehrere Variablen gleichzeitig anwendet:

```
det2([K,G,N]) -->          {dif(K,dativ),    dif(K,genitiv),
                            dif([G,N], [maskulin,plural])},
                            [die].
```

Die Verwendung von "dif" scheint also für die Aufgabe, Homonymie und damit Backtracking zu eliminieren, völlig auszureichen. Bei der ausschließlichen Verwendung von Ungleichungen ist jedoch Vorsicht angebracht, da es sehr leicht passiert, daß eine Unifikation von zwei Variablen, die über "dif" restringiert wurden, zu inkorrekten Ergebnissen führt. Nimmt man z.B. die beiden folgenden Einträge

```
n2([nominativ,maskulin,singular]) --> [astronom].
det2([nominativ,feminin,singular]) --> [die].
```

und kodiert das Genusmerkmal via "dif" wie folgt:

```
n2([nominativ,G,singular]) -->        {dif(G,feminin)},      [astronom].
det2([nominativ,G,singular]) -->      {dif(G,maskulin)},   [die].
```

dann wird die Wortkette "die astronom" fälschlicherweise als korrekt akzeptiert. Dies liegt daran, daß hier keine Information darüber vorliegt, daß "feminin" das Komplement von "maskulin" ist. Geht man von Ungleichungen zu aussagenlogischen Restriktionen über, dann lassen sich solche Fälle vermeiden. Zweiwertige Merkmale wie Numerus werden einfach als eine Aussagenvariable kodiert, so daß die negierte und die unnegierte Aussage einen Widerspruch erzeugen. Sei z.B.

Plural = numerus
Singular = ¬ numerus

dann führt die Unifikation der obigen Variablen zu der inkonsistenten Formel

numerus & ¬ numerus = 1

Die Semantik der Konjunktion verlangt für die Wahrheit des gesamten Ausdrucks die Wahrheit beider Konjunkte. Da das eine Konjunkt aber das Gegenteil des anderen darstellt, kann dies nicht bewiesen werden. Die Kodierung von Merkmalen, deren Wertigkeit größer 2 ist, erfolgt unter Zuhilfenahme weiterer Aussagenvariablen. Die 4 Werte des Kasus lassen sich z.B. mit 2 Aussagenvariablen k1 und k2 wie folgt darstellen:

Nominativ = k1 & k2
Akkusativ = k1 & ¬ k2
Dativ = ¬k1 & k2
Genitiv = ¬k1 & ¬k2

Für die Kodierung des dreiwertigen Genusmerkmals benötigen wir ebenfalls 2 Aussagenvariablen:

Maskulin = g1 & g2
Feminin = g1 & ¬ g2
Neutrum = ¬g1 & g2

Die im ursprünglichen Ansatz atomaren Merkmale werden also aussagenlogisch auf Konjunktionen kleinerer Einheiten zurückgeführt. Mit diesen Setzungen läßt sich das obige Lexikon neu kodieren. So wird der Eintrag für "die" folgendermaßen definiert:

```
det(C)  --> [die],
            {alias([Nominativ,Akkusativ,_,_,_,Feminin,_,Singular,Plural ]),
            C = (Nominativ v Akkusativ)& ((Singular & Feminin) v Plural)}.
```

Die Semantik dieses Eintrags lautet:

Die Wortkette "die" ist analysierbar, wenn die aussagenlogische Formel in C zu wahr ausgewertet werden kann.

Um nicht die zugrundeliegenden Aussagenvariablen k1, k2 etc. verwenden zu müssen, haben wir diese in einer geordneten Liste in dem Fakt "alias" definiert:

```
alias([(k1 & k2),(k1 & ¬ k2),(¬k1 & k2),(¬k1 & ¬k2),(g1 & g2),(g1 & ¬ g2),(¬g1 & g2),n,¬n]).
```

Dabei gelten die oben beschriebenen Beziehungen:

```
%  alias([Nominativ,Akkusativ,Dativ,Genitiv,Maskulin,Feminin,Neutrum,Singular,Plural]).
```

Im Lexikoneintrag greifen wir auf diese Definition zurück. So können wir die aussagenlogische Formel übersichtlicher gestalten.

Angenommen wir möchten die Wortkette "die sonne" analysieren. Die Kodierung der morphologischen Merkmale des Wortes "sonne" lautet aussagenlogisch:

```
n(C) --> [sonne],
            {alias([_,_,_,_,_,Feminin,_,Singular,_]),
            C = Singular & Feminin}.
```

Die Wortkette "die sonne" ist demnach analysierbar, wenn die beiden aussagenlogischen Formeln zusammen zu wahr auswertbar sind. Dies läßt sich über die Semantik der Konjunktion ausdrücken, was zu der folgenden syntaktischen Regel führt:

```
np(P & Q) --> det(P), n(Q).
```

Kongruenz wird also nun nicht mehr über direkte Unifikation, sondern als konjunktive Verknüpfung von Teilformeln zu einer Gesamtformel definiert. Das Resultat der Analyse von "die sonne" lautet daher:

```
(Nominativ v Akkusativ)
& ((Singular & Feminin) v Plural)
&  Singular & Feminin
```

Daß die Formel zu wahr auswertbar ist, kann man in diesem Fall leicht ablesen. Für den allgemeinen Fall braucht man dazu natürlich ein automatisches Verfahren. Im Kapitel über Theorembeweisen haben wir bereits Beweisverfahren kennengelernt. Mit einigen wenigen Modifikationen können wir einen Programmteil daraus für unsere Zwecke verwenden. Beim Theorembeweisen benötigten wir einen Algorithmus, der für eine gegebene Formel die Wahrheit unter **allen** Belegungen mit Wahrheitswerten prüfen konnte. Hier geht es jedoch nicht um diese Tautologieeigenschaft, sondern um kontingente Wahrheit, d.h. man braucht nur zu zeigen, daß es mindestens **eine** Belegung mit Wahrheitswerten gibt, die die Formel wahr macht. Daher reicht es aus, den Programmteil "tabelle" aus dem Theorembeweiser zu extrahieren und mit der Grammatik in einer Prozedur "parse" zu verbinden:

```
parse(Formel,Kette,Belegung) :-
    np(Formel,Kette,[]),
    tabelle(Formel,Belegung,1).
```

Damit ist der alternative Ansatz zu reiner Unifikation plus Backtracking bereits korrekt definiert. Leider ist diese Anordnung nicht sehr effizient, da ja zunächst jede beliebige Formel konstruiert wird, ihre Wahrheit aber erst am Schluß überprüft wird. Effizienter ist ein Verfahren, das jede Teilformel, sobald sie bekannt wird, auf Konsistenz überprüft. Im Kapitel über Koroutinierung haben wir die Verwendung von "freeze" als Mittel der Verschränkung von Programmen beschrieben. Diese Technik läßt sich auch hier einsetzen. Wir definieren daher "parse" neu als:

```
parse(Formel,Kette,Belegung) :-
    tabellef(Formel,Belegung,1),
    np(Formel,Kette,[]).
```

Hierbei ist "tabelle" in "tabellef" umbenannt worden, welches wiederum selbst via "freeze" umdefiniert wird:

```
tabellef(Formel,Belegung,Wahrheitswert):-
    freeze(Formel,  tabelle(Formel,Belegung,Wahrheitswert)).
```

Ebenso werden alle Regeln von "tabelle" so modifiziert, daß die Bedingungen einer jeden Regel eingefroren werden. Aus

```
tabelle(  P & Q , B, 1):-
    tabelle(P, B, 1),
    tabelle(Q, B, 1).
```

wird so z.B.

```
tabelle(  P & Q , B, 1):-
    tabellef(P, B, 1),
    tabellef(Q, B, 1).
```

Eine letzte Modifikation betrifft die terminale Regel von "tabelle". Da in dieser Verwendung des Programms die Anzahl der Aussagenvariablen nicht im voraus bekannt ist, die Konsistenz der Liste der Belegungen aber so früh wie möglich überprüft werden soll, verlangen wir vor der Aufnahme eines Paares Aussagenvariable-Belegung in diese Liste, daß der komplementäre Fall, also die gleiche Aussagenvariable mit der inversen Belegung, nicht bereits vorliegt:

```
tabelle( F , B, Wert):-
    freeze(F,
    (
    atom(F),
    complement(Wert,Wert1),
    not(member0(F=Wert1,B)),
    member1(F=Wert,B),!
    )).
```

Da "B" eine Liste mit offenem Ende ist, benötigt man für die Überprüfung der Elementschaft noch spezielle "member"-Prozeduren, die im Anhang definiert sind.

Das gerade beschriebene Verfahren, das aussagenlogische oder Boole'sche Constraints statt Unifikation verwendet, hat natürlich nur Beispielscharakter. Um wirklich effizient zu arbeiten, müssen derartige Constraints direkt auf Unifikationsebene wirksam werden, statt über Koroutinierung. Außerdem sollte die aussagenlogische Formel nicht einfach verlängert werden, wie es unser Programm nahelegt, sondern zu neuen Formeln führen, die eine Simplifizierung der Ausgangsformeln darstellen. Professionelle Prologsysteme stellen derartig optimierte Algorithmen zur Verfügung.

Generierung - Maschinelle Übersetzung

Das System, dessen Beschreibung gerade abgeschlossen wurde, ist als reines Erkennungswerkzeug konzipiert. Die Eingabe ist immer ein deutscher Satz und die Ausgabe stellt verschiedene Ebenen der Verarbeitung dar: vom syntaktischen Baum über die semantische Repräsentation bis hin zum Ergebnis der Datenbankevaluierung. Dieses Kapitel widmet sich nun der inversen Seite der Spracherkennung: der Generierung natürlichsprachlicher Sätze. Generierung definieren wir als eingabegesteuerte Generierung, wobei die Eingabe diesmal aus der semantischen Repräsentation bestehen soll. Eine Grammatik, die beide Richtungen der Verarbeitung inkorporiert, nennt man reversibel. Die Grammatik, die die Grundlage des zuvor beschriebenen Systems bildet, ist nicht reversibel. Ihre Modifikation zu einer reversiblen Variante ist möglich, für die Zwecke der Demonstration wählen wir jedoch eine kleinere Grammatik, die die Ebenen der Syntax und der Semantik schon integriert enthält. Gegeben sei die folgende DCG, die einige Sätze des Englischen erlaubt:

```
sentence(P) --> np(X,P1,P),vp(X,P1).

np(X,P1,P) --> det(X,P2,P1,P), n1(X,P2).

n1(X,(P2&P1)) --> a(X,P1), n1(X,P2).
n1(X,Q) --> n(X,P), rel(X,P,Q).

vp(X,P) --> iv(X,P).
vp(X,P) --> cop(X,Y),a(Y,P).
vp(X,P) --> tv(X,Y,P1),np(Y,P1,P).

rel(X,P1,(P1&P2)) --> [that],vp(X,P2).
rel(_,P,P) --> [].

det(X,P1,P2, all(X,(P1=>P2))) --> [every].
det(X,P1,P2, exists(X,(P1&P2))) --> [a].

n(X,man(X)) --> [man].
n(X,woman(X)) --> [woman].

a(X,funny(X)) --> [funny].

cop(X,X) --> [was].
tv(X,Y,love(X,Y)) --> [loved].
iv(X,die(X)) --> [died].
```

Die Semantik wird als prädikatenlogische Formel aufgebaut, in der gleichen Weise wie wir es zuvor kennengelernt haben, mit dem Unterschied, daß die Junktoren nun als Infixoperatoren erscheinen und die Funktion von Lambdaausdrücken nun direkt von der Unifikation übernommen wird. Verwenden wir diese Grammatik als Parser, dann bekommen wir für einen Satz wie

```
every man loved a woman
```

das folgende semantische Resultat:

```
?- sentence(Semantik,[every,man,loved,a,woman],[]).

Semantik = all(_4,man(_4) => exists(_42,woman(_42) & love(_4,_42)))
```

Die reversible Funktion erlaubt, die Grammatik nun ein zweites Mal aufzurufen, diesmal mit der Semantik als Spezifikation und der Eingabekette als Variable:

```
?- sentence(all(_4,man(_4) => exists(_42,woman(_42) & love(_4,_42))), Satz,[]).

Satz = [every,man,loved,a,woman]
```

Die Grammatik hat also eine eindeutige Zuordnung zwischen den beiden Sprachen Englisch und Prädikatenlogik. Diese Fähigkeit ist in vielen Bereichen nützlich, z.B. in der Generierung von Antworten als Reaktion von Benutzerabfragen wie in unserem Datenbankbeispiel. So könnte diese Technik eingesetzt werden, um z.B. eine Abfrage wie

```
welcher astronom hat einen mond entdeckt der groesser als ein planet ist
```

mit dem Satz

```
herschel hat einen mond entdeckt der groesser als ein planet ist
```

zu beantworten. Eine Verbesserungsmöglichkeit einer solchen Eigenschaft liegt in ihrer Varianz, d.h. in der Fähigkeit zu paraphrasieren, um die Antwort des Systems nicht zu schematisch erscheinen zu lassen. So wäre die folgende Reaktion des Systems semantisch äquivalent aber aus kommunkativer Sicht vorzuziehen:

```
der astronom herschel entdeckte einen mond namens ariel, dessen durchmesser groesser ist als
der durchmesser des planeten uranus.
```

Ein derartiges System müßte also die Fähigkeit besitzen, aus einer prädikatenlogischen Formel mehrere natürlichsprachliche Paraphrasierungen zu generieren. Die kleine Grammatik des Englischen ist dazu bereits in der Lage. Nehmen wir folgende Abfrage:

```
?-  sentence(Semantik,[every,funny,man,loved,a,woman],[]).
```

so antwortet Prolog mit einer Belegung der Semantik,

```
Semantik = all(_4,man(_4) & funny(_4) => exists(_54,woman(_54) & love(_4,_54)))
```

die wiederum als Eingabe genommen, diesmal zwei Sätze generiert:

```
?- sentence(all(_4,(man(_4) & funny(_4) ) => exists(_42,woman(_42) & love(_4,_42))), Satz,[]).

Satz = [every,funny,man,loved,a,woman] ;
Satz = [every,man,that,was,funny,loved,a,woman] ;
no
```

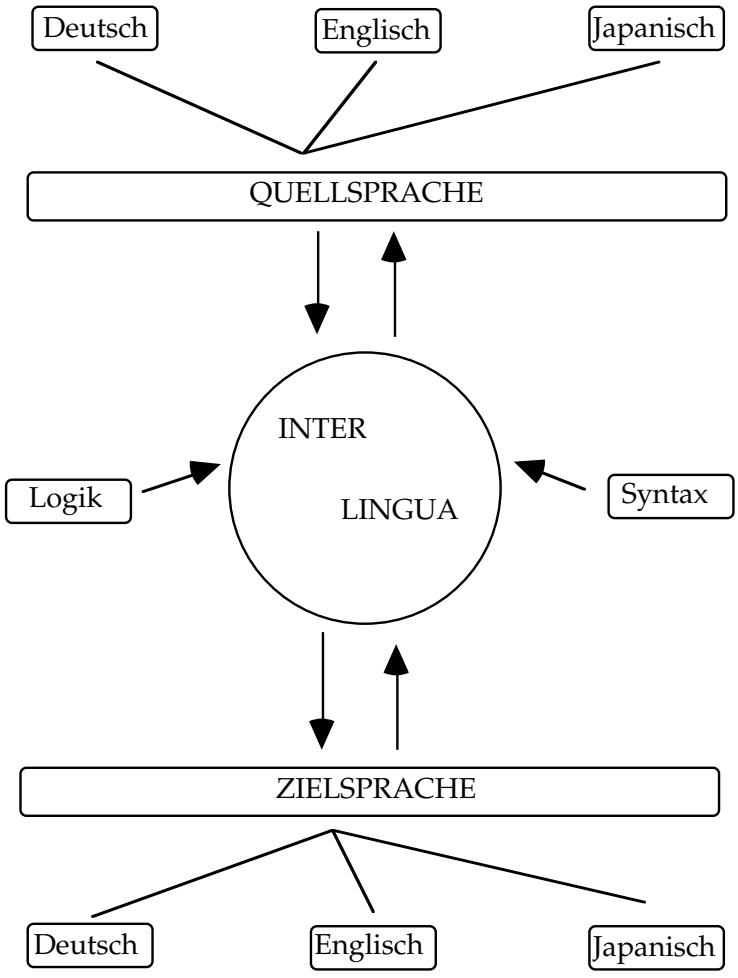
Die Eigenschaft "funny" wird einmal als pränominales Adjektiv und das zweite Mal als prädikativer Relativsatz realisiert. Dieses Resultat ist natürlich nur zu erwarten, wenn die Semantik in dieser Form auch mit beiden Regeln matchen kann. In der englischen Grammatik mußte bereits eine Modifikation vorgenommen werden, um dies zu ermöglichen: Innerhalb der Regel:

```
n1(X,(P2&P1)) --> a(X,P1), n1(X,P2).
```

haben wir die Reihenfolge der Semantik für das Adjektiv und der Semantik für das Nomen in die gleiche Ordnung gebracht, wie sie zwischen Nomen und Relativsatzsemantik herrscht. Andernfalls wäre die Reihenfolge Nomensemantik und Adjektivsemantik bzw. Nomensemantik und Relativsatzsemantik nicht dieselbe:

$$\begin{aligned} &\textit{semantik(Adjektiv + Nomen) = funny(X) \& man(X)} \\ &\qquad\qquad \neq \\ &\textit{semantik(Nomen + Relativsatz) = man(X) \& funny(X)} \end{aligned}$$

Diese semantischen Ausdrücke würden, obwohl sie logisch äquivalent sind, nicht matchen, da in Prolog Unifikation nichtkommutativ ist. Man sieht also, daß das Paraphrasierungsproblem auch ein Repräsentationsproblem ist. Eine Anwendungsmöglichkeit für ein System, das Paraphrasierungen enthält, haben wir schon erwähnt. Eine zweite wollen wir nun implementieren: die maschinelle Übersetzung. Das Verfahren, das wir dazu verwenden, die semantische Repräsentation als Bindeglied zwischen zwei einzelsprachlichen Grammatiken zu verwenden, wird Interlingua-Strategie genannt.



Dabei ist es prinzipiell möglich beliebige Sprachen als Zwischensprache zu verwenden, wenn sie die gleiche Ausdruckskraft besitzen. So kann man auch eine Art von syntaktischer Repräsentation als Interlingua verwenden oder sogar eine natürliche Sprache wie Englisch. In unserem Beispiel benutzen wir unsere prädikatenlogischen Ausdrücke als Zwischenrepräsentation. Die beiden folgenden Definitionen implementieren diese Vorgehensweise:

```
ed( Englisch, Deutsch):-
    sentence( Semantik, Englisch, []),
    satz( Semantik, Deutsch, []).

de( Deutsch, Englisch):-
    satz( Semantik, Deutsch, []),
    sentence( Semantik, Englisch, []).
```

Die erste Regel definiert den Übersetzungsvorgang von einer englischen zu einer deutschen Grammatik als Übergabe des semantischen Resultats des Quellsatzes als Eingabe in die Zielgrammatik, die damit den entsprechenden Satz der anderen Sprache generiert. Der gleiche Prozeß in der umgekehrten Reihenfolge ist für die Richtung Deutsch-Englisch mithilfe der zweiten Regel definiert. Die deutsche Grammatik für diesen Übersetzungsvorgang ist analog zur englischen aufgebaut und stellt eine Variante der am Anfang beschriebenen Beispielsgrammatiken dar. Ihre genaue Definition mit Übersetzungsbeispielen ist im Anhang zu finden. Auf zwei Besonderheiten möchten wir hier eingehen. Der erste Fall betrifft die Tatsache, daß manche natürlichsprachlichen Ausdrücke in einer verkürzten semantischen Form repräsentiert werden wie:

```
to kick the bucket
```

als

```
die(X)
```

Dies ist die Bedeutung des Verbs "sterben" im Englischen und um eine Paraphrasierung zu erlauben, ist die Semantik für "to die" und "to kick the bucket" die gleiche. Die folgenden Regeln implementieren diese Vorgehensweise:

```
iv(X,die(X)) --> [kicked,the,bucket].
iv(X,die(X)) --> [died].
```

Das umgekehrte Phänomen, nämlich daß einem natürlichsprachlichen Ausdruck ein längerer in der Semantik entspricht, ist bei dem Beispiel

```
unmarried man = bachelor
```

gegeben. Hier sieht die Implementation wie folgt aus:

```
n(X,man(X)) --> [man].
n(X,(man(X) & unmarried(X))) --> [bachelor].
a(X,unmarried(X)) --> [unmarried].
```

Diese Definitionen erlauben für einige Fälle eine Paraphrasierung der entsprechenden Semantiken. Dies ist jedoch nicht für alle Fälle möglich, da man nicht alle Abfolgen der semantischen Ausdrücke von vorneherein angeben kann. Das folgende Beispiel verdeutlicht dies. Das Ziel

```
?- de([jeder,tote,lustige,junggeselle,hat,eine,frau,geliebt], Englisch).
```

scheitert, obwohl die deutsche Grammatik eine Übersetzung in die logische Form findet:

```
?- satz(Logik,[jeder,tote,lustige,junggeselle,hat,eine,frau,geliebt],[]).

Logik =
all(_221,(
        (
                (man(_221) & unmarried(_221))
                &
                funny(_221)
        )
        &
        die(_221)
   =>   exists(_238,woman(_238) & love(_221,_238))))
```

Genauso findet die englische Grammatik eine Semantik für einen gleichbedeutenden englischen Satz:

```
?- sentence(Logik, [every,funny,bachelor,that,died,loved,a,woman],[]).
```

```
Logik =
all(_221,(
        (
                (man(_221) & unmarried(_221))
                &
                die(_221)
        )
        &
        funny(_221)
   =>   exists(_238,woman(_238) & love(_221,_238))))
```

Die beiden logischen Formeln unterscheiden sich jedoch in der Position von "funny(X)" und "die(X)". Die deutsche Grammatik konstruiert die Semantik für "tote lustige junggeselle" in der Reihenfolge "junggeselle lustige tote", wobei es beide Adjektive hinter die jeweilge Nomensemantik stellt. Eine passende (= unifizierende) englische Abfolge gibt es nicht, da es kein Adjektiv "dead" gibt. Der äquivalente Ausdruck mittels eines Relativsatzes "that died" wird mit der Nomensemantik vor die Adjektivsemantik gesetzt und bekommt so die inverse Reihenfolge, die nicht mehr mit der der deutschen Semantik unifizieren kann. Folgt man dieser Argumentation, dann erhält man das gewünschte Übersetzungsresultat, wenn man die Oberflächenreihenfolge der Adjektive einfach vertauscht:

```
?- satz(Logik,[jeder,lustige,tote,junggeselle,hat,eine,frau,geliebt],[]).

Logik =  all(_221,(((man(_221) & unmarried(_221)) & die(_221)) & funny(_221) =>
         exists(_238,woman(_238) & love(_221,_238))))
```

Wie man sieht ist dies der Fall und die entsprechende Übersetzung glückt:

```
?- de([jeder,lustige,tote,junggeselle,hat,eine,frau,geliebt], Englisch).

 Englisch=[every,funny,bachelor,that,kicked,the,bucket,loved,a,woman]
 Englisch=[every,funny,bachelor,that,died,loved,a,woman]
```

Das Übersetzungsproblem ist also auch ein Anordnungs- oder Repräsentationsproblem, dessen Lösung von allgemeinerer Art als die gerade vorgestellte sein sollte. So muß ein Übersetzungsprogramm auch in der Lage sein, Repräsentationen so verschiedener Sätze wie

peter schwimmt gerne
peter likes to swim

aufeinander abzubilden. Während sich der deutsche Satz nur aus einem Hauptsatz bestehend aus Eigennamen, Verb und Adverb zusammensetzt, ist die englische Übersetzung in einen Haupt- und einen untergeordneten Satz gegliedert. Da die Konstruktion der semantischen Repräsentation aber syntaktisch gesteuert wird, muß man für solche Fälle erst einen algorithmischen Weg finden, eine solche Ebenendifferenz nur für die Semantik aufzuheben.

Textverstehen und Diskursanalyse

Die computerlinguistische Behandlung von natürlichsprachlichen Texten steckt im Gegensatz zu der satzorientierten Sprachverarbeitung noch in den Kinderschuhen. Dies liegt natürlich zum einen an der Systematik, zuerst die weniger komplexen sprachlichen Phänomene zu betrachten. Auf der anderen Seite läßt sich die Dimension der textuellen Ebene auch nicht in einer einfachen Weise aus ihren Bestandteilen, den einzelnen Sätzen, herleiten. Die Semantik eines Textes kann also nicht direkt nach der Formel

$$\text{Semantik}(\text{Satz}^1) \cup ..... \cup \text{Semantik}(\text{Satz}^n) = \text{Semantik}(\text{Text})$$

berechnet werden, wie das folgende Beispiel belegt:

Ein Astronom entdeckt eine Galaxie. Er nennt sie Andromeda.

Die Semantik des ersten Satzes ist mit einem System wie dem unseren problemlos zu erstellen. Dagegen ist der zweite Satz isoliert überhaupt nicht behandelbar, da die Pronomen "er" und "ihn" nicht direkt referieren, also auf Individuen eines gegebenen Modells verweisen, sondern nur über den Umweg des Bezugs auf vorhergegangene Textsequenzen. Eine Möglichkeit, diese indirekte Beziehung zu den Individuen eines Modells herzustellen, besteht natürlich darin, die herkömmliche Satzsemantik so anzureichern, daß den Pronomen alle zuvor eingeführten Objekte (Diskursentitäten) zugänglich sind. Die Semantik des Textes setzt sich dann aus den Satzsemantiken plus den Bindungen ihrer Pronomen zusammen (Die Indexe kennzeichnen die Bindung), wobei gilt daß {1 = 3, 2 = 4 }:

Ein Astronom^1 entdeckt eine Galaxie^2. Er^3 nennt sie^4 Andromeda.

Daß es sich bei den Informationen über die zuvor eingeführten Individuen nicht nur um semantische Strukturen handeln darf, erkennt man an unserem Beispiel, in dem sich die Beziehung "Astronom - Er" bzw. "Galaxie - sie" nur bei Kenntnis der morpho-sytaktischen Merkmale maskulin/feminin erstellen läßt.
Sonst wären auch die Bindungen {1 = 4, 2 = 3} möglich. Daher sollte die Markierung der Elemente der einzelnen Sätze diese Tatsache berücksichtigen:

Ein Astronom$^{(1,mas)}$ entdeckt eine Galaxie$^{(1,fem)}$. Er$^{(3,mas)}$ nennt sie$^{(4,fem)}$ Andromeda.
{1 = 3, 2 = 4 }

Leider ist diese Vorstellung von Textsemantik immer noch viel zu simpel, um den möglichen Phänomenen Rechnung zu tragen. Ändern wir unseren Beispielsatz nur minimal ab, so ergeben sich neue Probleme:

Jeder Astronom$^{(1,mas)}$ entdeckt eine Galaxie$^{(2,fem)}$. Er$^{(3,mas)}$ nennt sie$^{(4,fem)}$ Andromeda.

Obwohl sich an den morpho-syntaktischen Verhältnissen nichts geändert hat, ist die vorherige Interpretation

$$\{1 = 3, 2 = 4\}$$

nicht nur nicht möglich, es muß sogar gelten, daß

$$\{1 \neq 3, 2 \neq 4\}$$

Die Unfähigkeit von Pronomen, auf Individuen zu referieren, die im Skopus eines Allquantors liegen, gilt allgemein. Dabei ist die Definition des Skopus von entscheidender Bedeutung, wie der folgende Satz zeigt:

Jeder Astronom$^{(1,mas)}$ der eine Galaxie$^{(2,fem)}$. entdeckt liebt sie$^{((,fem)}$
$\{2 = 4\}$

Hier kann die Bindung zustande kommen, weil das Pronomen selbst innerhalb des Skopus des Allquantors liegt.

Das größte Problem der Behandlung von Texten besteht nun aber nicht in der Identifikation der Besonderheiten wie im Falle der Interaktion zwischen Quantoren und Pronomen, sondern in der präzisen Definition des korrekten technischen Apparates, mit dessen Hilfe diese Phänomene behandelt werden können. Erinnern wir uns an die prädikatenlogische Semantik, die wir einem Satz wie dem letzten zuweisen würden:

```
fuer_alle(X, (astronom(X) & existiert(Y, galaxie(Y) & entdecken(X,Y)))=> lieben(X,Y)).
```

Diese Formel ist aber nicht geschlossen. Eine korrekte Übersetzung ist jedoch in unserem System, das sich an dem technischen Apparat von Montague orientiert, gar nicht zu erhalten. An dem Punkt nämlich, an dem die Semantik der VP mit der Semantik der Subjektsnp kombiniert wird

```
Z^lieben(Z,Q)
X^P^fuer_alle(X, (astronom(X) & existiert(Y, galaxie(Y) & entdecken(X,Y))) => P)
```

ist die einzige Variable, die noch zugänglich ist, die Variable Z bzw. X, die das Individuum der Subjektsnp bezeichnet. Die traditionelle kompositionelle Ansatz sieht das Verfügbarmachen von Individuenvariablen, die aus der Restriktion des Quantors stammen wie Y im letzten Beispiel, einfach nicht vor.

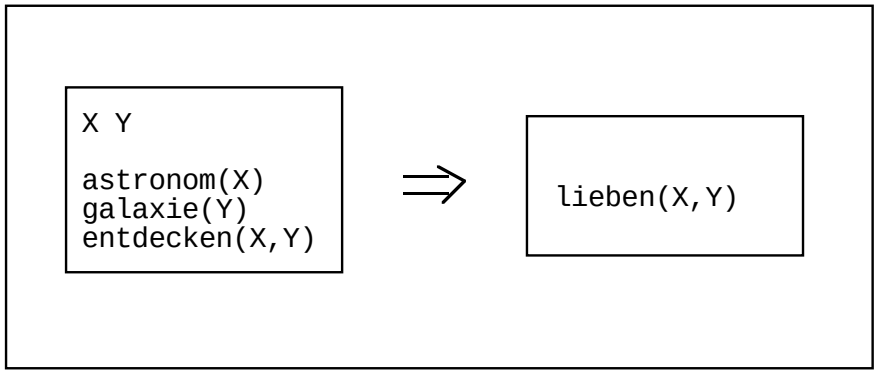
Eine mögliche Lösung dieses technischen Dilemmas stammt von H. Kamp. In seiner Diskursrepräsentationstheorie (DRT) geht er über den modelltheoretischen und wahrheitsfunktionalen Ansatz, wie er von Montague propagiert wurde, hinaus und präsentiert einen kontextbezogenen Wahrheitsbegriff. Statt Mengen von Formeln werden nun Diskursrepräsentationsstrukturen (DRS) als Semantik natürlichsprachlicher Sätze gebildet. Eine Diskursrepräsentation nach Kamp besteht aus zwei Teilen: dem Diskursuniversum U, das die zugänglichen Individuen (Referenzmarker) enthält. In unserer gerade eingeführten Notation wäre das z.B.

U = {(1,mas), (2,fem)}.

Die zweite Struktur einer Diskursrepräsentation enthält die Bedingungen oder Konditionen C. Dieses C entspricht den n-stelligen Prädikaten der Standardlogik, in unserem Beispiel etwa:

C = {astronom(X),galaxie(Y), entdecken(X,Y), lieben(X,Y) }

Das folgende Schaubild zeigt, wie unser Beispielsatz innerhalb der DRT dargestellt wird. Der äußere Kasten ist die oberste Diskursstruktur DRS0, deren Universum U0 leer ist. Das äußere C0 enthält als komplexe Bedingung zwei weitere DRSen, nähmlich DRS1 und DRS2. DRS1 besteht aus U1 = {X,Y} und C1 = {astronom(X),galaxie(Y), entdecken(X,Y), lieben(X,Y) }. DRS2 enthält keine Individuen, sondern nur C2 = {lieben(X,Y)}.



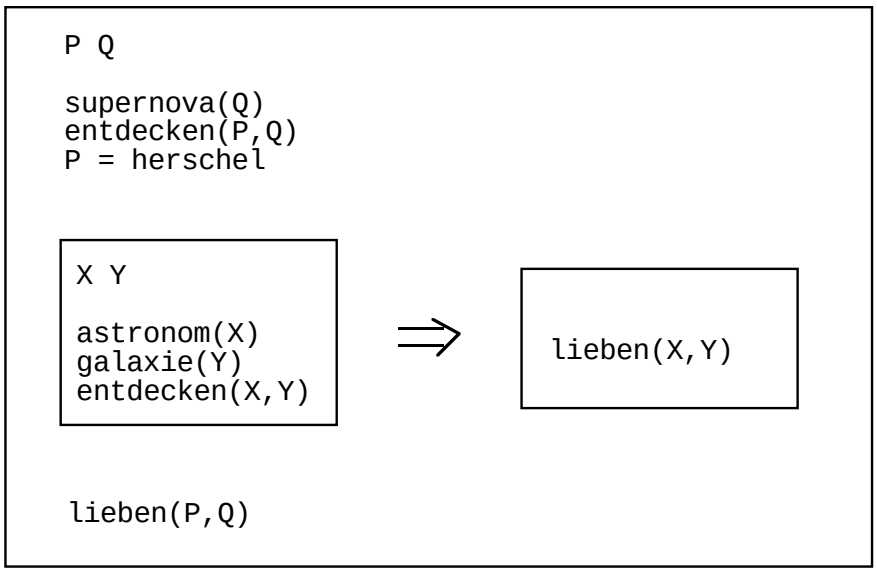
Ein entscheidender Begriff für die Referenzmöglichkeit von Pronomen ist die Hierarchie von Diskursrepräsentationen (DRSen). Die oberste DRS DRS0 ist der DRS1 übergeordnet (superordinate). DRS1 ist wiederum DRS2 übergeordnet. Die Zugänglichkeitsbedingung der DRT für Pronomina orientiert sich an diesem hierarchischen Aufbau:

Individuenvariablen können nur von Pronomina gebunden werden, die sich in derselben oder einer übergeordneten DRS befinden.

Das heißt, Individuenvariablen, die innerhalb von DRS0 eingeführt werden, sind von DRS1 und DRS2 aus zugänglich, aber nicht umgekehrt. Dazu einige Beispielsätze:

Herschel entdeckte eine Supernova.
Jeder Astronom der eine Galaxie entdeckt liebt sie
Herschel liebt sie.

und die Übersetzung in DRT-Notation:



Das Pronomen "sie" des dritten Satzes kann sich nur auf Q, nicht aber auf Y beziehen, da die Referenz eines Pronomens auf eine untergeordnete DRS laut Zugänglichkeitsdefinition nicht möglich ist. Satz 1 zeigt dabei, daß Artikel wie "eine" nicht wie "jeder" zu einer komplexen untergeordneten DRS führen, sondern in die aktuelle, d.h. oberste DRS eingebettet werden. Die prädikatenlogischen Junktoren Disjunktion und Negation führen zur Konstruktion von komplexen untergeordeten DRSen analog zu "==>". Die Implementierung (eine Adaptierung der Implementierung von Johnson/Klein) des kleinen DRT-Beispiels, das nun folgt, weicht geringfügig von dem gerade skizzierten Modell ab. Der einzig sichtbare Unterschied besteht darin, daß im Ergebnis die Individuenvariablen nicht alle zusammen als Universum vor den Konditionen stehen, sondern jede Variable ihrer Kondition vorausgeht.

Als Beispielgrammatik wählen wir wieder ein Programm, das Syntax und Semantik integriert. Eine Anpassung an unseren Modulansatz wäre aber genauso möglich. Die grundlegende Datenstruktur und einziges komplexes Argument der Grammatikregeln ist das folgende Achttupel:

(IN, OUT, DRS, INDEX, SCOPE, RES, ARG1, ARG2)

Die beiden Variablen IN und OUT enthalten die Listen der Individuenvariablen, also auch das jeweilige Universum. DRS bezeichnet die Konditionen des jeweiligen Knotens, zusammengesetzt aus seinem Skopus SCOPE und seiner Restriktion RES. ARG1 und ARG2 bezeichnen die Argumentstellen eines transitiven Verbs. INDEX bezeichnet das Argument eines einstelligen Prädikats, hier des Nomens.

Der Grundgedanke des Programms besteht darin, jedem syntaktischen Knoten ein Achttupel als Struktur zur Verfügung zu stellen, wobei dann bestimmte Stellen eines Tupels wie z.B. die Restriktion eines Quantors selbst wieder von dieser Datenstruktur ist . Das hat natürlich auch die Konsequenz, daß in vielen Fällen nicht alle Positionen der Tupel benötigt werden. Diese werden im Programm als anonyme Variable dargestellt. Der Kernmechanismus der DRT läßt sich an den Definitionen der beiden Quantoren zeigen:

```
det(Sem) --> [jeder],
    {
     Sem = (In,In,Drs, _,Res, Scope, _,_),
      Res = (_,_,Resdrs,_,_,_,_,_),
     Scope = (_,_,Scopedrs,_,_,_,_,_),
     Drs = (Resdrs ==> Scopedrs)
    }.

det(Sem) --> ([ein];[eine];[einen]),
    {
     Sem = (_,Out,Drs, _,Res, Scope, _,_),
      Res = (_,_,Resdrs,_,_,_,_,_),
     Scope = (_,Out,Scopedrs,_,_,_,_,_),
     Drs = [Resdrs,Scopedrs]
    }.
```

Während ein Allquantor keine Individuenvariablen den darüberliegenden DRSen zugänglich macht, was sich darin wiederspiegelt, daß die Inputliste gleich der Outputliste ist, erlaubt der Existenzquantor die Aufnahme der Individuenvariablen in die darüberliegenden DRSen. Die repräsentationelle Struktur ergibt sich aus den Skopus- und Restriktionsinformationen in einfacher Weise durch Gleichsetzung. Eine syntaktische Ebene weiter oben (Np) sehen wir, daß eine Nominalphrase bestehend aus Artikel und Nomen der Verbalphrase (Scope) diejenige Universumsinformation zukommen läßt, die aus dem Restriktionsbereich (SemN)

stammt. D.h., daß zum Beispiel alle Individuenvariablen, die von einem Nomen und einem davon abhängigen Relativsatz eingeführt werden, später einem Pronomen zugänglich sind, das innerhalb der Verbalphrase realisiert wird. Wie man an der Definition der zweiten Nominalphrase sieht, besteht die Aufgabe eines Pronomens darin, die hereinkommende Liste mit Individuenvariablen auf kongruierende Kandidaten zu durchsuchen und im Erfolgsfall die eigene Individuenvariablen damit zu unifizieren.

```
np(Sem) -->
    {
     Sem = (In,_,_, Index,SemN, Scope, _,_),
     SemN = (In,OutN,_,Index,_,_,_,_),
     Scope = ( OutN,_,_,_,_,_,_,_)
    },
     det(Sem),
     n(SemN).

np(Sem) -->
    {
     Sem = (In,Out,Drs, Index,_, Scope, _,_),
     Scope = ( In,Out,Drs,_,_,_,_,_)
    },
     pro(Index),
     {member(Index,In)}.
```

Die Verkettung der einzelnen Sätze eines Diskurses repräsentieren wir durch eine einzige Liste. Wir lassen also der Einfachheit halber alle Satzendezeichen und dergleichen weg. Somit reduziert sich die Verarbeitung der Diskursteile auf das Weiterreichen der Universumsinformation (In, Out) und der Verkettung der gesamten Repräsentationen zu einer Diskursstruktur (DrsS,DrsD):

```
d(In, Out, [DrsS | DrsD]) -->
      s((In, OutS, DrsS, _,_,_,_,_)),
      d(OutS, Out, DrsD).
d(X, X, []) --> [].
```

Die restlichen Regeln des kleinen Beispielprogramms führen wir zur Komplettierung auf:
In der Satzregel wird die Individuenvariable der Subjektsnominalphrase gleichgesetzt mit dem ersten Argument des zweistelligen Prädikats, das in der Verbalphrase eingeführt wird.

```
s(Sem) -->
    {
     Sem =  (_,_,_, Index, _, SemVP, _,_),
     SemVP = (_,_,_,_,_,_,_,Index,_)
    },
     np(Sem),
     vp(SemVP).
```

Analog zur Satzregel wird in der Verbalphrasenregel die Gleichsetzung von der Individuenvariable der Objektsnominalphrase mit der zweiten Position des Prädikats via Unifikation bewerkstelligt.

```
vp(Sem) -->
    {
     Sem =  (_,_,_, Index,_, SemV, A1,_),
     SemV = (_,_,_,_,_,_,A1,Index)
    },
     v(SemV),
     np(Sem).
```

Der n-Eintrag fügt der Liste der Individuenvariablen ein weiteres Element hinzu. Die v-Einträge dienen semantisch ausschließlich der Unifikation der Argumentstellen mit den Individuenvariablen von Subjekts- und Objektsnominalphrase:

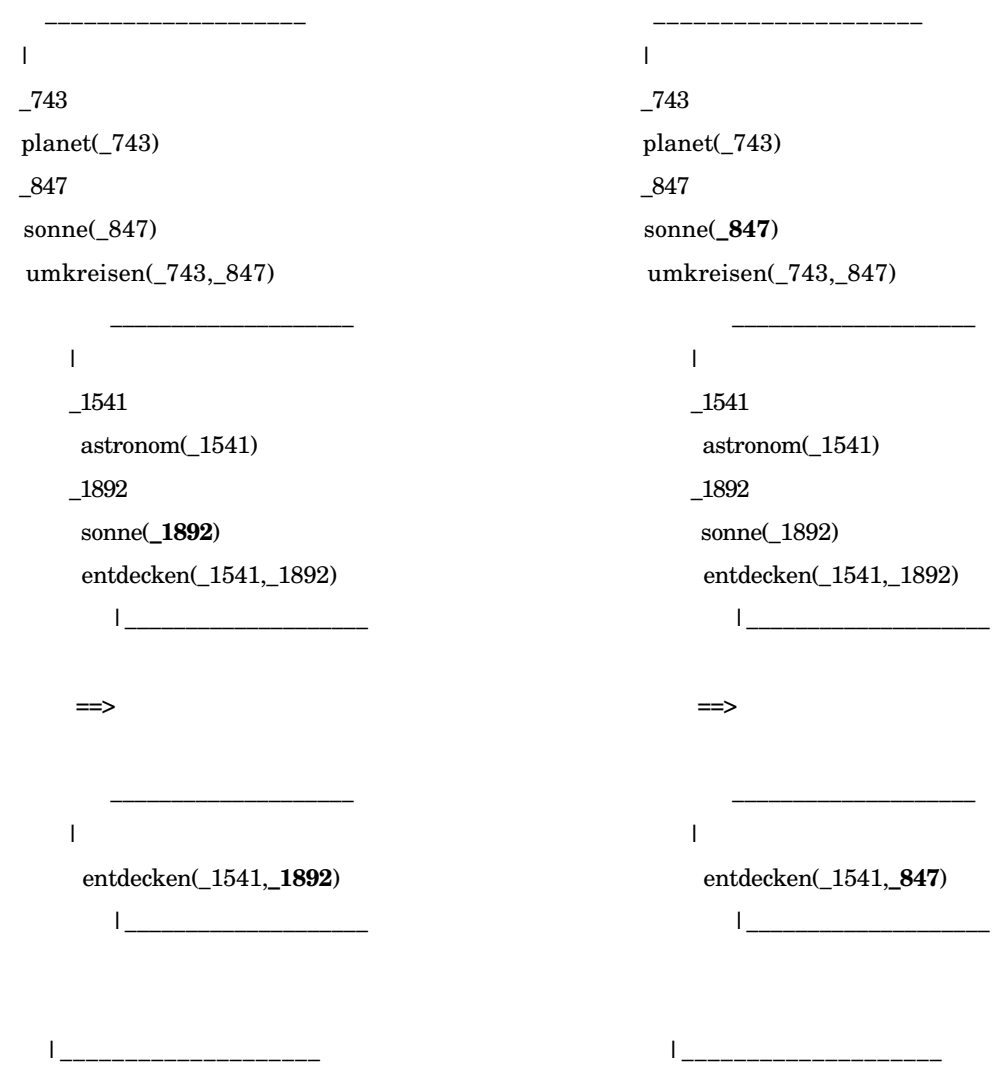
```
n((In, [Index|In],Drs,Index,_,_,_,_)) -->
      nomen(Index,Drs).

v((In,In,[umkreisen(A1,A2)],_,_,_,x(A1,_),x(A2,_))) --> [umkreist].
v((In,In,[entdecken(A1,A2)],_,_,_,x(A1,_),x(A2,_))) --> [entdeckt].
```

In "pro" und "nomen" werden die angereicherten Informationen (masc,fem) der Individuenvariablen bereitgestellt, die notwendig sind für die Überprüfung mit Kongruenzmerkmalen, die ein Pronomen auszeichnen:

```
pro(x( _, masc)) --> [er].
pro(x( _, fem)) --> [sie].

nomen(x(X, masc),[X,astronom(X)]) --> [astronom].
nomen(x(X, fem), [X,sonne(X)]) --> [sonne].
nomen(x(X, masc), [X,planet(X)]) --> [planet];[planeten].
```

Betrachten wir als Beispiel den folgenden Aufruf dieses Programms:

```
?- d([],_,DRS,[ein,planet,umkreist,eine,sonne,jeder,astronom,der,eine,sonne,entdeckt,entdeckt,sie],[]).
```



Die zwei resultierenden DRSen (in etwas anderer Notation) reflektieren die Tatsache, daß sich ein Pronomen, das in einer komplexen DRS eingeführt wird, auf übergeordnete DRSen beziehen kann.

Im Anhang ist das Programm in einer Erweiterung (Relativsatz) noch einmal wiedergegeben. Dort sind auch weitere Beispiele zu finden.

Wir gehen nun noch kurz auf den Evaluierungsaspekt von DRSen ein. Der Wahrheitsbegriff in der DRT ist praktisch identisch (es muß **mindestens** eine Einbettung der DRS in das Modell geben) mit dem Wahrheitsbegriff für prädikatenlogische Formeln. Daher können alle

Bedingungen einer nicht-komplexen DRS direkt dem zugrundeliegenden Inferenzsystem (Prolog) zur Auswertung übergeben werden.

```
einbettung(C):-   not(C=[_|_]),call(C).
```

Listen solcher Bedingungen werden einfach rekursiv abgearbeitet:

```
einbettung([]).
einbettung([H|T]):-
      einbettung(H),
      einbettung(T).
```

Individuenvariablen führen zu keiner Auswertung und werden einfach übergangen:

```
einbettung(C):-  var(C),!.
```

Komplexe DRSen, wie sie ein Allquantor induziert, werden schließlich auf die gleiche Art und Weise evaluiert wie ein prädikatenlogischer Allquantor (generalisierter Quantor !):

```
einbettung((U==>C)):-
      ¬(einbettung(U) & ¬(einbettung(C)).
```

Es darf keine Einbettung der Antezedent-DRS geben, so daß es keine Einbettung der Konsequenz-DRT gibt.

Bibliographie

Einführende Literatur

Zur Einführung in Prolog:

Bratko, I., PROLOG Programmierung für künstliche Intelligenz, Bonn, Addison-Wesley Verlag, 1987

Clocksin, W.F. und Mellish, C.S., Programming in Prolog (3rd edn), New York, Springer Verlag, 1987

Geske, Ulrich, Programmieren mit Prolog, Berlin, Akademie Verlag, 1988

Nigel Ford, Prolog Programming, Chichester, Wiley, 1989

O`Keefe, Richard A., The Craft of Prolog, Cambridge Mass.,
MIT Press, 1990

Zur Einführung in die natürlichsprachliche Syntax

Fanselow, Gisbert & Sascha W. Felix, Sprachtheorie, Bd.2 , Die Rektions- und Bindungstheorie, UTB, 1990

Zur Einführung in die Logik

Hodges, Wilfrid: Logic, Penguin Books, London, 1977

Zur Einführung in die Semantik

Cann, Ronnie: Formal Semantics, Cambridge University Press, Cambridge , 1993

Zur Einführung in die Pragmatik

Leech, Geoffrey, N.: Principles of Pragmatics, Longman, Harlow Essex, 1983

Zur Einführung in die Theorie der Formalen Sprachen

Brookshear, J.Glenn: Theory of Computation: Formal Languages, Automata, and Complexity, Benjamin/Cummings, Redwood City, 1989

Zur Einführung in die Computerlinguistik

Gazdar, Gerald und Chris Mellish, Natural Language Processing in Prolog: An Introduction to Computational Lingustics, Wokingham, Addison-Wesley, 1989

Pereira, Fernando C.N.; Shieber, Stuart M.: Prolog and Natural Language Analysis, Menlo Park CA u.a., 1987

Grishman, Ralph: Computational Linguistics, Cambridge University Press 1986

Zitierte Literatur

Pereira, Fernando C.N.; Shieber, Stuart M.: Prolog and Natural Language Analysis, Menlo Park CA u.a., 1987

Engel, Ulrich: Deutsche Grammatik, Heidelberg, 1988

Guenthner, Franz: Kursmaterialien: Natürlich-sprachliche Datenbankabfrage, unveröffentlichtes Manuskript, 1987

Guenthner, Franz: Computerlinguistik: 'A Personal View', CIS-Bericht-90-1, 1990

Johnson, Mark und Ewan Klein: Discourse, Anaphora and Parsing, CSLI-Report 86-63

Kalish, D.: Semantics, In: Edwards, D., Encyclopedia of philosophy VII, New York, Collier-MacMillan, 348-358

Fernando C.N. Pereira;David H.D. Warren: Definite clause grammars for language analysis, In: Readings in Natural Language Processing, Editor: Barbara J. Grosz, Karen Sparck-Jones and Bonnie Lynn Webber, Morgan Kaufmann,Los Alto, 1986 (1980)

Harrison, Michael H.: Introduction to formal language theory, Addison-Wesley, Reading (Mass.), 1978

Jerry R. Hobbs;Stuart M. Shieber: An algorithm for generating quantifier scopings, In: Computational Linguistics, Vol 13, 1-2, p. 47 ff, 1987

Montague, R.: The proper treatment of quanitification in ordinary English. In: J. Hintikka, J. Moravcsik, and P. Suppes, eds., Approaches to Natural Language. Dordrecht: Reidel. 1973

Pollard, Carl and Ivan Sag: Information Based Syntax and Semantics, Volume I, Center for the Study of Language and Information, Stanford, 1987

Index

Programmlisting

Modul Benutzerführung

```
:- op(1005, xfx, '--->').
:- op(20, xfy, &).
:- op(20, xfy, v).
:- op(25, xfy, =>).
:- op(25, xfy, <->).
:- op(10, fx, ¬).
:-  multifile  '--->'/2,'&'/2,'v'/2,'=>'/2'<->'/2,'¬'/1.

go :-
        write('Beenden der Interaktion mit "ende."'),
     repeat,
       nl,nl,nl,nl,nl,
      write('Geben Sie einen Satz ein ! '),  nl,
        write('Beenden Sie ihn mit "." oder "?" und RETURN. '),
     nl,
      write('>>  '),
      read_sent(Worte),
      los(Worte),
      Worte == [ende],
     !.

los(Eingabe) :- computerlinguistisches_problem(Eingabe, _),!.
los(_).

read_sent(Words) :- get(Char), read_sent(Char, Words).
read_sent(C, []) :- newline(C), !.
read_sent(C, Words):- space(C), !, get(Char), read_sent(Char, Words).
read_sent(Char, [Word|Words]):- read_word(Char, Chars, Next),
            name(Word, Chars), read_sent(Next, Words).

read_word(C, [], C) :- space(C), !.
read_word(C, [], C) :- newline(C), !.

read_word(Char, [Char|Chars], Last) :-
   get0(Next), read_word(Next, Chars, Last).
newline(46). /* . */
newline(63). /* ? *
space(32).
```

Modul Schnittstellen

```
computerlinguistisches_problem(Eingabe, Ausgabe):-
        grammatikalische_analyse( Eingabe, LogischeRepräsentation),
        anwendung( LogischeRepräsentation, Ausgabe).

anwendung(Eingabe, Ausgabe):-
        schreibe('EVALUIERUNG IM MODELL:'),
        evaluierung( Eingabe, Ausgabe).


grammatikalische_analyse( Eingabe, PragmatischeRepraesentation):-
        syntaktische_analyse(Eingabe, SyntaktischerBaum),
        nl,
        schreibe('SYNTAKTISCHE REPRAESENTATION:'),
        answer(SyntaktischerBaum),nl,nl,
        semantische_analyse(SyntaktischerBaum, [SemantischeRepraesentation,_,_]),
        schreibe('SEMANTISCHE REPRAESENTATION:'),
        answer(SemantischeRepraesentation),nl,nl,
        pragmatische_analyse(SyntaktischerBaum,SemantischeRepraesentation,
        PragmatischeRepraesentation),
        schreibe('PRAGMATISCHE REPRAESENTATION:'),
        schreibe(PragmatischeRepraesentation),nl,nl.

answer1(X):-
        answer(X),
        !.
answer1(_).
```

Modul Syntax

```
syntaktische_analyse( Eingabe, SyntaktischerBaum ):-
    s(finit, _, SyntaktischerBaum, Eingabe, [] ).

s(Form,[Subkat|SubkatRest]) --->
            np(Form,Subkat,Extrapos),
            vp(Form,[Subkat|SubkatRest],Extrapos).

s(finit,_)  --->
            aux(finit/Form,Subkat),
            s(Form,Subkat).

vp(Form,Subkat,Extrapos) --->
            vp2(Form,Subkat, ExtraposRest ),
            nachfeld([Extrapos|ExtraposRest]).

vp2(Form,Subkat,Extrapos) --->
            aux(Form/Form2,Subkat),
            vp1(Form2,Subkat, Extrapos ).

vp1(Form,Subkat,Extrapos) --->
            np_oder_ap(Form,Subkat,Extrapos),
            vg(Form,Subkat).

vg(Form,Subkat)  --->
            v(Form2,Subkat),
            aux(Form1,Subkat),
            { p(Form,Form1,Form2) }.

np(_,Subkat,[]) ---> en(Subkat).
np(Form,Subkat,Extrapos) --->
            det(Form,Subkat),
            n1(Subkat,Extrapos).

np_oder_ap(Form,[_|Rest],Extrapos)  --->
            nprec(Form,Rest,Extrapos).
np_oder_ap(Form,Subkat,Extrapos)  --->
            ap(Form,Subkat,Extrapos).
ap(Form,[Subkat1,Subkat2],Extrapos) --->
            komparativ([Subkat1,Subkat2]),
            als_np(Form,Subkat2,Extrapos).
```

```
nprec(_,[],[]) ---> [].
nprec(Form,[Subkat|SubkatRest],[Extrapos|ExtraposRest]) --->
            np(Form,Subkat,Extrapos),
            nprec(Form,SubkatRest,ExtraposRest).

n1(Subkat,Extrapos) --->
            n(Subkat),
            radj( Subkat , Extrapos).

radj( Subkat , Extrapos) ---> rs(rel,Subkat,Extrapos).

radj( Subkat ,Extrapos) ---> pp(Extrapos).

als_np(Form,Subkat2,[]) --->
            [als],
            np(Form,Subkat2,_).

als_np(Form,Subkat2,[als_np(Form,Subkat2,[])]) ---> [].


pp(Extrapos)  --->
            prep(Subkat,Form),
            np(Form, Subkat , Extrapos).

rs(Form,[_|Subkat],[]) --->
            rp([K|Subkat]),
            vp(Form,[[K|Subkat]|_],[]).
rs(_,Subkat, rs(Subkat )) ---> [].

nachfeld([]) ---> [].
nachfeld([H|T])  --->
            extraposition(H),
            nachfeld(T).

extraposition([]) ---> [].
extraposition(rs(Z))  --->
            rs(rel, Z, _).
extraposition(als_np(Form,Subkat,E))  --->
            als_np(Form,  Subkat,E).
```

Modul Lexikon

```
v(infinit,Subkat) ---> [entdeckt],        {subkat(entdecken,_,Subkat)}.
v(finit,Subkat) ---> [entdeckten],        {subkat(entdecken,plu,Subkat)}.
v(finit,Subkat) ---> [entdeckte],         {subkat(entdecken,sing,Subkat)}.

v(finit,Subkat) ---> [umkreist],          {subkat(umkreisen,sing,Subkat)}.
v(finit,Subkat) ---> [umkreisen],         {subkat(umkreisen,plu,Subkat)}.

v(finit,Subkat) ---> [besitzt],           {subkat(besitzen,sing,Subkat)}.
v(finit,Subkat) ---> [besitzen],          {subkat(besitzen,plu,Subkat)}.

v(finit,Subkat) ---> [sind],              {subkat(sein,plu,Subkat)}.
v(finit,Subkat) ---> [ist],               {subkat(sein,sing,Subkat)}.

v(infinit,Subkat) ---> [gegeben],         {subkat(es_geben,_,Subkat)}.
v(finit,Subkat) ---> [gibt],              {subkat(es_geben,sing,Subkat)}.

v([],_) ---> [].

komparativ(Subkat) ---> [groesser], { subkat(groesser_als,_,Subkat) }.
komparativ(Subkat) ---> [kleiner],  { subkat(groesser_als,_,Subkat) }.


aux(finit/[],Subkat)  ---> v(finit,Subkat).
aux(finit/infinit,_) ---> [hat].
aux(finit/infinit,Subkat) ---> [haben], {subkat(haben,plu,Subkat)}.
aux(X/X,_) ---> [], {(X = infinit; X = []; X = rel)}.

det(finit,Komp1) ---> dets(_,Komp1).
det(Form,Komp1) ---> dets(F,Komp1),
            {
            (Form = infinit; Form = []; Form = rel),
            (F = def; F = indef; F = all)
            }.
det(relpro,[_,sing,masc]) ---> [dessen].
det(relpro,[_,sing,neu]) ---> [dessen].
det(relpro,[_,sing,fem]) ---> [deren].
det(relpro,[_,plu,masc]) ---> [deren].
det(relpro,[_,plu,fem]) ---> [deren].
det(relpro,[_,plu,neu]) ---> [deren].
```

```
dets(all,[nom,sing,masc]) ---> [jeder].
dets(all,[akk,sing,masc]) ---> [jeden].
dets(def,[nom,sing,masc]) ---> [der].
dets(def,[dat,sing,fem]) ---> [der].
dets(def,[akk,sing,masc]) ---> [den].
dets(indef,[nom,sing,masc]) ---> [ein].
dets(indef,[akk,sing,masc]) ---> [einen].
dets(indef,[dat,sing,masc]) ---> [einem].
dets(indef,[nom,sing,fem]) ---> [eine].
dets(indef,[akk,sing,fem]) ---> [eine].
dets(indef,[dat,sing,fem]) ---> [einer].
dets(frage,[nom,sing,masc]) ---> [welcher].
dets(frage,[akk,sing,masc]) ---> [welchen].
dets(def,[nom,sing,fem]) ---> [die].
dets(def,[akk,sing,fem]) ---> [die].
dets(def,[nom,plu,fem]) ---> [die].
dets(def,[akk,plu,fem]) ---> [die].
dets(def,[nom,plu,masc]) ---> [die].
dets(def,[akk,plu,masc]) ---> [die].
dets(frage,[nom,plu,masc]) ---> [wieviele].
dets(frage,[nom,plu,masc]) ---> [welche].
dets(frage,[akk,plu,masc]) ---> [welche].
dets( def, [dat, plu, _] ) ---> [ Zahl ], { integer( Zahl ), Zahl > 1 }.
dets( def, [nom, plu, _] ) ---> [ Zahl ], { integer( Zahl ), Zahl > 1}.

n([nom,sing,masc]) ---> [astronom].
n([akk,sing,masc]) ---> [astronomen].
n([dat,sing,masc]) ---> [astronomen].
n([nom,plu,masc]) ---> [astronomen].
n([akk,plu,masc]) ---> [astronomen].
n([dat,plu,masc]) ---> [astronomen].

n([dat,sing,masc]) ---> [kilometer].
n([dat,plu,masc]) ---> [kilometern].
n([nom,sing,masc]) ---> [kilometer].
n([nom,plu,masc]) ---> [kilometer].

n([nom,sing,fem]) ---> [sonne].
n([nom,plu,fem]) ---> [sonnen].
n([akk,sing,fem]) ---> [sonne].
n([akk,plu,fem]) ---> [sonnen].
```

```
n([dat,sing,fem]) ---> [sonne].
n([dat,plu,fem]) ---> [sonnen].

n([nom,sing,fem]) ---> [erde].
n([akk,sing,fem]) ---> [erde].
n([dat,sing,fem]) ---> [erde].

n([nom,sing,masc]) ---> [planet].
n([nom,plu,masc]) ---> [planeten].
n([akk,sing,masc]) ---> [planeten].
n([akk,plu,masc]) ---> [planeten].
n([dat,sing,masc]) ---> [planeten].
n([dat,plu,masc]) ---> [planeten].

n([nom,sing,masc]) ---> [mond].
n([nom,plu,masc]) ---> [monde].
n([akk,sing,masc]) ---> [mond].
n([akk,plu,masc]) ---> [monde].
n([dat,sing,masc]) ---> [mond].
n([dat,plu,masc]) ---> [monden].

n([nom,sing,masc]) ---> [himmelskoerper].
n([nom,plu,masc]) ---> [himmelskoerper].
n([akk,sing,masc]) ---> [himmelskoerper].
n([akk,plu,masc]) ---> [himmelskoerper].
n([dat,sing,masc]) ---> [himmelskoerper].
n([dat,plu,masc]) ---> [himmelskoerpern].

n([nom,sing,masc]) ---> [durchmesser].
n([nom,plu,masc]) ---> [durchmesser].
n([akk,sing,masc]) ---> [durchmesser].
n([akk,plu,masc]) ---> [durchmessern].
n([dat,sing,masc]) ---> [durchmesser].
n([dat,plu,masc]) ---> [durchmessern].

n([nom,sing,fem]) ---> [entdeckung].
n([nom,plu,fem]) ---> [entdeckungen].
n([akk,sing,fem]) ---> [entdeckung].
n([akk,plu,fem]) ---> [entdeckungen].
n([dat,sing,fem]) ---> [entdeckung].
n([dat,plu,fem]) ---> [entdeckungen].
```

```
rp([nom,sing,masc]) ---> [der].
rp([akk,sing,masc]) ---> [den].
rp(S) ---> np(relpro,S,_).
rp([nom,plu,masc]) ---> [die].
rp([akk,plu,masc]) ---> [die].
rp([nom,sing,fem]) ---> [die].
rp([akk,sing,fem]) ---> [die].
rp([nom,plu,fem]) ---> [die].
rp([akk,plu,fem]) ---> [die].

prep([ dat, _, _], []) ---> [ von ].

en([Kasus,sing,masc]) ---> [herschel],
    { (Kasus = nom; Kasus = akk ; Kasus = dat) }.
en([nom,sing,_]) ---> [es].

en([Kasus,sing,masc]) ---> [Name],

    {  member(Name,
      [jupiter,mars,merkur,neptun,pluto,saturn,uranus,venus,
          adrastea,amalthea,ananke,ariel,callisto,carme,charon,
          diana,deimos,elara,enceladus,europa,ganymed,himalia,
         hyperion,iapetus,io,leda,lysithea,mimas,miranda,
        nereide,oberon,pasiphae,phobos,phoebe,rhea,sinope,tethys,
        titan,titania,triton,umbriel,'1979J2','1979J3','1980S1',
       '1980S3','1980S6','1980S13','1980S25','1980S26','1980S27',
         '1980S28', bond,cassini,dollfus,fountain,galilei,hall,
         huyghens,kuiper,lacques,lassell,melotte,nicholson,
        perrine,pickering,smith,tombaugh]) },

    { (Kasus = nom; Kasus = akk ; Kasus = dat) }.


p(rel,finit/infinit,  infinit).
p(rel,[]/[],  finit).
p(X,[]/[], X):- X = infinit; X = []; X = finit.

subkat(entdecken,Numerus,Subkat):-
    Subkat = [[nom,Numerus,_],[akk,_,_]];
    Subkat = [[akk,_,_],[nom,Numerus,_]].

subkat(sein,Numerus,Subkat):-
```

```
    Subkat = [[nom,Numerus,_],[nom,_,_]].

subkat(es_geben,_,Subkat):-
       Subkat = [[nom,sing,_],[akk,_,_]];
       Subkat = [[akk,_,_], [nom,sing,_]].



subkat(groesser_als,Numerus,Subkat):-
    Subkat = [[nom,Numerus,_],[nom,_,_]].

subkat(umkreisen,Numerus,Subkat):-
    Subkat = [[nom,Numerus,_],[akk,_,_]]; Subkat = [[akk,_,_],[nom,Numerus,_]].

subkat(besitzen,Numerus,Subkat):-
    Subkat = [[nom,Numerus,_],[akk,_,_]]; Subkat = [[akk,_,_],[nom,Numerus,_]].

subkat(haben,Numerus,Subkat):-
    Subkat = [[nom,Numerus,_],[akk,_,_]]; Subkat = [[akk,_,_],[nom,Numerus,_]].
```

Modul Semantik

```
semantische_analyse(SyntaktischerBaum,  Praedikatenlogik):-
        syn_sem(SyntaktischerBaum, Praedikatenlogik).


syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = v([Wort,_,[[nom|_],[_|_]]]),
    Logik = ((Y^V)^NP)^(X^NP),
   LogikTyp = (((e,t),t) , (e,t)),
    ConceptTyp = ((O/O)/O) / (S/S),
    praedikate(Wort, X^Y^V,S,O).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = v([Wort,_,[[akk|_],[_|_]]]),
    Logik = ((X^V)^NP)^(Y^NP),
   LogikTyp = (((e,t),t) , (e,t)),
    ConceptTyp = ((S/S)/S) / (O/O),
    praedikate(Wort, X^Y^V,S,O).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = n([Wort,_]),
   Logik = X^N,
   LogikTyp =  (e,t),
   ConceptTyp = T/T,
    praedikate(Wort, X^N,T).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = v([Wort,_,_]),
   Logik = X^V,
   LogikTyp =  (e,t),
   ConceptTyp = T/T,
    praedikate(Wort, X^V,T).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = det(dets([Name,Typ,_])),
    Logik = (X^N)^((X^VP)^ Det),
    LogikTyp =  ((e,t) ,((e,t),t)),
    ConceptTyp = (C/C) /((C/C)/C),
    quantoren(Name,Typ, X^N^VP^Det).
```

```
syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
      Syntax = prep([Wort|_]),
       Logik = ((X^V)^NP) ^ ((Y^N)^(Y^(N&NP))),
      LogikTyp = (((e,t),t) , ((e,t),(e,t))),
       ConceptTyp = ((O/O)/O) / ((S/S)/(S/S)),
       praedikate(Wort, X^Y^V,S,O).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
      Syntax = rp([_,_]),
       Logik = (Y^VP) ^ ((X^N)^(X^(N&VP))),
      LogikTyp =  ((e,t) , ((e,t),(e,t))),
       ConceptTyp = (C/C) / ((C/C)/(C/C)),
      X=Y.

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
       Syntax = en([Name,_]),
       Logik = (Name^VP)^VP,
      LogikTyp =  ((e,t),t),
       ConceptTyp = (T/T)/T,
       typ_name(Name,T).

syn_sem(komparativ([Wort,Subkat]), Logik ):-
        syn_sem(v([Wort,_,Subkat]), Logik ).

syn_sem(SynBaum,  Sem):-
       SynBaum =.. [_, Links, Rechts],
        semantische_analyse(Links, LSem),
        semantische_analyse(Rechts, RSem),
       beta_reduktion(LSem, RSem, Sem).
syn_sem(SynBaum,  Sem):-
       SynBaum =.. [_, Tochter],
        semantische_analyse(Tochter, Sem).
syn_sem([], [X^X,(E,E),C/C]).
syn_sem([Wort|_], [X^X,(E,E),C/C]):-
        member(Wort,[es,hat,haben,ist,sind,als]).


beta_reduktion( [P,E,A], [P^Q,(E,T),(A/B)],  [Q,T,B]).
beta_reduktion( [P^Q,(E,T),(A/B)],  [P,E,A], [Q,T,B]).

praedikate(von,  X^Y^gleich(X,Y),durchmesser_,kilometer_).
praedikate(von,  X^Y^besitzen(X,Y),durchmesser_,himmelskoerper_(_)).
```

```
praedikate(groesser, X^Y^groesser(X,Y),durchmesser_,durchmesser_).
praedikate(kleiner, X^Y^kleiner(X,Y),durchmesser_,durchmesser_).

praedikate(astronom, X^astronom(X),astronom_).
praedikate(astronomen, X^astronom(X),astronom_).

praedikate(planet, X^planet(X),himmelskoerper_(planet_)).
praedikate(planeten, X^planet(X),himmelskoerper_(planet_)).

praedikate(sonne, X^sonne(X),himmelskoerper_(sonne_)).
praedikate(erde, X^erde(X),himmelskoerper_(planet_)).
praedikate(himmelskoerper,X^himmelskoerper(X),himmelskoerper_(_)).

praedikate(mond, X^mond(X),himmelskoerper_(mond_)).
praedikate(monde, X^mond(X),himmelskoerper_(mond_)).

praedikate(durchmesser, X^durchmesser(X),durchmesser_).

praedikate(kilometer, X^kilometer(X),kilometer_).
praedikate(kilometern, X^kilometer(X),kilometer_).

praedikate(gibt, X^gleich(X,X),_).

praedikate(entdeckte, X^Y^entdecken(X,Y),astronom_,himmelskoerper_(_)).
praedikate(entdeckten, X^Y^entdecken(X,Y),astronom_,himmelskoerper_(_)).
praedikate(entdeckt, X^Y^entdecken(X,Y),astronom_,himmelskoerper_(_)).

praedikate(ist, X^Y^gleich(X,Y),himmelskoerper_(_),himmelskoerper_(_)).

praedikate(umkreist, X^Y^umkreisen(X,Y),himmelskoerper_(_),himmelskoerper_(_)).
praedikate(umkreisen, X^Y^umkreisen(X,Y),himmelskoerper_(_),himmelskoerper_(_)).

praedikate(besitzt, X^Y^besitzen(X,Y),himmelskoerper_(_),durchmesser_).
praedikate(besitzen, X^Y^besitzen(X,Y),himmelskoerper_(_),durchmesser_).


quantoren(_,indef, X^N^VP^existiert(X,N&VP)).
quantoren(_,all, X^N^VP^fuer_alle(X,N=>VP)).
quantoren(_,frage, X^N^VP^existiert(X,N&VP)).
quantoren(I,def, X^N^VP^existiert(X,(gleich(I,X)&N)&VP)):- integer(I),!.
 quantoren(_,def, X^N^VP^existiert(X,(N & fuer_alle(X2,N2 <-> gleich(X,X2))) & VP)):-
```

```
        freeze(N,copy_term(X^N,X2^N2)).


typ_name(Name,himmelskoerper_(_)):-
        member(Name,[jupiter,mars,merkur,neptun,pluto,saturn,uranus,
          venus,adrastea,amalthea,ananke,ariel,callisto,carme,
          charon,diana,deimos,elara,enceladus,europa,ganymed,
          himalia,hyperion,iapetus,io,leda,lysithea,mimas,miranda,
        nereide,oberon,pasiphae,phobos,phoebe,rhea,sinope,
        tethys,titan,titania,triton,umbriel,'1979J2','1979J3',
        '1980S1','1980S3','1980S6','1980S13','1980S25','1980S26',
        '1980S27','1980S28']).
typ_name(Name,astronom_):-
          member(Name,[tombaugh,herschel,nicholson,lassell,galilei,melotte,
          cassini,hall,bond,perrine,kuiper,pickering,
         huyghens,dollfus,fountain,lacques,smith]).
```

Modul Semantik2

```
semantische_analyse(SyntaktischerBaum,  Praedikatenlogik):-
        syn_sem(SyntaktischerBaum, Praedikatenlogik).


syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = v([Wort,_,[[nom|_],[_|_]]]),
    Logik = ((Y^V)^NP)^(X^NP),
    LogikTyp = (((e,t),t) , (e,t)),
    ConceptTyp = ((O/O)/O) / (S/S),
    praedikate(Wort, X^Y^V,S,O).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = v([Wort,_,[[akk|_],[_|_]]]),
    Logik = ((X^V)^NP)^(Y^NP),
    LogikTyp = (((e,t),t) , (e,t)),
    ConceptTyp = ((S/S)/S) / (O/O),
    praedikate(Wort, X^Y^V,S,O).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = n([Wort,_]),
    Logik = X^N,
    LogikTyp =  (e,t),
    ConceptTyp = T/T,
    praedikate(Wort, X^N,T).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = v([Wort,_,_]),
    Logik = X^V,
    LogikTyp =  (e,t),
    ConceptTyp = T/T,
    praedikate(Wort, X^V,T).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
    Syntax = det(dets([Name,Typ,_])),
    Logik = (X^N)^((X^VP)^ Det),
    LogikTyp =  ((e,t) ,((e,t),t)),
    ConceptTyp = (C/C) /((C/C)/C),
    quantoren(Name,Typ, X^N^VP^Det).
```

```
syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
       Syntax = det([Name,relpro,_]),
        Logik = (X^N)^((X^VP)^ ((Y^N2)^(Y^(N2&Det)))),
       LogikTyp =  ((e,t) , ((e,t) , ((e,t),(e,t))) ),
        ConceptTyp = (C/C) / ((C/C) / ((O/O)/(O/O)))  ,
        quantoren(Name,relpro, Y^X^N^VP^Det,O).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
       Syntax = prep([Wort|_]),
        Logik = ((X^V)^NP) ^ ((Y^N)^(Y^(N&NP))),
       LogikTyp = (((e,t),t) , ((e,t),(e,t))),
        ConceptTyp = ((O/O)/O) / ((S/S)/(S/S)),
        praedikate(Wort, X^Y^V,S,O).

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
       Syntax = rp([_,_]),
        Logik = (Y^VP) ^ ((X^N)^(X^(N&VP))),
       LogikTyp =  ((e,t) , ((e,t),(e,t))),
        ConceptTyp = (C/C) / ((C/C)/(C/C)),
      X=Y.

syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
       Syntax = en([Name,_]),
       Logik = (Name^VP)^VP,
      LogikTyp =  ((e,t),t),
       ConceptTyp = (T/T)/T,
        typ_name(Name,T).

syn_sem(komparativ([Wort,Subkat]),  Logik  ):-
        syn_sem(v([Wort,_,Subkat]), Logik ).


syn_sem(Syntax, [Logik, LogikTyp, ConceptTyp]):-
       Syntax = adj([Wort,_]),
        Logik = (X^N)^((X^N0)^((X^VP)^ Det))^ (X^VP)^Det,
      N0 = (N &
        ¬ existiert(Y, copy_term(X^VP,Y^VP0) & copy_term(X^N,Y^M) & M & GK & VP0)),
       LogikTyp =   ((e,t), (((e,t) ,((e,t),t)), ((e,t),t))),
         ConceptTyp = ((C/C)/ (((C/C) /((C/C)/C))/ ((C/C)/C))),
       praedikate(Wort, Y^X^GK,C,C).
```

```
syn_sem(SynBaum,  Sem):-
       SynBaum =.. [_, Links, Rechts],
        semantische_analyse(Links, LSem),
        semantische_analyse(Rechts, RSem),
       beta_reduktion(LSem, RSem, Sem).
syn_sem(SynBaum,  Sem):-
       SynBaum =.. [_, Tochter],
        semantische_analyse(Tochter, Sem).
syn_sem([],  [X^X,(E,E),C/C]).
syn_sem([Wort|_],  [X^X,(E,E),C/C]):-
        member(Wort,[es,hat,haben,ist,sind,als]).


beta_reduktion( [P,E,A], [P^Q,(E,T),(A/B)],   [Q,T,B]).
beta_reduktion( [P^Q,(E,T),(A/B)],   [P,E,A], [Q,T,B]).


praedikate(von,  X^Y^gleich((X,_),(Y,_)),durchmesser_,kilometer_).
praedikate(von,  X^Y^besitzen((X,T),(Y,durchmesser_)),durchmesser_,himmelskoerper_(T)).

praedikate(groesser,  X^Y^groesser(X,Y),durchmesser_,durchmesser_).
praedikate(kleiner,  X^Y^kleiner(X,Y),durchmesser_,durchmesser_).

praedikate(groessten,  X^Y^groesser(X,Y),durchmesser_,durchmesser_).
praedikate(kleinsten,  X^Y^kleiner(X,Y),durchmesser_,durchmesser_).

praedikate(astronom,  X^astronom(X),astronom_).
praedikate(astronomen,  X^astronom(X),astronom_).

praedikate(planet,  X^planet(X),himmelskoerper_(planet_)).
praedikate(planeten,  X^planet(X),himmelskoerper_(planet_)).

praedikate(sonne,  X^sonne(X),himmelskoerper_(sonne_)).
praedikate(erde,  X^erde(X)&gleich(X,erde),himmelskoerper_(planet_)).
praedikate(himmelskoerper,X^himmelskoerper(X),himmelskoerper_(_)).

praedikate(mond,  X^mond(X),himmelskoerper_(mond_)).
praedikate(monde,  X^mond(X),himmelskoerper_(mond_)).

praedikate(durchmesser,  X^durchmesser(X),durchmesser_).

praedikate(kilometer,  X^kilometer(X),durchmesser_).
```

```
praedikate(kilometern,  X^kilometer(X),durchmesser_).

praedikate(gibt,  X^es_gibt(_,(X,T)),T).

praedikate(entdeckte,  X^Y^entdecken((X,astronom_),(Y,T)),astronom_,himmelskoerper_(T)).
praedikate(entdeckten,
X^Y^entdecken((X,astronom_),(Y,T)),astronom_,himmelskoerper_(T)).
praedikate(entdeckt,  X^Y^entdecken((X,astronom_),(Y,T)),astronom_,himmelskoerper_(T)).

praedikate(ist,  X^Y^gleich(X,Y),himmelskoerper_(_),himmelskoerper_(_)).

praedikate(umkreist,
X^Y^umkreisen((X,T),(Y,T2)),himmelskoerper_(T),himmelskoerper_(T2)).
praedikate(umkreisen,
X^Y^umkreisen((X,T),(Y,T2)),himmelskoerper_(T),himmelskoerper_(T2)).

praedikate(besitzt,
X^Y^besitzen((X,T),(Y,durchmesser_)),himmelskoerper_(T),durchmesser_).
praedikate(besitzen,
X^Y^besitzen((X,T),(Y,durchmesser_)),himmelskoerper_(T),durchmesser_).

quantoren(_,relpro,
Y^X^N^VP^existiert(X,N&VP&besitzen((Y,T),(X,durchmesser_))),himmelskoerper_(T)  ).
quantoren(_,indef,  X^N^VP^existiert(X,N&VP)).
quantoren(_,all,  X^N^VP^fuer_alle(X,(N=>VP))).
quantoren(_,frage,  X^N^VP^existiert(X,N&VP)).
quantoren(I,def,  X^N^VP^existiert(X,(gleich(I,X)&N)&VP)):-  integer(I),!.
quantoren(_,def,  X^N^VP^existiert(X,N&VP)).

typ_name(Name,himmelskoerper_(_)):-
        member(Name,[jupiter,mars,merkur,neptun,pluto,saturn,uranus,
          venus,adrastea,amalthea,ananke,ariel,callisto,carme,
          charon,diana,deimos,elara,enceladus,europa,ganymed,
          himalia,hyperion,iapetus,io,leda,lysithea,mimas,miranda,
        nereide,oberon,pasiphae,phobos,phoebe,rhea,sinope,
        tethys,titan,titania,triton,umbriel,'1979J2','1979J3',
        '1980S1','1980S3','1980S6','1980S13','1980S25','1980S26',
        '1980S27','1980S28']).
typ_name(Name,astronom_):-
          member(Name,[tombaugh,herschel,nicholson,lassell,galilei,melotte,
          cassini,hall,bond,perrine,kuiper,pickering,
          huyghens,dollfus,fountain,lacques,smith]).
```

Modul Evaluierung-Variante1

```
wahr(Formel):-
     atomar(Formel),erfuellbar(Formel).
wahr(¬P):-
    nicht_erfuellbar(wahr(P)).
wahr( P & Q ):-
   wahr(P), wahr(Q).
wahr(P => Q ):-
   wahr(¬(P) v Q).
wahr(P v Q):-
    wahr(¬(¬(P) & ¬(Q))).
wahr(existiert(X,Formel)):-
    individuum(X), wahr(Formel).
wahr(fuer_alle(X,Formel)):-
    wahr(nicht(existiert(X,nicht(Formel)))).
wahr(P <-> Q ) :- wahr(P) , wahr(Q).
wahr(P <-> Q ) :- not(wahr(P)) , not(wahr(Q)).

erfuellbar(F):-  call(F).
nicht_erfuellbar(F):- call(F), !, fail.
nicht_erfuellbar(_).

atomar(F):- F =.. [R,X], relationsbezeichnung(R), individuum(X).
atomar(F):- F =.. [R,X,Y], relationsbezeichnung(R), individuum(X), individuum(Y).
atomar(F):- F =.. [R,X,Y,Z], relationsbezeichnung(R), individuum(X), individuum(Y),
individuum(Z).

individuum(X):-
     member(X,[sonne,erde,jupiter,mars,merkur,neptun,pluto,saturn,uranus,venus,
          adrastea,amalthea,ananke,ariel,callisto,carme,charon,
          diana,deimos,elara,enceladus,europa,ganymed,himalia,
          hyperion,iapetus,io,leda,lysithea,mimas,miranda,mond,
        nereide,oberon,pasiphae,phobos,phoebe,rhea,sinope,tethys,
         titan,titania,triton,umbriel,'1979J2','1979J3','1980S1',
       '1980S3','1980S6','1980S13','1980S25','1980S26','1980S27',
       '1980S28',
    1392000,12756,142800,6887,4878,49500,3000,120600,51800,12100,
      270,30,9000,4848,30,1500,1100,15,30,500,3126,5276,30,
      400,1500,3638,140,30,400,200,3473,300,1600,30,27,200,1400,
      1200,5150,1700,5000,700,
      bond,cassini,dollfus,fountain,galilei,hall,herschel,
```

```
        huyghens,kuiper,lacques,lassell,melotte,nicholson,
        perrine,pickering,smith,tombaugh]).

individuum(X):-  integer(X).

relationsbezeichnung(R):-
     member(R,[entdecken,umkreisen,besitzen,astronom,durchmesser,groesser,kleiner,gleich,
     sonne,erde,mond,planet,himmelskoerper,kilometer]).

%  db(Himmelskoerper,Typ,Durchmesser,Entdecker,Zentralgestirn).

himmelskoerper(Himmelskoerper):-
   db(Himmelskoerper,_,_,_,_).
planet(Himmelskoerper):-
   db(Himmelskoerper,planet,_,_,_).
mond(Himmelskoerper):-
   db(Himmelskoerper,mond,_,_,_).
sonne(Himmelskoerper):-
   db(Himmelskoerper,sonne,_,_,_).
erde(X):-
   X = erde,
   db(erde,planet,_,_,_).

durchmesser(Durchmesser):-
   db(_,_,Durchmesser,_,_).
durchmesser(X):-  integer(X).

astronom(Entdecker):-
   db(_,_,_,E,_), nonvar(E), E = Entdecker.

entdecken(Entdecker,Himmelskoerper):-
   db(Himmelskoerper,_,_,E,_),nonvar(E), E = Entdecker.
umkreisen(Himmelskoerper,Zentralgestirn):-
   db(Himmelskoerper,_,_,_,E),nonvar(E), E = Zentralgestirn.
besitzen(Himmelskoerper,Durchmesser):-
   db(Himmelskoerper,_,E,_,_),nonvar(E), E = Durchmesser.

kilometer(_).

groesser(X,Y):- X > Y.
kleiner(X,Y):- X < Y.
gleich(X,X).
```

Modul Evaluierung-Variante2

```
wahr(Formel):-  call(Formel).

existiert(_,Formel):-  call(Formel).
fuer_alle(_,Formel):-  call(Formel).
P => Q:- call(¬(P & ¬Q)).
P & Q :- call((P,Q)).
P v Q :- ( call(P) ; call(Q)).
¬ P :- not(P).
P <-> Q :- call(¬ ((P & ¬ Q) v (¬P & Q))).

himmelskoerper(Himmelskoerper):-
        db(Himmelskoerper,_,_,_,_).
planet(Himmelskoerper):-
        db(Himmelskoerper,planet,_,_,_).
mond(Himmelskoerper):-
        db(Himmelskoerper,mond,_,_,_).
sonne(Himmelskoerper):-
        db(Himmelskoerper,sonne,_,_,_).
erde(X):-
     X = erde,
      db(erde,planet,_,_,_).
durchmesser(Durchmesser):-
       db(_,_,E,_,_),nonvar(E), E = Durchmesser.
durchmesser(X):-  integer(X).

astronom(Entdecker):-
      db(_,_,_,E,_), nonvar(E), E = Entdecker.

entdecken(Entdecker,Himmelskoerper):-
        db(Himmelskoerper,_,_,E,_),nonvar(E), E = Entdecker.
umkreisen(Himmelskoerper,Zentralgestirn):-
        db(Himmelskoerper,_,_,_,E),nonvar(E), E = Zentralgestirn.
besitzen(Himmelskoerper,Durchmesser):-
        db(Himmelskoerper,_,E,_,_),nonvar(E), E = Durchmesser.

kilometer(_).

groesser(X,Y):- X > Y.
kleiner(X,Y):- X < Y.
gleich(X,X).
```

Modul Evaluierung-Variante3

```
wahr(Formel):-
      scan(Formel,F),
      call(F).


existiert(_,Formel):-  Formel.
fuer_alle(_,Formel):-  Formel.
P => Q:-  ¬ (P & ¬ Q).
P & Q :- P,Q.
P v Q :- P ; Q.
¬ P :- not(P).
P <-> Q :- ¬ ((P & ¬ Q) v (¬P & Q)).


%  db(Himmelskoerper,Typ,Durchmesser,Entdecker,Zentralgestirn).


himmelskoerper(Himmelskoerper):-
       db(Himmelskoerper,_,_,_,_).
planet(Himmelskoerper):-
       db(Himmelskoerper,planet,_,_,_).
mond(Himmelskoerper):-
       db(Himmelskoerper,mond,_,_,_).
sonne(Himmelskoerper):-
       db(Himmelskoerper,sonne,_,_,_).
erde(X):-
     X = erde,
      db(erde,planet,_,_,_).
durchmesser(Durchmesser):-
       db(_,_,E,_,_),nonvar(E), E = Durchmesser.
durchmesser(X):-  integer(X).

astronom(Entdecker):-
       db(_,_,_,E,_), nonvar(E), E = Entdecker.

entdecken((Entdecker,astronom_),(Himmelskoerper,Typ_)):-
      map(Typ_,Typ),
       db(Himmelskoerper,Typ,_,E,_),nonvar(E), E = Entdecker.
```

```
umkreisen((Himmelskoerper,Typ_),(Zentralgestirn,TypZ_)):-
      map(Typ_,Typ),
      map(TypZ_,TypZ),
       db(Himmelskoerper,Typ,_,_,E),nonvar(E), E = Zentralgestirn,
           (TypZ = planet ; TypZ = sonne),
            db(Zentralgestirn,TypZ,_,_,_).

besitzen((Himmelskoerper,Typ_),(Durchmesser,durchmesser_)):-
       map(Typ_,Typ),
        db(Himmelskoerper,Typ,E,_,_),nonvar(E), E = Durchmesser.


es_gibt(_,(X,_)):-
       nonvar(X),
      !.

es_gibt(_,(Himmelskoerper,himmelskoerper_(Typ_))):-
       map(Typ_,Typ),
        db(Himmelskoerper,Typ,_,_,_).
es_gibt(_,(Durchmesser,durchmesser_)):-
        db(_,_,Durchmesser,_,_).
es_gibt(_,(Entdecker,astronom_)):-
        db(_,_,_,E,_), nonvar(E), E = Entdecker.


kilometer(_).

groesser(X,Y):-  freeze(X,freeze(Y,X > Y)).
kleiner(X,Y):-  freeze(X,freeze(Y,X < Y)).
gleich(X,X).

scan(X,X):-  var(X),!.

scan(¬ A ,¬ B):-
      !, scan(A,B).

scan(fuer_alle(_,A => B),¬ (A1 & ¬ B1) ):-
      treat(A,A1),
      !, scan(B,B1).

scan(existiert(_,A),B):-
      !, scan(A,B).
```

```
scan((A & B),AB):-
      !, scan(A,A2),
      scan(B,B2),
      trues(A2,B2,AB).
scan((A v B),(A2 v B2)):-
      !, scan(A,A2),
      scan(B,B2).
scan(gleich(A,B),true):- !, A = B.
scan(copy_term(A,B),true):- !, copy_term(A,B).

scan(X,X):- X =.. [_,_,_|_].
scan(_,true).

treat( _ & R, R1):-
     !,
      scan(R, R1).
treat( N , N).

trues(true,B,B):- !.
trues(A,true,A):- !.
trues(¬A,B,B & ¬A):- !.
trues(A,B,A & B).

map(planet_,planet).
map(mond_,mond).
map(sonne_,sonne).
```

Modul Pragmatik

```
pragmatische_analyse(s( _, s(_,_)), Repraesentation, ja_nein_frage(Repraesentation)).

pragmatische_analyse(SynBaum ,Repraesentation, wert_frage_ein(Repraesentation)):-
    SynBaum = s(np(det(dets([_,frage,[_,sing,_]])),_), _) , !.

pragmatische_analyse(SynBaum ,Repraesentation, wert_frage_mehr(Repraesentation)):-
    SynBaum = s(np(det(dets([_,frage,[_,plu,_]])),_), _).

pragmatische_analyse(s(_,vp(_,_)),Repraesentation, aussage(Repraesentation)).



evaluierung( ja_nein_frage(Logik), ja ):-
    wahr( Logik),
    schreibe(ja), ! .
evaluierung( ja_nein_frage(_), nein ):- schreibe(nein).

evaluierung( wert_frage_ein(Logik), Wert ):-Logik =.. [_, Wert, _],
    wahr( Logik) ,!,
    schreibe(Wert).
evaluierung( wert_frage_ein(_), keine_werte_berechnet ).

evaluierung( wert_frage_mehr(Logik), Wert ):-
       Logik =.. [_, Wert, _],
         findall(Wert,wahr( Logik),Werte),
       sort(Werte,W),
       schreiben(W),
      !.
evaluierung( wert_frage_mehr(_), keine_werte_berechnet ).

evaluierung( aussage(Logik), relation_abgespeichert(Logik) ):-
    variablenfrei(Logik),
    assert(Logik),
    schreibe(relation_abgespeichert(Logik)),!.
evaluierung( aussage(Logik), relation_nicht_gespeichert ):-
    schreibe(relation_nicht_abgespeichert(Logik)).
```

```
variablenfrei( Logik ) :-
    Logik =..  [ Relation | T],
    not( quantor( Relation ) ),
    not( junktor1( Relation ) ),
    not( junktor2( Relation ) ),
    not((member(X,T),  var(X))),
    not(call(Logik)).

schreibe(X):-  write(X),  nl,nl.


schreiben([]).
schreiben([H|T]):-
        write(H), write(' '), schreiben(T).
```

Modul Datenbank

```
/* db(Himmelskoerper, Typ, Durchmesser, Entdecker, UmkreisterHimmelskoerper). */

db(sonne, sonne, 1392000,_,_).
db(erde,planet,12756,_,sonne).
db(jupiter,planet,142800,_,sonne).
db(mars,planet,6887,_,sonne).
db(merkur,planet,4878,_,sonne).
db(neptun,planet,49500,_,sonne).
db(pluto,planet,3000,tombaugh,sonne).
db(saturn,planet,120600,_,sonne).
db(uranus,planet,51800,herschel,sonne).
db(venus,planet,12100,_,sonne).
db(adrastea,mond,_,_,jupiter).
db(amalthea,mond,270,_,jupiter).
db(ananke,mond,30,nicholson,jupiter).
db(ariel,mond,900,lassell,uranus).
db(callisto,mond,4848,galilei,jupiter).
db(carme,mond,30,melotte,jupiter).
db(charon,mond,1500,_,jupiter).
db(diana,mond,1100,cassini,saturn).
db(deimos,mond,15,hall,mars).
db(elara,mond,30,nicholson,jupiter).
db(enceladus,mond,500,herschel,saturn).
db(europa,mond,3126,galilei,jupiter).
db(ganymed,mond,5276,galilei,jupiter).
db(himalia,mond,30,perrine,jupiter).
db(hyperion,mond,400,bond,saturn).
db(iapetus,mond,1500,cassini,saturn).
db(io,mond,3638,galilei,jupiter).
db(leda,mond,140,perrine,jupiter).
db(lysithea,mond,30,nicholson,jupiter).
db(mimas,mond,400,herschel,saturn).
db(miranda,mond,200,kuiper,uranus).
db(mond,mond,3473,_,erde).
db(nereide,mond,300,kuiper,uranus).
db(oberon,mond,1600,herschel,uranus).
db(pasiphae,mond,30,nicholson,jupiter).
db(phobos,mond,27,hall,mars).
db(phoebe,mond,200,pickering,saturn).
db(rhea,mond,1400,cassini,saturn).
db(sinope,mond,_,_,jupiter).
db(tethys,mond,1200,cassini,saturn).
db(titan,mond,5150,huyghens,saturn).
db(titania,mond,1700,herschel,uranus).
db(triton,mond,5000,lassell,neptun).
db(umbriel,mond,700,lassell,uranus).
db('1979J2',mond,_,_,jupiter).
db('1979J3',mond,_,_,jupiter).
db('1980S1',mond,_,dollfus,saturn).
db('1980S3',mond,_,fountain,saturn).
db('1980S6',mond,_,lacques,saturn).
db('1980S13',mond,_,smith,saturn).
db('1980S25',mond,_,_,saturn).
db('1980S26',mond,_,_,saturn).
db('1980S27',mond,_,_,saturn).
db('1980S28',mond,_,_,saturn).
```

Modul Skopus

```
skopus_analyse(SynBaum,SynBaumi):-
        scope(SynBaum,SynBaumi,Vars,[],Nps,[]),
        einsetzen(Nps,Vars).

scope(B,Bi,V1,V3,F1,F3):-
       dif(M,np),
       B =.. [M,L,R],
       Bi =.. [M,L1,R1],
       scope(L,L1,V1,V2,F1,F2),
       scope(R,R1,V2,V3,F2,F3).

scope(B,Bi,V1,V2,F1,F2):-
       B =.. [M,L],
       Bi =.. [M,L1],
       scope(L,L1,V1,V2,F1,F2).

scope(np(L,R),X,[X|V],V,[np(L,R)|F],F).
scope(B,B,V,V,F,F):-
       B =.. [M,[_|_]]; B =.. [M,[]].

einsetzen([],[]).
einsetzen([X|Y],Vars):-
       delete(X,Vars,RestVars),
       einsetzen(Y,RestVars).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|Z1]):- dif(X,Y), delete(X,Z,Z1).
```

```
geschlossene_formel(Formel) :- wohl_geformte_formel(Formel, []).

wohl_geformte_formel(W,X) :- atomare_formel(W,X).
wohl_geformte_formel(WGF,X):- WGF =.. [J,W1,W2],
                    junktor2(J),
                    wohl_geformte_formel(W1,X),
                    wohl_geformte_formel(W2,X).
wohl_geformte_formel(WGF,X) :- WGF =.. [J,W],
                    junktor1(J),
                    wohl_geformte_formel(W,X).
wohl_geformte_formel(WGF,X) :-  WGF =.. [Q,V,W],
                    quantor(Q),
                    variable(V),
                    wohl_geformte_formel(W,[V|X]).

junktor2(&).
junktor2(v).
junktor2(=>).
junktor2(<->).
junktor1(¬).

quantor(fuer_alle).
quantor(existiert).

atomare_formel(T,X):- T =.. [P|A], praedikat(P),  argumente(A,X).
atomare_formel(P,_):- praedikat(P).

praedikat(P) :- atom(P), not((P = []; quantor(P); junktor2(P); junktor1(P))).

argumente([T],X) :- term(T,X).
argumente([T|R],X) :- term(T,X), argumente(R,X).

term(T,_) :- atomic(T).
term(T,X) :- variable(T), element(T,X).

variable(X) :- var(X).

element(X, [S|_]):- X == S.
element(X, [_|T]):- element(X,T).
```

Modul Aussagenlogische Merkmalskodierung

```
:- op(20, xfy, &).
:- op(20, xfy, v).
:- op(25, xfy, =>).
:- op(25, xfy, <->).
:- op(10, fx, ¬).

tabellef(F,B,W):-
      freeze(F, tabelle(F,B,W)).

tabelle( P & _ , B, 0):-
      tabellef(P, B, 0).
tabelle( _ & Q , B, 0):-
      tabellef(Q, B, 0).
tabelle( P & Q , B, 1):-
      tabellef(P, B, 1),
      tabellef(Q, B, 1).

tabelle( P v Q , B, 0):-
      tabellef(P, B, 0),
      tabellef(Q, B, 0).
tabelle( P v _ , B, 1):-
      tabellef(P, B, 1).
tabelle( _ v Q , B, 1):-
      tabellef(Q, B, 1).

tabelle( P => Q , B, 0):-
      tabellef(P, B, 1),
      tabellef(Q, B, 0).
tabelle( P => _ , B, 1):-
      tabellef(P, B, 0).
tabelle( _ => Q , B, 1):-
      tabellef(Q, B, 1).

tabelle( P <-> Q , B, 1):-
      tabellef(P, B, 1),
      tabellef(Q, B, 1).
tabelle( P <-> Q , B, 1):-
      tabellef(P, B, 0),
      tabellef(Q, B, 0).

tabelle( P <-> Q , B, 0):-
      tabellef(P, B, 1),
      tabellef(Q, B, 0).
tabelle( P <-> Q , B, 0):-
      tabellef(P, B, 0),
      tabellef(Q, B, 1).

tabelle( ¬ F , B, 0):-
      tabellef(F, B, 1).
tabelle( ¬ F , B, 1):-
      tabellef(F, B, 0).

tabelle( F , B, Wert):-
      freeze(F,
     (
      atom(F),
       complement(Wert,Wert1),
      not(member0(F=Wert1,B)),
      member1(F=Wert,B),!
     )).

member0(A, [X|_]):-
      nonvar(X),
     X=A.
member0(A, [C|B]) :-
      nonvar(C),
      not(A = C),
      member0(A, B).

member1(A, [A|_]).
member1(A, [C|B]) :-
      not(A = C),
      member1(A, B).

complement(0,1).
complement(1,0).
```

```
parse(Formel,Kette,Belegung) :-
        tabellef(Formel,Belegung,1),
        np(Formel,Kette,[]),
        ! .

np(P & Q) --> det(P), n(Q).

n(C) --> [astronom],
        {alias([Nominativ,_,_,_,Maskulin,_,_,Singular,_]),
        C = Nominativ & Singular & Maskulin}.

n(C) --> [astronomen],
        {alias([Nominativ,_,_,_,Maskulin,_,_,Singular,Pural]),
        C = ((¬Nominativ &  Singular) v Pural) & Maskulin}.

n(C) --> [sonne],
        {alias([_,_,_,_,_,Feminin,_,Singular,_]),
        C = Singular & Feminin}.

det(C)  --> [der],
        {alias([Nominativ,_,Dativ,Genitiv,Maskulin,Feminin,_,Singular,Plural]),
        C = (((Nominativ & Maskulin)
        v ((Dativ  v  Genitiv) & Feminin))
        &  Singular)
        v (Genitiv & Maskulin & Plural)}.

det(C)  --> [die],
        {alias([Nominativ,Akkusativ,_,_,_,Feminin,_,Singular,Plural]),
        C = (Nominativ  v  Akkusativ)
        & ((Singular & Feminin) v Plural)}.


alias([(k1 & k2),(k1 & ¬ k2),(¬k1 & k2),(¬k1 & ¬k2),(g1 & g2),(g1 & ¬ g2),(¬g1 & g2),n,¬n]).

%  alias([Nominativ,Akkusativ,Dativ,Genitiv,Maskulin,Feminin,Neutrum,Singular,Plural]).
```

Modul Compiler

```
:- op(1005, xfx, '--->').
% In der Grammatik muss --> in ---> umgewandelt werden

uebersetze( (P1 ---> P2), (G1 :- G2) ):-
      linke_seite(B2, P1, S0, S, G1 ),
      rechte_seite( B, P1, P2, S0, S, G2 ),
       P1 =.. [Phrasensymbol | _],
       B2 =.. [Phrasensymbol | B].



linke_seite(B, P0, S0, S, G ):-
      fuege_hinzu(B, P0, S0, S, G ).

rechte_seite( B, P,  (P1, P2) , S0, S, G ):-
      rechte_seite( B1, P,  P1, S0, S1, G1 ),
      rechte_seite( B2, P, P2, S1, S, G2 ),
      append(B1,B2,B),
      und( G1, G2, G) , !.
rechte_seite([], _,{X} , S, S, X ).
rechte_seite([P1],P0,  P , S0, S, true ):-
      P0 =.. [_ | T],
      P = [Wort],
      P1 = [Wort | T],
      append( P, S, S0 ).
rechte_seite([[]],_, [] , S, S, true ).
rechte_seite([P0],_, P , S0, S, G ):-
      fuege_hinzu(P0, P, S0, S, G ).



fuege_hinzu(B, P, S0, S, G ):-
       P =.. [Phrasensymbol | Argumente],
         not((Phrasensymbol = ','; Phrasensymbol = '.';Phrasensymbol =[])),
       append( Argumente, [B, S0, S], AlleArgumente),
        G =.. [ Phrasensymbol | AlleArgumente].

und( true, G, G) .
und( G, true, G) .
und( G1, G2, (G1, G2)).
```

```
loesche_alte_definitionen  :-
        findall(F/S, ((A ---> _),A=..[F|Args],length(Args,Le),S is Le + 3 ), L),
       sort(L,K),
       loesche(K),
      !.



loesche([]).
loesche([H|T]):-
       abolish(H),
        write(H),write(' '),
       loesche(T).

uebersetze_neue_definitionen  :-
       (A ---> B),
        uebersetze((A ---> B), (C :- D)),
        assert( (C :-  D) ),
        write('.'),
       fail.

% Aufruf des Compilers
compile  :-
        write('DCG-Compiler'),nl,
        write('Uebersetzt ---> Regeln in Prolog + Differenzliste + Argument fuer Syntaxbaum'),nl,
        write('Loesche alte Definitionen: '),
      nl,
        loesche_alte_definitionen,
      nl,
        write('Beginne Kompilierung: '),
       uebersetze_neue_definitionen.
compile :- nl,write(' Kompilierung beendet '), nl.
```

Modul Maschinelle Übersetzung

```
ed( Englisch, Deutsch):-
    sentence( Semantik, Englisch, []),
    satz( Semantik, Deutsch, []).

de( Deutsch, Englisch):-
    satz( Semantik, Deutsch, []),
    sentence( Semantik, Englisch, []).

% Englische Grammatik:

sentence(P) --> np(X,P1,P),vp(X,P1).

np(X,P1,P) --> det(X,P2,P1,P), n1(X,P2).

n1(X,(P2&P1)) --> a(X,P1), n1(X,P2).
n1(X,Q) --> n(X,P), rel(X,P,Q).

vp(X,P) --> iv(X,P).
vp(X,P) --> cop(X,Y),a(Y,P).
vp(X,P) --> tv(X,Y,P1),np(Y,P1,P).

rel(X,P1,(P1&P2)) --> [that],vp(X,P2).
rel(_,P,P) --> [].

det(X,P1,P2, all(X,(P1=>P2))) --> [every].
det(X,P1,P2, exists(X,(P1&P2))) --> [a].

n(X,man(X)) --> [man].
n(X,woman(X)) --> [woman].
n(X,(man(X) & unmarried(X))) --> [bachelor].

a(X,funny(X)) --> [funny].
a(X,unmarried(X)) --> [unmarried].

cop(X,X) --> [was].
tv(X,Y,love(X,Y)) --> [loved].
iv(X,die(X)) --> [kicked,the,bucket].
iv(X,die(X)) --> [died].
```

% Deutsche Grammatik:

```
satz(P) --> s(finit,P).

s(Form,P) --> np_d(X,P1,P),vp(Form,X,P1).

np_d(X,P1,P) --> det_d(X,P2,P1,P), n1_d(X,P2).

n1_d(X,(P2&P1)) --> a_dn(X,P1), n1_d(X,P2).
n1_d(X,Q) --> n_d(X,P), rs(rel,X,P,Q).

vp(Form,X,P) --> aux(Form/Form2,X,Y,P1), np_d(Y,P1,P), vg(Form2,X,Y,P1).
vp(Form,X,P) --> v(Form,X,P).
vp(Form,X,P) -->  cop(Form/Form2,X,Y,P1), a_d(Y,P), cop(Form2/_,X,Y,P1).

vg(Form,X,Y,P1) --> v(Form2,X,Y,P1), aux(Form1,X,Y,P1), {p(Form,Form1,Form2)}.

rs(Form,X,P1,(P1&P2)) --> rp(P1), vp(Form,X,P2).
rs(Form,_,P,P) --> [].

det_d(X,P1,P2, exists(X,(P1&P2))) --> [eine].
det_d(X,P1,P2, all(X,(P1=>P2))) --> [jeder].

n_d(X,man(X)) --> [mann].
n_d(X,woman(X)) --> [frau].
n_d(X,(man(X) & unmarried(X))) --> [junggeselle].

rp(man(_)) --> [der].
rp((_ & man(_))) --> [der].
rp(woman(_)) --> [die].

a_dn(X,funny(X)) --> [lustige].
a_dn(X,die(X)) --> [tote].
a_dn(X,unmarried(X)) --> [unverheiratete].
a_d(X,unmarried(X)) --> [unverheiratet].
a_d(X,funny(X)) --> [lustig].

v(infinit,X,Y,love(X,Y)) --> [geliebt].
v(finit,X,Y,love(X,Y)) --> [liebte].
v([],_,_,_) --> [].
v(Form,X,die(X)) --> [starb], {(Form = finit; Form = rel)}.
v(finit,X,die(X)) --> [biss,ins,grass].
```

```
v(rel,X,die(X)) --> [ins,grass,biss].

aux(finit/[],X,Y,love(X,Y)) --> [liebte].
aux(finit/infinit,_,_,_) --> [hat].
aux(X/X,_,_,_) --> [], {(X = infinit ; X = [] ; X = rel)}.

cop(finit/rel,X,X,_) --> [war].
cop(rel/finit,_,_,_) --> [].

p(rel,finit/infinit, infinit).
p(rel,[]/[], finit).
p(X,[]/[], X):- X \== rel.
```

Modul Theorembeweiser

```
tabelle( P & Q , B, 1):-    tabelle(P, B, 1),  tabelle(Q, B, 1).
tabelle( P &  _ , B, 0):-   tabelle(P, B, 0).
tabelle( _ &  Q , B, 0):-   tabelle(Q, B, 0).

tabelle( P v Q , B, 0):-    tabelle(P, B, 0),  tabelle(Q, B, 1).
tabelle( P v _ , B, 1):-    tabelle(P, B, 1).
tabelle( _ v Q , B, 1):-    tabelle(Q, B, 1).

tabelle( P => Q , B, 0):-   tabelle(P, B, 1), tabelle(Q, B, 0).
tabelle( P => _ , B, 1):-   tabelle(P, B, 0).
tabelle( _ => Q , B, 1):-   tabelle(Q, B, 1).

tabelle( P <-> Q , B, 1):-  tabelle(P, B, 1),  tabelle(Q, B, 1).
tabelle( P <-> Q , B, 1):-  tabelle(P, B, 0),  tabelle(Q, B, 0).
tabelle( P <-> Q , B, 0):-  tabelle(P, B, 1),  tabelle(Q, B, 0).
tabelle( P <-> Q , B, 0):-  tabelle(P, B, 0),  tabelle(Q, B, 1).

tabelle( ¬ F , B, 0):-      tabelle(F, B, 1).
tabelle( ¬ F , B, 1):-      tabelle(F, B, 0).

tabelle( F , B, Wert):-     atom(F),   member([F,Wert],B),   !.


vars( Ausdruck , Variablen) :-
    Ausdruck =.. [_, Formel1, Formel2 ],
    !,
    vars( Formel1, Var1),
    vars( Formel2, Var2),
    vereinigung(Var1, Var2, Variablen).

vars( ¬ Formel , Variable) :-
    !,
    vars( Formel, Variable).

vars(V, [V] ).

belegung([],[]).
belegung([V|Rest], [[V,0]|Reste]):- belegung(Rest,Reste).
belegung([V|Rest], [[V,1]|Reste]):- belegung(Rest,Reste).
```

```
vereinigung([],Verein,Verein).
vereinigung([Kopf|Rest], Vergleichsmenge, [Kopf|Verein]):-
       delete(Kopf, Vergleichsmenge, V2), !,
        vereinigung(Rest, V2, Verein).
vereinigung([Kopf|Rest], Vergleichsmenge, [Kopf|Verein]):-
        vereinigung(Rest, Vergleichsmenge, Verein).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|Z1]):- dif(X,Y), delete(X,Z,Z1).

theorem_wahrheitswert(Formel):-
   vars(Formel,Variablen),
   belegung(Variablen, Wahrheitswerte),
   tabelle( Formel, Wahrheitswerte, 0),
   !,
   fail.
theorem_wahrheitswert(_).

theorem_tableau(Formel):-
   vars(Formel,Variablen),
   tabelle( Formel, Wahrheitswerte, 0),
   belegung(Variablen, Wahrheitswerte),
   !,
   fail.
theorem_tableau(_).

theorem_tableau(Formel):-
        vars(Formel,Variablen,VariablenWahrheitswerte),
        alle_verschieden(VariablenWahrheitswerte),
       tabelle( Formel, VariablenWahrheitswerte, 0),
     !,
      fail.

not_member(_,[]).
not_member(X,[[H,_]|T]):-
      dif(X,H),
       not_member(X,T).

alle_verschieden([]).
alle_verschieden([[X,_]|T]):-
      not_member(X,T),
       alle_verschieden(T).
```

Modul Diskurs

```
:- op(1200,xfy, ==>).

d(In, Out, [DrsS | DrsD]) -->
      s((In, OutS, DrsS, _,_,_,_,_)),
      d(OutS, Out, DrsD).
d(X, X, []) --> [].

s(Sem) -->
      {Sem =  (_,_,_, Index, _, SemVP, _,_),
       SemVP = (_,_,_,_,_,_,Index,_)},
      np(Sem),
      vp(SemVP).

vp(Sem) -->
      {Sem =  (_,_,_, Index,_, SemV, A1,_),
       SemV = (_,_,_,_,_,_,A1,Index)},
      v(SemV),
      np(Sem).

vp_rs(Sem) -->{
      Sem =  (_,_,_, Index,_, SemV, A1,_),
       SemV = (_,_,_,_,_,_,A1,Index)},
      np(Sem),
      v(SemV).

np(Sem) -->
      {Sem = (In,_,_, Index,SemN, Scope, _,_),
       SemN = (In,OutN,_,Index,_,_,_,_),
       Scope = ( OutN,_,_,_,_,_,_,_)},
      det(Sem),
      n(SemN).

np(Sem) -->
      {Sem = (In,Out,Drs, Index,_, Scope, _,_),
       Scope = ( In,Out,Drs,_,_,_,_,_)},
      pro(Index),
       {member(Index,In)}.

det(Sem) --> [jeder],
      {Sem = (In,In,Drs, _,Res, Scope, _,_),
       Res = (_,_,Resdrs,_,_,_,_,_),
       Scope = (_,_,Scopedrs,_,_,_,_,_),
      Drs = (Resdrs ==> Scopedrs)}.

det(Sem) --> ([ein];[eine];[einen]),
      {Sem = (_,Out,Drs, _,Res, Scope, _,_),
       Res = (_,_,Resdrs,_,_,_,_,_),
      Scope = (_,Out,Scopedrs,_,_,_,_,_),
      Drs = [Resdrs,Scopedrs]}.

n((In,  [Index | In],Drs,Index,_,_,_,_)) -->
       nomen(Index,Drs).

n(Sem) -->
       {Sem = (In, Out,[Drs,DrsVp],Index,_,_,_,_),
       SemVp = ([Index | In],Out,DrsVp,_,_,_,Index,_)},
       nomen(Index,Drs),
      relpro(Index),
      vp_rs(SemVp).

relpro(x( _, masc)) --> [der].
relpro(x( _, fem)) --> [die].

v((In,In,[umkreisen(A1,A2)],_,_,_,x(A1,_),x(A2,_))
) --> [umkreist].
v((In,In,[entdecken(A1,A2)],_,_,_,x(A1,_),x(A2,_)))
--> [entdeckt].

pro(x( _, masc)) --> [er].
pro(x( _, fem)) --> [sie].

nomen(x(X, masc),[X,astronom(X)]) --> [astronom].
nomen(x(X, fem), [X,sonne(X)]) --> [sonne].
nomen(x(X, masc), [X,planet(X)]) -->
[planet];[planeten].
```

Die Kompilierte Grammatik

```
s(A, [H|I], s(F,G), B, C) :-
      np(A, H, D, F, B, E),
      vp(A, [H|I], D, G, E, C).
s(finit, _, s(F,G), A, B) :-
       aux(finit/C, D, F, A, E),
      s(C, D, G, E, B).

vp(A, B, C, vp(H,I), D, E) :-
      vp2(A, B, F, H, D, G),
       nachfeld([C|F], I, G, E).

vp1(A, B, C, vp1(G,H), D, E) :-
       np_oder_ap(A, B, C, G, D, F),
      vg(A, B, H, F, E).

vp2(A, B, C, vp2(H,I), D, E) :-
       aux(A/F, B, H, D, G),
      vp1(F, B, C, I, G, E).

vg(A, B, vg(H,I), C, D) :-
      v(F, B, H, C, G),
      aux(E, B, I, G, D),
      p(A, E, F).

nachfeld([], nachfeld([]), A, A).
nachfeld([F|G], nachfeld(D,E), A, B) :-
       extraposition(F, D, A, C),
       nachfeld(G, E, C, B).

extraposition([], extraposition([]), A, A).
extraposition(rs(D), extraposition(C), A, B) :-
      rs(rel, D, _, C, A, B).
extraposition(als_np(F,E,D),extraposition(C),
A, B) :-
       als_np(F, E, D, C, A, B).



np(_, A, [], np(D), B, C) :-
      en(A, D, B, C).

np(A, B, C, np(G,H), D, E) :-
      det(A, B, G, D, F),
      n1(B, C, H, F, E).
np_oder_ap(A, [_|F], B, np_oder_ap(E), C, D) :-
      nprec(A, F, B, E, C, D).
np_oder_ap(A, B, C, np_oder_ap(F), D, E) :-
      ap(A, B, C, F, D, E).

nprec(_, [], [], nprec([]), A, A).
nprec(A, [I|J], [G|H], nprec(E,F), B, C) :-
      np(A, I, G, E, B, D),
       nprec(A, J, H, F, D, C).

n1(A, B, n1(F,G), C, D) :-
      n(A, F, C, E),
       radj(A, B, G, E, D).

radj(A, B, radj(E), C, D) :-
       rs(rel, A, B, E, C, D).
radj(_, E, radj(D), B, C) :-
      pp(E, D, B, C).

pp(A, pp(G,H), B, C) :-
      prep(D, E, G, B, F),
      np(E, D, A, H, F, C).

rs(A, [_|B], [], rs(G,H), C, D) :-
      rp([K|B], G, C, F),
      vp(A, [[K|B]|_], [], H, F, D).
rs(_, A, rs(A), rs([]), B, B).

als_np(A, B, [], als_np([als,A,B,[]],E), [als|D], C) :-
      np(A, B, _, E, D, C).
als_np(A, B, [als_np(A,B,[])], als_np([]), C, C).

ap(A, [I,H], B, ap(F,G), C, D) :-
       komparativ([I,H], F, C, E),
                als_np(A, H, B, G, E, D).
```

Das Kompilierte Lexikon

```
v(infinit, A, v([entdeckt,infinit,A]), [entdeckt|B], B) :-
      subkat(entdecken, _, A).
v(finit, A, v([entdeckten,finit,A]), [entdeckten|B], B) :-
      subkat(entdecken, plu, A).
v(finit, A, v([entdeckte,finit,A]), [entdeckte|B], B) :-
      subkat(entdecken, sing, A).
v(finit, A, v([umkreist,finit,A]), [umkreist|B], B) :-
      subkat(umkreisen, sing, A).
v(finit, A, v([umkreisen,finit,A]), [umkreisen|B], B) :-
      subkat(umkreisen, plu, A).
v(finit, A, v([besitzt,finit,A]), [besitzt|B], B) :-
      subkat(besitzen, sing, A).
v(finit, A, v([besitzen,finit,A]), [besitzen|B], B) :-
      subkat(besitzen, plu, A).
v(finit, A, v([sind,finit,A]), [sind|B], B) :-
      subkat(sein, plu, A).
v(finit, A, v([ist,finit,A]), [ist|B], B) :-
      subkat(sein, sing, A).
v(finit, A, v([gibt,finit,A]), [gibt|B], B) :-
      subkat(geben, sing, A).
v(infinit, A, v([gegeben,infinit,A]), [gegeben|B], B) :-
      subkat(es_geben, _, A).
v(finit, A, v([gibt,finit,A]), [gibt|B], B) :-
      subkat(es_geben, sing, A).
v([], _, v([]), A, A).

n([nom,sing,masc], n([astronom,[nom,sing,masc]]), [astronom|A], A).
n([akk,sing,masc], n([astronomen,[akk,sing,masc]]), [astronomen|A], A).
n([dat,sing,masc], n([astronomen,[dat,sing,masc]]), [astronomen|A], A).
n([nom,plu,masc], n([astronomen,[nom,plu,masc]]), [astronomen|A], A).
n([akk,plu,masc], n([astronomen,[akk,plu,masc]]), [astronomen|A], A).
n([dat,plu,masc], n([astronomen,[dat,plu,masc]]), [astronomen|A], A).
n([dat,sing,masc], n([kilometer,[dat,sing,masc]]), [kilometer|A], A).
n([dat,plu,masc], n([kilometern,[dat,plu,masc]]), [kilometern|A], A).
n([nom,sing,masc], n([kilometer,[nom,sing,masc]]), [kilometer|A], A).
n([nom,plu,masc], n([kilometer,[nom,plu,masc]]), [kilometer|A], A).
n([nom,sing,fem], n([sonne,[nom,sing,fem]]), [sonne|A], A).
n([nom,plu,fem], n([sonnen,[nom,plu,fem]]), [sonnen|A], A).
n([akk,sing,fem], n([sonne,[akk,sing,fem]]), [sonne|A], A).
n([akk,plu,fem], n([sonnen,[akk,plu,fem]]), [sonnen|A], A).
```

```
n([dat,sing,fem], n([sonne,[dat,sing,fem]]), [sonne|A], A).
n([dat,plu,fem], n([sonnen,[dat,plu,fem]]), [sonnen|A], A).
n([nom,sing,fem], n([erde,[nom,sing,fem]]), [erde|A], A).
n([akk,sing,fem], n([erde,[akk,sing,fem]]), [erde|A], A).
n([dat,sing,fem], n([erde,[dat,sing,fem]]), [erde|A], A).
n([nom,sing,masc], n([planet,[nom,sing,masc]]), [planet|A], A).
n([nom,plu,masc], n([planeten,[nom,plu,masc]]), [planeten|A], A).
n([akk,sing,masc], n([planeten,[akk,sing,masc]]), [planeten|A], A).
n([akk,plu,masc], n([planeten,[akk,plu,masc]]), [planeten|A], A).
n([dat,sing,masc], n([planeten,[dat,sing,masc]]), [planeten|A], A).
n([dat,plu,masc], n([planeten,[dat,plu,masc]]), [planeten|A], A).
n([nom,sing,masc], n([mond,[nom,sing,masc]]), [mond|A], A).
n([nom,plu,masc], n([monde,[nom,plu,masc]]), [monde|A], A).
n([akk,sing,masc], n([mond,[akk,sing,masc]]), [mond|A], A).
n([akk,plu,masc], n([monde,[akk,plu,masc]]), [monde|A], A).
n([dat,sing,masc], n([mond,[dat,sing,masc]]), [mond|A], A).
n([dat,plu,masc], n([monden,[dat,plu,masc]]), [monden|A], A).
n([nom,sing,masc], n([himmelskoerper,[nom,sing,masc]]), [himmelskoerper|A], A).
n([nom,plu,masc], n([himmelskoerper,[nom,plu,masc]]), [himmelskoerper|A], A).
n([akk,sing,masc], n([himmelskoerper,[akk,sing,masc]]), [himmelskoerper|A], A).
n([akk,plu,masc], n([himmelskoerper,[akk,plu,masc]]), [himmelskoerper|A], A).
n([dat,sing,masc], n([himmelskoerper,[dat,sing,masc]]), [himmelskoerper|A], A).
n([dat,plu,masc], n([himmelskoerpern,[dat,plu,masc]]), [himmelskoerpern|A], A).
n([nom,sing,masc], n([durchmesser,[nom,sing,masc]]), [durchmesser|A], A).
n([nom,plu,masc], n([durchmesser,[nom,plu,masc]]), [durchmesser|A], A).
n([akk,sing,masc], n([durchmesser,[akk,sing,masc]]), [durchmesser|A], A).
n([akk,plu,masc], n([durchmessern,[akk,plu,masc]]), [durchmessern|A], A).
n([dat,sing,masc], n([durchmesser,[dat,sing,masc]]), [durchmesser|A], A).
n([dat,plu,masc], n([durchmessern,[dat,plu,masc]]), [durchmessern|A], A).
n([nom,sing,fem], n([entdeckung,[nom,sing,fem]]), [entdeckung|A], A).
n([nom,plu,fem], n([entdeckungen,[nom,plu,fem]]), [entdeckungen|A], A).
n([akk,sing,fem], n([entdeckung,[akk,sing,fem]]), [entdeckung|A], A).
n([akk,plu,fem], n([entdeckungen,[akk,plu,fem]]), [entdeckungen|A], A).
n([dat,sing,fem], n([entdeckung,[dat,sing,fem]]), [entdeckung|A], A).
n([dat,plu,fem], n([entdeckungen,[dat,plu,fem]]), [entdeckungen|A], A).

aux(finit/[], A, aux(D), B, C) :-
    v(finit, A, D, B, C).
aux(finit/infinit, A, aux([hat,finit/infinit,A]), [hat|B], B).
aux(finit/infinit, A, aux([haben,finit/infinit,A]), [haben|B], B) :-
    subkat(haben, plu, A).
aux(B/B, _, aux([]), A, A) :-
```

```
    (  B=infinit;  B=[] ;  B=rel ).

det(finit, A, det(D), B, C) :-
    dets(_, A, D, B, C).
det(A, B, det(F), C, D) :-
    dets(E, B, F, C, D),
    (  A=infinit;  A=[];  A=rel),
    (  E=def ;  E=indef;  E=all).
det(relpro, [B,sing,masc], det([dessen,relpro,[B,sing,masc]]), [dessen|A], A).
det(relpro, [B,sing,neu], det([dessen,relpro,[B,sing,neu]]), [dessen|A], A).
det(relpro, [B,sing,fem], det([deren,relpro,[B,sing,fem]]), [deren|A], A).
det(relpro, [B,plu,masc], det([deren,relpro,[B,plu,masc]]), [deren|A], A).
det(relpro, [B,plu,fem], det([deren,relpro,[B,plu,fem]]), [deren|A], A).
det(relpro, [B,plu,neu], det([deren,relpro,[B,plu,neu]]), [deren|A], A).

dets(all, [nom,sing,masc], dets([jeder,all,[nom,sing,masc]]), [jeder|A], A).
dets(all, [akk,sing,masc], dets([jeden,all,[akk,sing,masc]]), [jeden|A], A).
dets(def, [nom,sing,masc], dets([der,def,[nom,sing,masc]]), [der|A], A).
dets(def, [dat,sing,fem], dets([der,def,[dat,sing,fem]]), [der|A], A).
dets(def, [akk,sing,masc], dets([den,def,[akk,sing,masc]]), [den|A], A).
dets(indef, [nom,sing,masc], dets([ein,indef,[nom,sing,masc]]), [ein|A], A).
dets(indef, [akk,sing,masc], dets([einen,indef,[akk,sing,masc]]), [einen|A], A).
dets(indef, [dat,sing,masc], dets([einem,indef,[dat,sing,masc]]), [einem|A], A).
dets(indef, [nom,sing,fem], dets([eine,indef,[nom,sing,fem]]), [eine|A], A).
dets(indef, [akk,sing,fem], dets([eine,indef,[akk,sing,fem]]), [eine|A], A).
dets(indef, [dat,sing,fem], dets([einer,indef,[dat,sing,fem]]), [einer|A], A).
dets(frage, [nom,sing,masc], dets([welcher,frage,[nom,sing,masc]]), [welcher|A], A).
dets(frage, [akk,sing,masc], dets([welchen,frage,[akk,sing,masc]]), [welchen|A], A).
dets(def, [nom,sing,fem], dets([die,def,[nom,sing,fem]]), [die|A], A).
dets(def, [akk,sing,fem], dets([die,def,[akk,sing,fem]]), [die|A], A).
dets(def, [nom,plu,fem], dets([die,def,[nom,plu,fem]]), [die|A], A).
dets(def, [akk,plu,fem], dets([die,def,[akk,plu,fem]]), [die|A], A).
dets(def, [nom,plu,masc], dets([die,def,[nom,plu,masc]]), [die|A], A).
dets(def, [akk,plu,masc], dets([die,def,[akk,plu,masc]]), [die|A], A).
dets(frage, [nom,plu,masc], dets([wieviele,frage,[nom,plu,masc]]), [wieviele|A], A).
dets(frage, [nom,plu,masc], dets([welche,frage,[nom,plu,masc]]), [welche|A], A).
dets(frage, [akk,plu,masc], dets([welche,frage,[akk,plu,masc]]), [welche|A], A).
dets(def, [dat,plu,C], dets([B,def,[dat,plu,C]]), [B|A], A) :-
    integer(B),
    B>1.
dets(def, [nom,plu,C], dets([B,def,[nom,plu,C]]), [B|A], A) :-
    integer(B),
```

```
    B>1.

komparativ(A, komparativ([groesser,A]), [groesser|B], B) :-
      subkat(groesser_als, _, A).
komparativ(A, komparativ([kleiner,A]), [kleiner|B], B) :-
      subkat(groesser_als, _, A).

rp([nom,sing,masc], rp([der,[nom,sing,masc]]), [der|A], A).
rp( [akk,sing,masc], rp([den,[akk,sing,masc]]), [den|A], A).
rp( [D|E], rp(C), A, B) :-
      np(relpro, [D|E], _, C, A, B).
rp( [nom,plu,masc], rp([die,[nom,plu,masc]]), [die|A], A).
rp([akk,plu,masc], rp([die,[akk,plu,masc]]), [die|A], A).
rp([nom,sing,fem], rp([die,[nom,sing,fem]]), [die|A], A).
rp([akk,sing,fem], rp([die,[akk,sing,fem]]), [die|A], A).
rp([nom,plu,fem], rp([die,[nom,plu,fem]]), [die|A], A).
rp( [akk,plu,fem], rp([die,[akk,plu,fem]]), [die|A], A).

prep([dat,C,B], [], prep([von,[dat,C,B],[]]), [von|A], A).

en([B,sing,masc], en([herschel,[B,sing,masc]]), [herschel|A], A) :-
     ( B=nom
     ; B=akk
     ; B=dat
     ).
en([nom,sing_], en([es,[nom,sing_]), [es|A], A).
en([C,sing,masc], en([B,[C,sing,masc]]), [B|A], A) :-
      member(B,
[jupiter,mars,merkur,neptun,pluto,saturn,uranus,venus,adrastea,amalthea,ananke,ariel,calli
sto,carme,charon,diana,deimos,elara,enceladus,europa,ganymed,himalia,hyperion,iapetus,io,
leda,lysithea,mimas,miranda,nereide,oberon,pasiphae,phobos,phoebe,rhea,sinope,tethys,titan,ti
tania,triton,umbriel,'1979J2','1979J3','1980S1','1980S3','1980S6','1980S13','1980S25','1980S26','1
980S27','1980S28',bond,cassini,dollfus,fountain,galilei,hall,huyghens,kuiper,lacques,lassell,m
elotte,nicholson,perrine,pickering,smith,tombaugh]),
     ( C=nom
     ; C=akk
     ; C=dat
     ).
```

Modul Tiefenstruktur

```
s(A, [H|I], s(F,G), B, C) :-
        np(A, H, D, F, B, E),
        vp(A, [H|I], D, G, E, C).
s(finit, _, s(Np,s(F,Vp)), A, B) :-
        aux(finit/C, D, F, A, E),
        s(C, D, s(Np,Vp), E, B).

rs(A, [_|B], [], rs(G,H), C, D) :-
        rp([K|B], G, C, F),
        vp(A, [[K|B]|_], [], H, F, D).
rs(rel, A, rs(X,Y,A), rs(X,Y), B, B).

als_np(A, B, [], als_np([als,A,B,[]],E), [als|D], C) :-
        np(A, B, Q, E, D, C),
        (Q = []; Q =.. [_,[],[],_]).
als_np(A, B, [als_np([als,A,B,[]],E)], als_np([als,A,B,[]],E), C, C).

nachfeld([], nachfeld([]), B, B).
nachfeld([H|T], nachfeld([]), B, C) :-
        extraposition(H, B, D),
        nachfeld(T, nachfeld([]), D, C).

extraposition([], B, B).
extraposition(rs(X,Y,Z), B, C):-
        rs(rel, Z, [], rs(X,Y), B, C).
extraposition(rs([],[],_), B, B).
extraposition(als_np([als,X,Y,[]],Z), B, C):-
        als_np(X, Y, [], als_np([als,X,Y,[]],Z), B, C).
```

Modul Initialisierung

```
:- ['Modul Benutzerführung'].
:- ['Modul Compiler'].
:- ['Modul Syntax'].
:- ['Modul Lexikon'].
:- ['Modul Semantik'].
:- ['Modul Schnittstellen'].
:- ['Modul Datenbank'].
:- ['Modul Evaluierung-Variante4'].
:- ['Modul Pragmatik'].
:- ['Modul Skopus SynBaum'].

:- compile.

:- ['Modul Tiefenstruktur'].

:- go.
```

Beispiele

Datenbankabfrage

Das System beantwortet einen Eingabesatz mit bis zu vier verschiedenen Antworttypen. Typ 1 besteht in der syntaktischen Repräsentation, die ausgegeben wird, wenn ein Satz syntaktisch erkannt wurde. Dies ist zum Beispiel auch dann der Fall, wenn es sich um Sätze wie "entdeckte ein astronom einen astronomen" handelt, die keine semantische Repräsentation erhalten. Die semantische Repräsentation, die dem zweiten Antworttyp entspricht, besteht aus der logischen Formel. Der dritte Antworttyp stellt die optimierte Fassung der logischen Repräsentation dar, die schließlich im Modell ausgewertet wird. Die Ergebnisse dieser Berechnung erscheinen als vierter Antworttyp.

Der Aufruf des Systems erfolgt mit "go":

```
?- go.
Beenden der Interaktion mit "ende."
```

Der Abbruch der Interaktion wird mit "ende" angezeigt:

```
>>  ende.
  yes
```

>> gibt es einen himmelskoerper den jeder planet umkreist ?

SYNTAKTISCHE REPRAESENTATION:

s(np(en([es,[nom,sing,_2524]])),s(aux(v([gibt,finit,[[nom,sing,_2524],[akk,sing,masc]]])),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([himmelskoerper,[akk,sing,masc]]),radj(rs(rp([den,[akk,sing,masc]]),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([jeder,all,[nom,sing,masc]])),n1(n([planet,[nom,sing,masc]]),radj(rs([],[])))),nprec([])),vg(v([umkreist,finit,[[akk,sing,masc],[nom,sing,masc]]]),aux([])))),nachfeld([])))))),nprec([])),vg(v([]),aux([])))),nachfeld([]))))

SEMANTISCHE REPRAESENTATION:

```
existiert(_4305,
        himmelskoerper(_4305)
        &
         fuer_alle(_5488,
               planet(_5488)
              =>
                 umkreisen((_5488,planet_),
                      (_4305,_4912)))
       &
        es_gibt(_4342,
              (_4305,himmelskoerper_(_4912))))
```

EVALUIERUNG IM MODELL:

```
es_gibt(_4342,
        (_4305,himmelskoerper_(_4912)))
&
¬planet(_5488)
  &
  ¬umkreisen((_5488,planet_),
         (_4305,_4912))
```

RESULTATE:

ja

>> gibt es einen planeten den jeder mond umkreist ?

SYNTAKTISCHE REPRAESENTATION:

s(np(en([es,[nom,sing,_2388]])),s(aux(v([gibt,finit,[[nom,sing,_2388],[akk,sing,masc]]])),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([planeten,[akk,sing,masc]]),radj(rs(rp([den,[akk,sing,masc]]),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([jeder,all,[nom,sing,masc]])),n1(n([mond,[nom,sing,masc]]),radj(rs([],[])))),nprec([]))),vg(v([umkreist,finit,[[akk,sing,masc],[nom,sing,masc]]]),aux([])))),nachfeld([])))))),nprec([]))),vg(v([]),aux([]))),nachfeld([])))

SEMANTISCHE REPRAESENTATION:

```
existiert(_4169,
        planet(_4169)
        &
          fuer_alle(_5352,
                mond(_5352)
              =>
                 umkreisen((_5352,mond_),
                      (_4169,planet_)))
       &
        es_gibt(_4206,
              (_4169,himmelskoerper_(planet_))))
```

EVALUIERUNG IM MODELL:

```
es_gibt(_4206,
         (_4169,himmelskoerper_(planet_)))
&
¬mond(_5352)
  &
   ¬umkreisen((_5352,mond_),
          (_4169,planet_))
```

RESULTATE:

nein

>> **entdeckte jeder astronom einen planeten ?**

SYNTAKTISCHE REPRAESENTATION:

s(np(det(dets([jeder,all,[nom,sing,masc]])),n1(n([astronom,[nom,sing,masc]]),radj(rs([],[])))),s(aux(v([entdeckte,finit,[[nom,sing,masc],[akk,sing,masc]]])),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([planeten,[akk,sing,masc]]),radj(rs([],[])))),nprec([]))),vg(v([]),aux([])))),nachfeld([]))))

SEMANTISCHE REPRAESENTATION:

```
fuer_alle(_3255,
        astronom(_3255)
       =>
        existiert(_3805,
              planet(_3805)
             &
               entdecken((_3255,astronom_),
                    (_3805,planet_))))
```

EVALUIERUNG IM MODELL:

```
¬astronom(_3255)
 &
  ¬entdecken((_3255,astronom_),
         (_3805,planet_))
```

RESULTATE:

nein

>> **entdeckte jeder astronom einen himmelskoerper ?**

SYNTAKTISCHE REPRAESENTATION:

s(np(det(dets([jeder,all,[nom,sing,masc]])),n1(n([astronom,[nom,sing,masc]]),radj(rs([],[])))),s(aux(v([entdeckte,finit,[[nom,sing,masc],[akk,sing,masc]]])),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([himmelskoerper,[akk,sing,masc]]),radj(rs([],[])))),nprec([]))),vg(v([]),aux([])))),nachfeld([]))))

SEMANTISCHE REPRAESENTATION:

```
fuer_alle(_3357,
       astronom(_3357)
      =>
       existiert(_3907,
             himmelskoerper(_3907)
           &
             entdecken((_3357,astronom_),
                 (_3907,_3958))))
```

EVALUIERUNG IM MODELL:

```
¬astronom(_3357)
 &
 ¬entdecken((_3357,astronom_),
        (_3907,_3958))
```

RESULTATE:

ja

>> **entdeckte jeder astronom der einen mond entdeckt hat einen himmelskoerper ?**

SYNTAKTISCHE REPRAESENTATION:

s(np(det(dets([jeder,all,[nom,sing,masc]])),n1(n([astronom,[nom,sing,masc]]),radj(rs(rp([der,[nom,sing,masc]]),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([mond,[akk,sing,masc]]),radj(rs([],[])))),nprec([]))),vg(v([entdeckt,infinit,[[nom,sing,masc],[akk,sing,masc]]]),aux([hat,finit/infinit,[[nom,sing,masc],[akk,sing,masc]]])))),nachfeld([])))))),s(aux(v([entdeckte,finit,[[nom,sing,masc],[akk,sing,masc]]])),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([himmelskoerper,[akk,sing,masc]]),radj(rs([],[])))),nprec([]))),vg(v([]),aux([])))),nachfeld([]))))

SEMANTISCHE REPRAESENTATION:

```
fuer_alle(_4807,
        astronom(_4807)
        &
          existiert(_5553,
                mond(_5553)
              &
                 entdecken((_4807,astronom_),
                      (_5553,mond_)))
       =>
        existiert(_6754,
              himmelskoerper(_6754)
            &
              entdecken((_4807,astronom_),
                  (_6754,_6805))))
```

EVALUIERUNG IM MODELL:

```
¬entdecken((_4807,astronom_),
          (_5553,mond_))
 &
  ¬entdecken((_4807,astronom_),
          (_6754,_6805))
```

RESULTATE:

ja

>> **welcher astronom der einen mond entdeckte hat einen planeten entdeckt ?**

SYNTAKTISCHE REPRAESENTATION:

s(np(det(dets([welcher,frage,[nom,sing,masc]])),n1(n([astronom,[nom,sing,masc]]),radj(rs(rp([der,[nom,sing,masc]]),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([mond,[akk,sing,masc]]),radj(rs([],[])))),nprec([]))),vg(v([entdeckte,finit,[[nom,sing,masc],[akk,sing,masc]]]),aux([])))),nachfeld([])))))),vp(vp2(aux([hat,finit/infinit,[[nom,sing,masc],[akk,sing,masc]]]),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([planeten,[akk,sing,masc]]),radj(rs([],[])))),nprec([]))),vg(v([entdeckt,infinit,[[nom,sing,masc],[akk,sing,masc]]]),aux([])))),nachfeld([])))

SEMANTISCHE REPRAESENTATION:

```
existiert(_4626,
        astronom(_4626)
        &
         existiert(_5372,
                mond(_5372)
               &
                  entdecken((_4626,astronom_),
                       (_5372,mond_)))
       &
        existiert(_6824,
               planet(_6824)
              &
                 entdecken((_4626,astronom_),
                      (_6824,planet_))))
```

EVALUIERUNG IM MODELL:

```
entdecken((_2673,astronom_),
         (_5372,mond_))
&
entdecken((_2673,astronom_),
        (_6824,planet_))
```

RESULTATE:

herschel

>> welche planeten besitzen einen durchmesser der kleiner als ein durchmesser von einem mond ist ?

SYNTAKTISCHE REPRAESENTATION:

s(np(det(dets([welche,frage,[nom,plu,masc]])),n1(n([planeten,[nom,plu,masc]]),radj(rs([],[])))),vp(vp2(aux(v([besitzen,finit,[[nom,plu,masc],[akk,sing,masc]]])),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([durchmesser,[akk,sing,masc]]),radj(rs(rp([der,[nom,sing,masc]]),vp(vp2(aux([]),vp1(np_oder_ap(ap(komparativ([kleiner,[[nom,sing,masc],[nom,sing,masc]]])),als_np([als,rel,[nom,sing,masc],[]],np(det(dets([ein,indef,[nom,sing,masc]])),n1(n([durchmesser,[nom,sing,masc]]),radj(pp(prep([von,[dat,sing,masc],[]]),np(det(dets([einem,indef,[dat,sing,masc]])),n1(n([mond,[dat,sing,masc]]),radj(rs([],[])))))))))),vg(v([ist,finit,[[nom,sing,masc],[nom,sing,masc]]]),aux([])))),nachfeld([])))))),nprec([]))),vg(v([]),aux([]))),nachfeld([])))

SEMANTISCHE REPRAESENTATION:

```
existiert(_5523,planet(_5523)
      &
       existiert(_6122,
             durchmesser(_6122)
            &
               existiert(_7144,durchmesser(_7144)
                   &
                      existiert(_7806,mond(_7806)
                         &
                               besitzen((_7806,mond_),(_7144,durchmesser_)))
                  &
                     kleiner(_6122,_7144))
           &
              besitzen((_5523,planet_),(_6122,durchmesser_))))
```

EVALUIERUNG IM MODELL:

```
besitzen((_7806,mond_),(_7144,durchmesser_))
 &
 kleiner(_6122,_7144)
&
besitzen((_3265,planet_),(_6122,durchmesser_))
```

RESULTATE:

merkur pluto

>> welche monde besitzen einen durchmesser der groesser als ein durchmesser von einem planeten ist ?

SYNTAKTISCHE REPRAESENTATION:

s(np(det(dets([welche,frage,[nom,plu,masc]])),n1(n([monde,[nom,plu,masc]]),radj(rs([],[])))),vp(vp2(aux(v([besitzen,finit,[[nom,plu,masc],[akk,sing,masc]]])),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([durchmesser,[akk,sing,masc]]),radj(rs(rp([der,[nom,sing,masc]]),vp(vp2(aux([]),vp1(np_oder_ap(ap(komparativ([groesser,[[nom,sing,masc],[nom,sing,masc]]])),als_np([als,rel,[nom,sing,masc],[]],np(det(dets([ein,indef,[nom,sing,masc]])),n1(n([durchmesser,[nom,sing,masc]]),radj(pp(prep([von,[dat,sing,masc],[]]),np(det(dets([einem,indef,[dat,sing,masc]])),n1(n([planeten,[dat,sing,masc]]),radj(rs([],[])))))))))),vg(v([ist,finit,[[nom,sing,masc],[nom,sing,masc]]]),aux([])))),nachfeld([])))))),nprec([]))),vg(v([]),aux([]))),nachfeld([])))

SEMANTISCHE REPRAESENTATION:

```
existiert(_5557,mond(_5557)
    &
     existiert(_6156,durchmesser(_6156)
         &
            existiert(_7178,durchmesser(_7178)
               &
                  existiert(_7840,planet(_7840)
                     &
                           besitzen((_7840,planet_),(_7178,durchmesser_)))
               &
                  groesser(_6156,_7178))
        &
           besitzen((_5557,mond_),(_6156,durchmesser_))))
```

EVALUIERUNG IM MODELL:

```
besitzen((_7840,planet_),(_7178,durchmesser_))
 &
 groesser(_6156,_7178)
&
besitzen((_3299,mond_),(_6156,durchmesser_))
```

RESULTATE:

callisto europa ganymed io mond titan triton

>> welche monde die kuiper entdeckte umkreisen einen planeten den herschel entdeckte ?

SYNTAKTISCHE REPRAESENTATION:

s(np(det(dets([welche,frage,[nom,plu,masc]])),n1(n([monde,[nom,plu,masc]]),radj(rs(rp([die,[akk,plu,masc]]),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(en([kuiper,[nom,sing,masc]])),nprec([])),vg(v([entdeckte,finit,[[akk,plu,masc],[nom,sing,masc]]]),aux([])))),nachfeld([])))))),vp(vp2(aux(v([umkreisen,finit,[[nom,plu,masc],[akk,sing,masc]]])),vp1(np_oder_ap(nprec(np(det(dets([einen,indef,[akk,sing,masc]])),n1(n([planeten,[akk,sing,masc]]),radj(rs(rp([den,[akk,sing,masc]]),vp(vp2(aux([]),vp1(np_oder_ap(nprec(np(en([herschel,[nom,sing,masc]])),nprec([]))),vg(v([entdeckte,finit,[[akk,sing,masc],[nom,sing,masc]]]),aux([])))),nachfeld([])))))),nprec([]))),vg(v([]),aux([])))),nachfeld([])))

SEMANTISCHE REPRAESENTATION:

```
existiert(_5984,
        mond(_5984)
        &
          entdecken((kuiper,astronom_),
                (_5984,mond_))
       &
        existiert(_7668,
              planet(_7668)
              &
                 entdecken((herschel,astronom_),
                      (_7668,planet_))
             &
                umkreisen((_5984,mond_),
                     (_7668,planet_))))
```

EVALUIERUNG IM MODELL:

```
entdecken((kuiper,astronom_),(_2923,mond_))
&
entdecken((herschel,astronom_),(_7668,planet_))
&
umkreisen((_2923,mond_),(_7668,planet_))
```

RESULTATE:

miranda nereide

Geben Sie einen Satz ein !
Beenden Sie ihn mit "." oder "?" und RETURN.
>> **welche monde gibt es ?**

SYNTAKTISCHE REPRAESENTATION:

s(np(det(dets([welche,frage,[akk,plu,masc]])),n1(n([monde,[akk,plu,masc]]),radj(rs([],[])))),vp(vp2(aux(v([gibt,finit,[[akk,plu,masc],[nom,sing,_1938]]]))),vp1(np_oder_ap(nprec(np(en([es,[nom,sing,_1938]])),nprec([]))),vg(v([]),aux([])))),nachfeld([]))

SEMANTISCHE REPRAESENTATION:

```
existiert(_2600,
        mond(_2600)
       &
        es_gibt(_3211,
              (_2600,himmelskoerper_(mond_))))
```

EVALUIERUNG IM MODELL:

```
es_gibt(_3211,
        (_1564,himmelskoerper_(mond_)))
```

RESULTATE:

1979J2 1979J3 1980S1 1980S13 1980S25 1980S26 1980S27 1980S28 1980S3 1980S6 adrastea amalthea ananke ariel callisto carme charon deimos diana elara enceladus europa ganymed himalia hyperion iapetus io leda lysithea mimas miranda mond nereide oberon pasiphae phobos phoebe rhea sinope tethys titan titania triton umbriel

Maschinelle Übersetzung

```
?- ed([every,man,that,kicked,the,bucket,loved,a,woman,that,was,funny],L).

L = [jeder,tote,mann,liebte,eine,lustige,frau] ? ;
L = [jeder,tote,mann,liebte,eine,frau,die,lustig,war] ? ;
L = [jeder,tote,mann,hat,eine,lustige,frau,geliebt] ? ;
L = [jeder,tote,mann,hat,eine,frau,die,lustig,war,geliebt] ? ;
L = [jeder,mann,der,starb,liebte,eine,lustige,frau] ? ;
L = [jeder,mann,der,starb,liebte,eine,frau,die,lustig,war] ? ;
L = [jeder,mann,der,starb,hat,eine,lustige,frau,geliebt] ? ;
L = [jeder,mann,der,starb,hat,eine,frau,die,lustig,war,geliebt] ? ;
L = [jeder,mann,der,ins,grass,biss,liebte,eine,lustige,frau] ? ;
L = [jeder,mann,der,ins,grass,biss,liebte,eine,frau,die,lustig,war] ? ;
L = [jeder,mann,der,ins,grass,biss,hat,eine,lustige,frau,geliebt] ? ;
L = [jeder,mann,der,ins,grass,biss,hat,eine,frau,die,lustig,war,geliebt] ? ;
no

 ?- ed([every,unmarried,man,loved,a,woman,that,was,funny],L).
L = [jeder,unverheiratete,mann,liebte,eine,lustige,frau] ? ;
L = [jeder,unverheiratete,mann,liebte,eine,frau,die,lustig,war] ? ;
L = [jeder,unverheiratete,mann,hat,eine,lustige,frau,geliebt] ? ;
L = [jeder,unverheiratete,mann,hat,eine,frau,die,lustig,war,geliebt] ? ;
L = [jeder,mann,der,unverheiratet,war,liebte,eine,lustige,frau] ? ;
L = [jeder,mann,der,unverheiratet,war,liebte,eine,frau,die,lustig,war] ? ;
L = [jeder,mann,der,unverheiratet,war,hat,eine,lustige,frau,geliebt] ? ;
L = [jeder,mann,der,unverheiratet,war,hat,eine,frau,die,lustig,war,geliebt] ? ;
L = [jeder,junggeselle,liebte,eine,lustige,frau] ? ;
L = [jeder,junggeselle,liebte,eine,frau,die,lustig,war] ? ;
L = [jeder,junggeselle,hat,eine,lustige,frau,geliebt] ? ;
L = [jeder,junggeselle,hat,eine,frau,die,lustig,war,geliebt] ? ;
no

?- ed([every,bachelor,loved,a,funny,woman],K).
K = [jeder,unverheiratete,mann,liebte,eine,lustige,frau]
K = [jeder,unverheiratete,mann,liebte,eine,frau,die,lustig,war]
K =  [jeder,unverheiratete,mann,hat,eine,lustige,frau,geliebt]
K = [jeder,unverheiratete,mann,hat,eine,frau,die,lustig,war,geliebt]
K = [jeder,mann,der,unverheiratet,war,liebte,eine,lustige,frau]
K = [jeder,mann,der,unverheiratet,war,liebte,eine,frau,die,lustig,war]
K = [jeder,mann,der,unverheiratet,war,hat,eine,lustige,frau,geliebt]
K = [jeder,mann,der,unverheiratet,war,hat,eine,frau,die,lustig,war,geliebt]
```

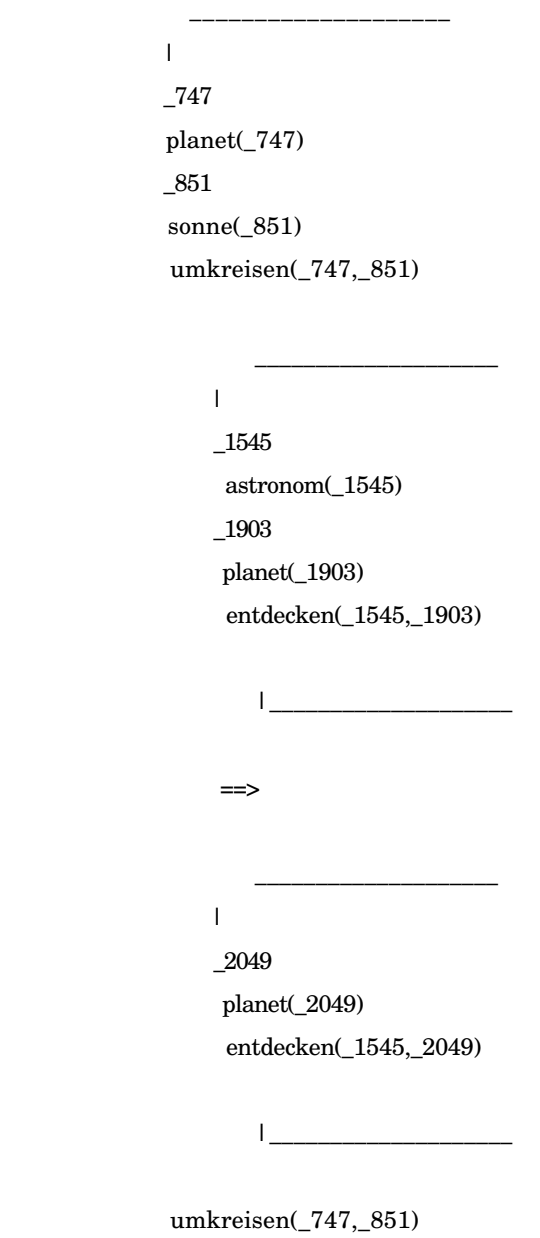
```
K = [jeder,junggeselle,liebte,eine,lustige,frau]
K = [jeder,junggeselle,liebte,eine,frau,die,lustig,war]
K = [jeder,junggeselle,hat,eine,lustige,frau,geliebt]
K = [jeder,junggeselle,hat,eine,frau,die,lustig,war,geliebt]

 ?- de([jeder,tote,junggeselle,hat,eine,lustige,frau,geliebt],L).
L = [every,bachelor,that,kicked,the,bucket,loved,a,funny,woman] ? ;
L = [every,bachelor,that,kicked,the,bucket,loved,a,woman,that,was,funny] ? ;
L = [every,bachelor,that,died,loved,a,funny,woman] ? ;
L = [every,bachelor,that,died,loved,a,woman,that,was,funny] ? ;
no
 ?- de([jeder,tote,unverheiratete,mann,hat,eine,lustige,frau,geliebt],L).

L = [every,bachelor,that,kicked,the,bucket,loved,a,funny,woman] ? ;
L = [every,bachelor,that,kicked,the,bucket,loved,a,woman,that,was,funny] ? ;
L = [every,bachelor,that,died,loved,a,funny,woman] ? ;
L = [every,bachelor,that,died,loved,a,woman,that,was,funny] ? ;
no
 ?- de([jeder,unverheiratete,mann,hat,eine,lustige,frau,geliebt],L).

L = [every,unmarried,man,loved,a,funny,woman] ? ;
L = [every,unmarried,man,loved,a,woman,that,was,funny] ? ;
L = [every,man,that,was,unmarried,loved,a,funny,woman] ? ;
L = [every,man,that,was,unmarried,loved,a,woman,that,was,funny] ? ;
L = [every,bachelor,loved,a,funny,woman] ? ;
L = [every,bachelor,loved,a,woman,that,was,funny] ? ;
no

 ?- de([jeder,lustige,unverheiratete,mann,hat,eine,frau,geliebt],L).
L = [every,funny,unmarried,man,loved,a,woman] ;
L = [every,funny,man,that,was,unmarried,loved,a,woman] ;
L = [every,funny,bachelor,loved,a,woman] ;
L = [every,bachelor,that,was,funny,loved,a,woman] ;
no
 ?- de([jeder,lustige,junggeselle,hat,eine,frau,geliebt],L).
L = [every,funny,unmarried,man,loved,a,woman] ;
L = [every,funny,man,that,was,unmarried,loved,a,woman] ;
L = [every,funny,bachelor,loved,a,woman] ;
L = [every,bachelor,that,was,funny,loved,a,woman] ;
no
```

Diskursanalyse

Kommentar:

Es ist nur eine Analyse möglich, da sich die Pronomen nur auf den ersten Satz beziehen können.

d([],_,DRS,[ein,planet,umkreist,eine,sonne,jeder,astronom,der,einen,planeten,entdeckt,entdeckt,einen,planeten,er,umkreist,sie],[]).



| ?- d([],_,DRS, [ein,planet,umkreist,eine,sonne,jeder,astronom,der,einen,planeten,entdeckt,entdeckt,einen,planeten,der,eine,sonne,die,jeder,planeten,umkreist,umkreist,ein,astronom,entdeckt,eine,sonne],[]), drsp0(DRS,2).