Computerlinguistik II
CIS, WS 2008/09

Hans Leiß

Universität München

Centrum für Informations- und Sprachverarbeitung

26. Januar 2009

Angewandte Computerlinguistik
am Beispiel eines Datenbankabfragesystems

Inhalt:

- Lexikographie: Vollformen- vs. Stammformenlexikon
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing)
- Semantik:
  1. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  2. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

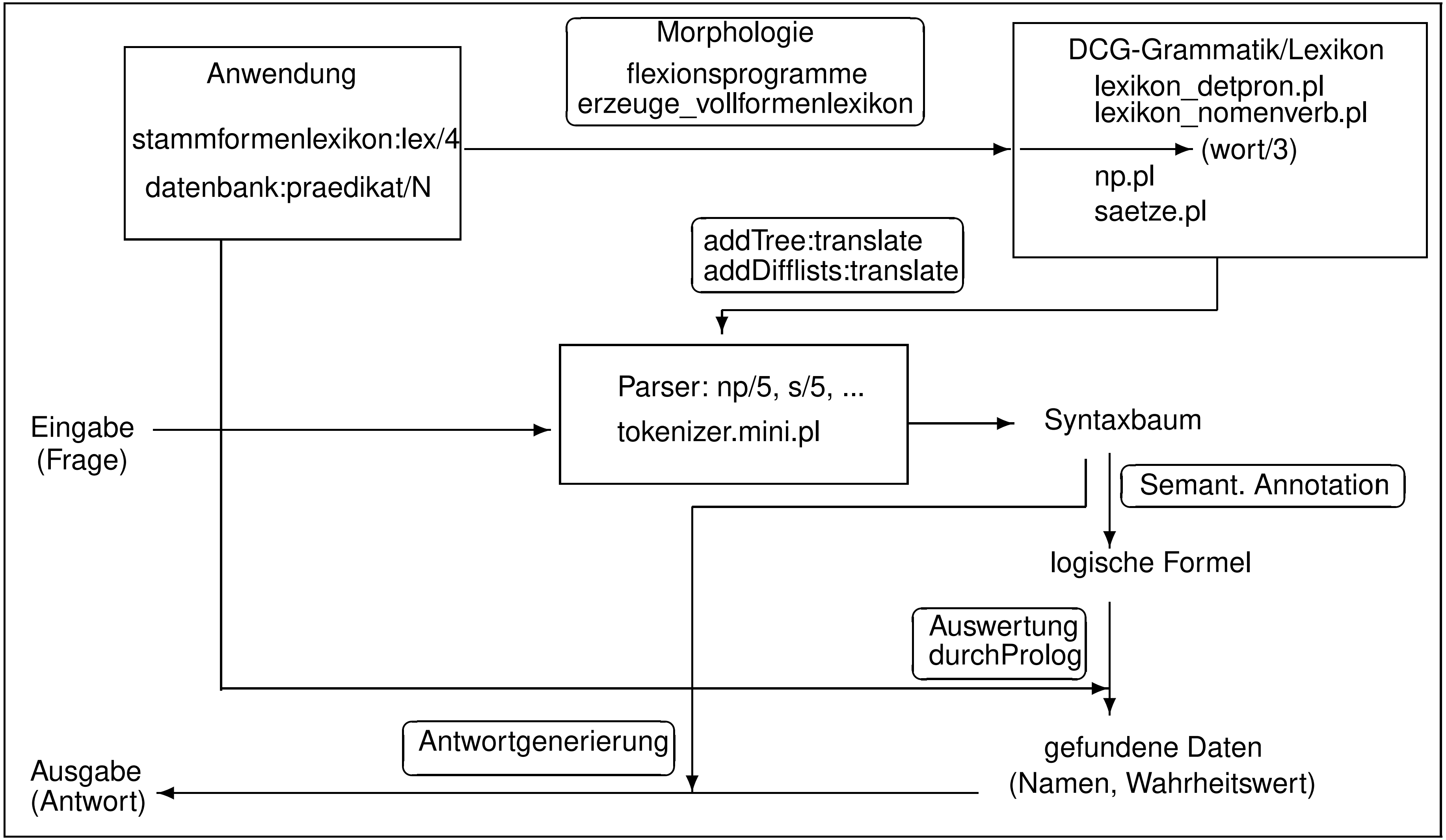
Programmiersprache:

- Prolog (SWI-Prolog ab Version 5.6.14)

Überblick

- Datenbank: Fakten über Sterne, Planeten, Astronomen, Größen
- Natürlichsprachliche Anfragen: z.B.
  “Welche Astronomen haben drei Monde eines Planeten entdeckt?”
- Lexikalische und syntaktische Analyse der Anfrage (Syntaxbaum)
- Berechnung der Semantik: Syntaxbaum $\mapsto$ Formel
- Datenbanksuche: Welche Belegung der Variablen macht die Formel wahr?
- Antwortgenerierung in natürlicher Sprache:
  “Galilei und Herschel haben drei Monde entdeckt.”

Überblick


Morphologie
flexionsprogramme
erzeuge_vollformenlexikon
Anwendung
stammformenlexikon:lex/4
datenbank:praedikat/N
DCG-Grammatik/Lexikon
lexikon_detpron.pl
lexikon_nomenverb.pl
(wort/3)
np.pl
saetze.pl
addTree:translate
addDifflists:translate
Parser: np/5, s/5, ...
tokenizer.mini.pl
Eingabe
(Frage)
Syntaxbaum
Semant. Annotation
logische Formel
Auswertung
durchProlog
Antwortgenerierung
Ausgabe
(Antwort)
gefundene Daten
(Namen, Wahrheitswert)

Prolog: Terme

Terme in Prolog sind einfach oder zusammengesetzt:

$\langle$*Term*$\rangle\equiv$

```
Term := Var
      | Atom
      | Const
      | Atom(Term,...,Term)
```

Einfache Terme sind *Variable* oder *Atome*.

Variable sind Zeichenreihen (ohne Sonderzeichen außer _), die mit einem Großbuchstaben oder mit _ beginnen:

$\langle$*Variable von Prolog*$\rangle\equiv$

```
Var := [A-Z]([a-zA-Z0-9_])*
     | _([a-zA-Z0-9_])*        % anonyme Variable
```

Kommt eine Variable nur einmal vor, so sollte sie anonym sein.

Atome beginnen mit Kleinbuchstaben oder stehen in einfachen Anführungszeichen (dann auch mit Sonderzeichen wie `+\./-`):

⟨*Atome von Prolog*⟩≡

```
Atom := [a-z]([a-zA-Z0-9_])*
      | '[A-Z]([a-zA-Z0-9_\/.-+])*'
```

Atomare(!) Terme sind einfache Terme oder Konstante, d.h. nichtleere Ziffernfolgen:

⟨*Konstante von Prolog*⟩≡

```
Const := [0-9]([0-9])*
```

Zusammengesetzte Terme sind die `Atom(Term,...,Term)`. Dazu zählen die oft verwendeten *Listen* `[Term,...,Term]`:

⟨*Listen in Prolog*⟩≡

```
Liste := []                 % Atom, die leere Liste
      | [Term | Liste]      % statt: '.'(Term,Liste)
```

Prolog: Programme, Regeln

Ein Prolog-*Programm* ist eine Folge von *Regeln* oder *Klauseln*, die durch Leerzeichen (oder Zeilenumbruch) getrennt sind:

⟨*Programm*⟩≡

```
Regel1
...
RegelN
```

Eine *Regel* ist von der Form

⟨*Regel*⟩≡

```
Kopf :- Rumpf.        % Kopf gilt, wenn Rumpf gilt
```

wobei der *Kopf* ein Atom oder zusammengesetzter Term und der *Rumpf* ein Prolog-Ziel ist. Der Rumpf `true` darf fehlen:

⟨*Faktum := Regel mit Rumpf* `true`⟩≡

```
Kopf :- true.
Kopf.                 % alternative Schreibweise
```

Prolog: (Beweis-) Ziele

Ziele können einfach oder zusammengesetzt sein:

⟨*Ziele* ⟩≡

```
  Ziel := Term                        % einfaches Ziel
       | true                         % erfuelltes Ziel
       | fail                         % unerfuellbares Ziel
       | (Ziel, Ziel)                 % (Ziel1 und  Ziel2)
       | (Ziel; Ziel)                 % (Ziel1 oder Ziel2)
       | (Ziel -> Ziel1; Ziel2)       % (Wenn -> Dann ; Sonst)
       | ...
```

Klammerersparung: `(A,B,C)=(A,(B,C))`, `(A;B;C)=(A;(B;C))`

Bindungsstärke: `(A,B;C)=((A,B);C)`, `(A;B,C)=(A;(B,C))`.

Prolog: Abarbeitung von Zielen

Ist ein Programm `Regel1 ... RegelN` geladen, so kann man Prolog eine Frage oder ein (Beweis-)Ziel stellen:

$\langle$*Beweissuche*$\rangle \equiv$

```
?- Ziel.
```

Das Ziel `true` gilt als bewiesen, `fail` als unbeweisbar.

Ein einfaches Ziel wird wie folgt bearbeitet:

1. Suche die erste Regel, deren Kopf zum Ziel „paßt“;
2. Mache Kopf und Ziel durch Variablenbelegung gleich;
3. Bearbeite den Rumpf der Regel (mit belegten Variablen);
4. Falls es gelingt, antworte mit der Variablenbelegung/`Yes`;
5. Falls es scheitert, suche die nächste passende Regel;
6. Falls es keine passende Regel gibt, antworte mit `No`.

Ein zusammengesetztes Ziel wird wie folgt bearbeitet:

1. Bei `(Ziel1, Ziel2)` bearbeite `Ziel1` und mit dessen Lösung (Variablenbelegung) dann `Ziel2`. Falls das keine Lösung hat, suche weitere Lösung von `(Ziel1, Ziel2)`.

2. Bei `(Ziel1; Ziel2)` bearbeite `Ziel1`; falls es keine Lösung hat, bearbeite (ohne Variablenbelegung) `Ziel2`.

3. Bei `(Ziel -> Ziel1; Ziel2)` bearbeite `Ziel`, und falls es eine Lösung gibt, nimm die erste und bearbeite `Ziel1`, andernfalls bearbeite `Ziel2`.

Beispiel: Beweissuche

⟨*Beispiele/form.pl*⟩≡

```
form(nomen,[Num,Kas]) :-  (numerus(Num), kasus(Kas)).
numerus(X) :- (X = sg ; X = pl).
kasus(X) :- (X = nom ; X = gen ; X = dat ; X = akk).
```

⟨*Frage 1*⟩≡

```
?- form(nomen,[sg,Y]).
```

Das Ziel paßt mit `sg=Num,Y=Kas` zum Kopf der ersten Regel, also wird statt des Ziels der (belegte) Rumpf bearbeitet:

⟨*Frage 2*⟩≡

```
?- numerus(sg), kasus(Y).
```

Das erste Teilziel paßt zum Kopf der zweiten Regel, also wird

⟨*Frage 3*⟩≡

```
?- (sg = sg ; sg = pl), kasus(Y).
```

bearbeitet.

Das erste Teilziel hiervon ergibt `Yes` mit leerer Belegung, also wird als nächstes

⟨*Frage 4*⟩≡

```
?- kasus(Y).
```

bearbeitet. Dieses Ziel paßt mit `Y=X` zum Kopf der dritten Regel, also wird der Rumpf

⟨*Frage 5*⟩≡

```
(Y = nom ; Y = gen ; Y = dat ; Y = akk).
```

bearbeitet. Das erste Teilziel hiervon hat die Lösung `Y=nom`, also ist nichts mehr zu zeigen.

Die (erste) Lösung von *Frage 1* ist daher `Y = nom.`

Prolog: Konventionen

Im Term `Atom(Term1,...,TermN)` heißt `Atom` der *Funktor* und `Term1` bis `TermN` die *Argumente*.

Im Unterschied zum `Atom` hat der Funktor eine *Stelligkeit* `N`, die Anzahl der Argumente, und wird als `Atom/N` angegeben.

Dasselbe Atom kann als Funktor mit unterschiedlicher Stelligkeit benutzt werden:

⟨*Atom* `planet` *als 1-stelliger Funktor* `planet/1`⟩≡

```
planet(X) :- fixstern(Y), umkreist(X,Y).
```

⟨*Atom* `planet` *als 2-stelliger Funktor* `planet/2`⟩≡

```
planet(X,Y) :- fixstern(Y), umkreist(X,Y).
```

Ein *Prädikat* ist ein Funktor `P/N` zusammen mit einer Definition durch Programmregeln:

⟨*Definition des Prädikats* `P`⟩≡

```
P(Term_1_1,...,Term_1_N) :- Rumpf_1.
...
P(Term_K_1,...,Term_K_N) :- Rumpf_K.
```

Argumentstellen eines Prädikats dienen i.a. zur Ein- *und* Aus- gabe. Für viele Prädikate sind aber manche nur zur Eingabe und manche nur zur Ausgabe gedacht; das deutet man durch `+`, `?`, `-` im Kommentar `% ...` an.

⟨*Angabe der erlaubten Verwendungsweise der Argumente*⟩≡

```
% Atom(+Eingabe,?Ein/Ausgabe,-Ausgabe).
```

⟨*Beispiel* ⟩≡

```
% concat_atom(+AtomList, -Atom)
?- concat_atom([das,' ',lange,' ','Atom'],Atom).
Atom = 'das lange Atom'.
```

Äquivalente Schreibweisen

Äquivalente Programme sind:

$\langle$*Alternative im Rumpf*$\rangle \equiv$

```
Kopf :- Ziel1; Ziel2.
```

$\langle$*Alternative Regeln mit gleichem Kopf*$\rangle \equiv$

```
Kopf :- Ziel1.
Kopf :- Ziel2.
```

Simulation einer Schleife

⟨*Schleife in Prolog*⟩≡

```
Kopf :- (Ziel, fail ; true).
```

⟨*Schleife, alternativ geschrieben*⟩≡

```
Kopf :- Ziel,   % suche Loesuung fuer Ziel
        fail.   % suche andere Loesung
Kopf.           % fertig
```

⟨*Beispiele/schleife.pl*⟩≡

```
alle_kinder(V,M) :-
       (kind(K,V,M), write(K), fail) ; true.

kind(abel,adam,eva).
kind(kain,adam,eva).
```

Prolog: Programme laden

Man lädt von Prolog aus Programmdateien durch

⟨*Datei laden*⟩≡

```
?- consult(<Pfad/Dateiname>).
```

oder gleich mehrere Dateien:

⟨*Dateien laden*⟩≡

```
?- consult(['Pfad/datei_1.pl','datei_2.pl']).
```

Eine andere Schreibweise ist:

⟨*Dateien laden*⟩+≡

```
?- ['Pfad/datei_1.pl', 'datei_2.pl'].
```

Wenn der Dateiname ohne Suffix `.pl` oder `.pro` ein Atom ist, braucht man Anführungszeichen und Suffix nicht:

⟨*Dateien laden*⟩+≡

```
?- ['Pfad/datei_1', datei_2].
```

Hilfsdateien aus einer Datei laden lassen

Soll beim Laden einer Datei eine Hilfsdatei `hilfsdatei.pl` ebenfalls geladen werden, so kann man das in der Datei verlangen:

$\langle$*In der Hauptdatei*$\rangle\equiv$

```
:- consult(hilfsdatei).
```

Man kann auch die alternative Schreibweise verwenden:

$\langle$*In der Hauptdatei*$\rangle{+}{\equiv}$

```
:- ['Pfadname/hilfsdatei.pl',...]. % ... weitere Dateien
```

Soll die Hilfsdatei nur dann geladen werden, wenn sie nicht schon geladen ist, so benutze man

$\langle$*In der Hauptdatei*$\rangle{+}{\equiv}$

```
:- ensure_loaded(hilfsdatei).
```

Prolog: Fehlersuche

Die Schritte beim Lösen eines Ziels kann man anzeigen lassen.

⟨*Beispiele/testprogramm.pl*⟩≡

```
% Fakten:
planet(jupiter,sonne).    % Jupiter ist ein Planet der Sonne
planet(erde,sonne).       % Die Erde ist ein Planet der Sonne
mond(uranus,jupiter).     % Uranus ist ein Mond des Jupiter
mond(mond,erde).          % Der Mond ist ein Mond der Erde
% Regeln:
% X umkreist Y, wenn X ein Mond oder ein Planet von Y ist:
umkreist(X,Y) :-
    (mond(X,Y) ; planet(X,Y)).
% X umkreist Y, wenn X Mond eines Sterns Z ist, der Y umkreis
umkreist(X,Y) :-
    mond(X,Z), umkreist(Z,Y).
```

⟨*Ablaufverfolgung (Tracing)*⟩≡

```
?- ['Beispiele/testprogramm'].    % Lade das Programm
?- trace(planet/2,[+call,+exit]). % Zeige Aufrufe und Antwort
                                  % von planet/2 beim folg.Zi
[debug] ?- umkreist(X,sonne).     % Welche X umkreisen die So
 T Call: (8) planet(_G273, sonne)
 T Exit: (8) planet(jupiter, sonne)
X = jupiter ;                     % erste Lösung. ; für weite
 T Exit: (8) planet(erde, sonne)
X = erde ;
 T Call: (9) planet(jupiter, sonne)
 T Exit: (9) planet(jupiter, sonne)
X = uranus ;
 T Call: (9) planet(erde, sonne)
 T Exit: (9) planet(erde, sonne)
X = mond ;
No                               % keine weiteren Lösungen
?- nodebug.                      % Ablaufverfolgung ausschalt
```

Prolog: Module

Ein *Modul* ist eine Programmdatei, die nur bestimmte ihrer Definitionen weitergeben soll. Das wird deklariert durch

⟨*Moduldatei: Modulname 'paket', Exportliste [p/2]*⟩≡

```
:- module(paket,[p/2]).     % am Dateianfang!

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

Nach dem Laden ist das exportierte 2-stellige Prädikat mit *kurzem Namen* `p/2` bekannt; die Hilfsprädikate `p/1` und `q/1` sind nur mit *langem Namen* `paket:p/1,paket:q/1` bekannt:

⟨*Kurze und lange Prädikatnamen*⟩≡

```
?- p(X,Y), paket:p(U), paket:q(V).
```

Vorteil: (Modul = Namensraum) Gleich benannte Hilfsfunktionen aus verschiedenen Modulen stören einander nicht.

Module von anderen Dateien laden

Soll eine andere Datei eine Moduldatei laden, benutze

⟨*Moduldatei mit Dateiname und Importliste laden*⟩≡

```
:- use_module(<Moduldateiname>). % alles importieren
:- use_module(<Moduldateiname>,<Importliste>).
```

Die Importliste ist eine Auswahl von Prädikaten der Exportliste.

Das Top-Level kann als Modul `user`, die Systemdateien als Modul `system` betrachtet werden. Prädikate, die ein Modul nicht selbst definiert, importiert er von `user` und `system`.

Der Modulname in einer Moduldatei muß nicht der Moduldateiname sein.

Modul: Wortformen

Ein Programmmodul steht in einer eigenen Datei, hat einen Namen und eine Liste der exportierten Prädikate:

⟨*Morphologie/wortformen.pl*⟩≡

```
:- module(wortformen,
          [form/2,
           person/1,numerus/1,tempus/1,modus/1]).

% --- Finite Verbformen: [Pers,Num,Temp,Mod] -----

form(vfin,[Pers,Num,Temp,Mod]) :-
    person(Pers), numerus(Num),
    tempus(Temp), modus(Mod).

% Fehlt: Einschraenkung von Pers bei Verben mit es-Subjekt,
%        Imperativformen im sg und pl.
```

⟨*Morphologie/wortformen.pl*⟩+≡

```
% --- Definition der Wertebereiche der Formmerkmale:

person(1). person(2). person(3).

numerus(sg).    % Singular
numerus(pl).    % Plural

tempus(praes). % Praesens
tempus(praet). % Praeteritum

modus(ind).     % Indikativ
modus(konj).    % Konjunktiv
```

Später kommen Definitionen der Formmerkmale für Nomen usw. hinzu.

Vollformenlexion

Ein Vollformenlexikon gibt zu jedem Lexem (in Stammform) und jeder durch Formmerkmale angegebenen abstrakten Wortform die konkrete Vollform des Lexems an.

In Prolog kann man das leicht durch eine Tabelle der Form

$\langle$*Vollformenlexikon, Schema*$\rangle\equiv$

```
wort(?Stammform,Wortart(?Art-,?Formmerkmale),?Vollform).
```

machen, ein Prädikat `wort/3`.

Je nach Belegung der Komponenten fragt man nach der Vollform, der Stammform, oder den Formmerkmalen.

Bei Verben hätte man z.B. folgende Einträge:

⟨*Beispiele/vollformenlexikon.pl*⟩≡

```
:- module(vollformenlexikon,[wort/3]).

% Schwach konjugiertes ([nom,akk]=transitives) Verb:

wort(entdecken,v([nom,akk],[1,sg,praes,ind]),entdecke ).
wort(entdecken,v([nom,akk],[2,sg,praes,ind]),entdeckst).
wort(entdecken,v([nom,akk],[3,sg,praes,ind]),entdeckt ).
wort(entdecken,v([nom,akk],[1,pl,praes,ind]),entdecken).
wort(entdecken,v([nom,akk],[2,pl,praes,ind]),entdeckt ).
wort(entdecken,v([nom,akk],[3,pl,praes,ind]),entdecken).
wort(entdecken,v([nom,akk],[1,sg,praet,ind]),entdeckte  ).
wort(entdecken,v([nom,akk],[2,sg,praet,ind]),entdecktest).
wort(entdecken,v([nom,akk],[3,sg,praet,ind]),entdeckte  ).
wort(entdecken,v([nom,akk],[1,pl,praet,ind]),entdeckten ).
wort(entdecken,v([nom,akk],[2,pl,praet,ind]),entdecktet ).
wort(entdecken,v([nom,akk],[3,pl,praet,ind]),entdeckten ).
```

⟨*Beispiele/vollformenlexikon.pl*⟩+≡

```
% Stark konjugiertes ([nom]=intransitives) Verb:

wort(gehen,v([nom],[1,sg,praes,ind]),gehe ).
wort(gehen,v([nom],[2,sg,praes,ind]),gehst).
wort(gehen,v([nom],[3,sg,praes,ind]),geht ).
wort(gehen,v([nom],[1,pl,praes,ind]),gehen).
wort(gehen,v([nom],[2,pl,praes,ind]),geht ).
wort(gehen,v([nom],[3,pl,praes,ind]),gehen).
wort(gehen,v([nom],[1,sg,praet,ind]),ging  ).
wort(gehen,v([nom],[2,sg,praet,ind]),gingst).
wort(gehen,v([nom],[3,sg,praet,ind]),ging  ).
wort(gehen,v([nom],[1,pl,praet,ind]),gingen).
wort(gehen,v([nom],[2,pl,praet,ind]),gingt ).
wort(gehen,v([nom],[3,pl,praet,ind]),gingen).
```

Falls eine Form für den Konjunktiv gesucht wird, soll eine Warnung ausgegeben werden:

⟨*Beispiele/vollformenlexikon.pl*⟩+≡

```
  wort(_,v(_,[_,_,_,konj]),_) :-
      write(user,'Finite Verbformen im Konjunktiv
                  sind nicht definiert.\n'),
      fail.
```

Die *unzusammenhängenden* Wortformen, z.B. die mit abgetrenntem Verbpräfix wie in `hörst du auf?`, werden nicht behandelt.

Bem: Wenn (wie bisher) alle Prädikate exportiert werden, hat die Verpackung als Modul wenig Sinn.

Flexionsprogramm mit Vollformenlexikon

Die Vollform wird durch Nachsehen in der Tabelle `wort/3` bestimmt. Im Normalfall soll die Vollform weiter benutzt werden; daher ist sie ein Ausgabeargument:

⟨*Morphologie/flexion_vlex.pl*⟩≡

```
:- module('flexion_vlex',[]).
:- use_module(['wortformen.pl']).
% Verbflexion (zu einem geladenen Vollformenlexikon word/3)
% verbflexion(+Stammform,+Formmerkmale,-Vollform)

verbflexion(Stammform,Formmerkmale,Vollform) :-
    wort(Stammform,v(_,Formmerkmale),Vollform).
```

Will man die Vollformen nur sehen, nicht benutzen, kann man sich alle Formen am Bildschirm anzeigen lassen:

⟨*Morphologie/flexion_vlex.pl*⟩+≡

```
% Flexionsprogramm: verbflexion(+Stammform)
verbflexion(Stammform) :-    % Stammform = Infinitiv
    form(vfin,Formmerkmale),      % moegl.Merkmale holen
    wort(Stammform,v(_,Formmerkmale),Vollform),
    write(Formmerkmale),write(': '),write(Vollform),nl,
    fail.                         % weitere Loesungen?
verbflexion(_).                  % fertig.
```

⟨*Anwendungsbeispiel* ⟩≡

```
?- ['Morphologie/flexion_vlex',
    'Beispiele/vollformenlexikon.pl'],
   flexion_vlex:verbflexion(gehen).
```

Stammformenlexikon

Ein Stammformenlexikon enthält zu jedem Lexem eine *Stammform* und die Angabe der *Flexionsklasse*, nach der aus der Stammform die Vollformen gebildet werden.

Die Flexionsklasse kann in vielen Fällen durch eine *Endungstabelle* angegeben werden, aber oft müssen auch *Veränderungen am Wortstamm* (Ablaute u.a.) vorgenommen oder Laute zwischen Stamm und Endung eingefügt werden.

Trennt man von der Angabe der Flexionsklasse die Angabe der Wortart (und davon die Artmerkmale, bei Verben den Komplementrahmen `[nom,akk]`) ab, so ist das Stammlexikon eine Tabelle `lex/4`:

⟨*Beispiele/stammformenlexikon.pl*⟩≡

```
:- module(stammformenlexikon,[lex/4]).

% lex(?Stammform,?Wortart,?Artmerkmale,?Flex.klasse)
lex(ablegen, v,[nom,akk],     rg(2)            ).
lex(aufgeben,v,[nom,akk],     urg(3,(e,i,a,e),5)).
lex(geben,   v,[nom,dat,akk], urg(0,(e,i,a,e),2)).
lex(heben,   v,[nom,akk],     urg(0,(e,e,o,o),2)).
lex(entdecken,v,[nom,akk],    rg(0)            ).
lex(singen,  v,[nom,akk],     urg(0,(i,i,a,u),2)).
lex(umfahren,v,[nom,akk],     urg(2,(a,ä,u,a),4)).
lex(umfahren,v,[nom,akk],     urg(0,(a,ä,u,a),4)).
```

Die Angabe der Flexionsklasse wird unten erläutert.

⟨*Unregelmäßige Verbflexion:*⟩≡

```
urg(|betontes Praefix|,Ablautreihe,Ablautposition)
```

Flexionsprogramm mit Stammformenlexikon

Aus den Einträgen eines Stammlexikons `lex/4` bildet man die Vollformen, indem man

- die Wortart und die Flexionsklasse abliest (mit `lex/4`),
- eine für die Wortart erlaubte abstrakte Form sucht (mit `form/2`),
- aus dem Stamm und der abstrakten Form mit einem Flexionsprogramm die Vollform konstruiert (mit `beuge/5`).

Verschiedene Wortarten haben verschiedene Flexionsweisen, z.B. Konjugation beim Verb und Deklination beim Nomen.

Für die Verben fügen wir in `wortfomen.pl` ein, daß mindestens die finiten Verbformen erlaubte Formen sind:

⟨*Morphologie/wortformen.pl*⟩+≡

```
form(v,Formmerkmale) :- form(vfin,Formmerkmale).
```

⟨*Morphologie/flexion_slex.pl*⟩≡

```
:- module(flexion_slex,[erzeuge_vollformenlexikon/1]).
:- use_module([wortformen]).

flektiere(Stammform,Vollform) :-
    lex(Stammform,Wortart,_,Flexionsklasse),
    form(Wortart,Formmerkmale),
    beuge(Wortart,Flexionsklasse,Stammform,
          Formmerkmale,Vollform).

beuge(v,Flexionsklasse,Stammform,Form,Vollform) :-
    konjugiere(Flexionsklasse,Stammform,Form,Vollform).
beuge(Wortart,Flexionsklasse,Stammform,Form,Vollform) :-
    (Wortart = n ; Wortart = rn ; Wortart = en),
    dekliniere(Flexionsklasse,Stammform,Form,Vollform).
```

Anzeige aller Vollformen

Bei der Anzeige aller Vollformen am Bildschirm testet man am besten, ob die Stammform im Stammlexikon vorkommt:

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
flexion(Stammform) :-
    ( lex(Stammform,v,_,Flexionsklasse)          % Verbstamm?
    ->  verbflexion(Flexionsklasse,Stammform)
    ; lex(Stammform,Wortart,_,Flexionsklasse), % Nomenstamm?
        member(Wortart,[n,rn,en])
    ->  nomenflexion(Flexionsklasse,Stammform)
    ; nl,
        write('Das Stammlexikon hat keinen Eintrag zu: '),
        write(Stammform)
    ).
```

Je nach der Konjugationsklasse wird der Verbstamm in eine finite Vollform umgewandelt und angezeigt:

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
% verbflexion(+Konjugationsklasse,+Stammform)
verbflexion(Konjugationsklasse,Stammform) :-
    form(vfin,Formmerkmale),
    konjugiere(Konjugationsklasse,Stammform,
               Formmerkmale,Vollform),
    write(Formmerkmale),write(': '),write(Vollform),nl,
    fail.               % weitere Formmerkmale suchen
verbflexion(_,_).      % fertig.
```

Das analoge `nomenflexion(+Deklinationsklasse,Stammform)` muß noch geschrieben werden (Übung).

Beispiel: Verbflexion

Bei der Flexion der Verben, der *Konjugation*, unterscheidet man *schwache/regelmäßige* und *starke/unregelmäßige* Konjugation. Grob gesagt, wird bei der schwachen nur die Endung verändert, bei der starken auch der Stammlaut.

Die *Ablautfolge* ist die Folge der Stammvokale bestimmter Formen, etwa `(i,i,a,u)` bei *s<u>i</u>ngen,s<u>i</u>ngt,s<u>a</u>ng,ges<u>u</u>ngen*. Man braucht den Stammvokal der 3.Person Singular Präsens:

h<u>e</u>ben,h<u>e</u>bt,h<u>o</u>b,geh<u>o</u>ben: `(e,e,o,o)`, *g<u>e</u>ben,g<u>i</u>bt,g<u>a</u>b,geg<u>e</u>ben*: `(e,i,a,o)`

Weiter muß man beachten, ob das Verb ein *betontes Präfix* hat: das wird bei bestimmten Verbstellungen im Satz vom Stamm getrennt, und bei den infiniten Formen (Partizip-2 und zu-Infinitiv) stehen `ge` bzw. `zu` zwischen Präfix und Stamm.

Die Endungstabellen bei schwacher und starker Konjugation unterscheiden sich, und manchmal muß ein `e` vor die Endung eingeschoben werden.

Aus der Länge des betonten Präfixes, der Ablautfolge, und der Position des Ablauts kann man mit den passenden Endungstabellen die Verbformen konstruieren.

Die Flexionsklassen kodieren wir daher durch

- `rg(|betontes Präfix|)` für die regelmäßige Konjugation,
- `rge(|betontes Präfix|)` für die regelmäßige Konjugation mit `e`-Einschub,
- `urg(|betontes Präfix|,Ablaute,Ablautposition)` für die unregelmäßige Konjugation.

Bem. Ablautung kommt auch bei einigen schwachen Verben vor: *brennen,brennt,brannte,gebrannt*.

a. Regelmäßige Verbflexion

Aus der Stammform und den Merkmalen einer finiten Form bildet man die Vollform, indem man

- von der Stammform die Infinitiv-Endung `en` entfernt, was den Verbstamm ergibt,
- bei manchen Stämmen an den Stamm ein `e` anfügt, und
- daran die für die finite Form nötige Endung anhängt.

Die Einfügung eines `e` machen wir nur für Dentalstämme.

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
% konjugiere(+Konjugationsklasse,+Infinitiv,
%             ?[Pers,Num,Temp,Mod],-Vollform)

% --- Regelmaessige Konjugation (Finite Formen) ---

konjugiere(rg(_),Infinitiv,[Pers,Num,Temp,Mod],Vollf) :-
    concat(Stamm,'en',Infinitiv),
    (dentalstamm(Stamm)
    -> endung(rge,[Pers,Num,Temp,Mod],Endung)
    ;  endung(rg, [Pers,Num,Temp,Mod],Endung)
    ),
    concat(Stamm,Endung,Vollf).

dentalstamm(Stamm) :-
    concat(_,'d',Stamm) ; concat(_,'t',Stamm).
```

Endungstabellen für die regelmäßige Konjugation

Die Endungen entnimmt man einer Tabelle `endung/3`.

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
% -- regelmaessige Konjugation --

endung(rg,[1,sg,praes,ind], 'e').
endung(rg,[1,sg,praes,konj],'e').
endung(rg,[1,sg,praet,ind], 'te').
endung(rg,[1,sg,praet,konj],'te').

endung(rg,[1,pl,praes,ind], 'en').
endung(rg,[1,pl,praes,konj],'en').
endung(rg,[1,pl,praet,ind], 'ten').
endung(rg,[1,pl,praet,konj],'ten').

endung(rg,[2,sg,praes,ind], 'st').
endung(rg,[2,sg,praes,konj],'est').
```

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
endung(rg,[2,sg,praet,ind], 'test').
endung(rg,[2,sg,praet,konj],'test').

endung(rg,[2,pl,praes,ind], 't').
endung(rg,[2,pl,praes,konj],'et').
endung(rg,[2,pl,praet,ind], 'tet').
endung(rg,[2,pl,praet,konj],'tet').

endung(rg,[3,sg,praes,ind], 't').
endung(rg,[3,sg,praes,konj],'e').
endung(rg,[3,sg,praet,ind], 'te').
endung(rg,[3,sg,praet,konj],'te').

endung(rg,[3,pl,praes,ind], 'en').
endung(rg,[3,pl,praes,konj],'en').
endung(rg,[3,pl,praet,ind], 'ten').
endung(rg,[3,pl,praet,konj],'ten').
```

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
% ---- mit e-Erweiterung:

endung(rge,[1,sg,praes,ind], 'e').
endung(rge,[1,sg,praes,konj],'e').
endung(rge,[1,sg,praet,ind], 'ete').
endung(rge,[1,sg,praet,konj],'ete').

endung(rge,[1,pl,praes,ind], 'en').
endung(rge,[1,pl,praes,konj],'en').
endung(rge,[1,pl,praet,ind], 'eten').
endung(rge,[1,pl,praet,konj],'eten').

endung(rge,[2,sg,praes,ind], 'est').
endung(rge,[2,sg,praes,konj],'est').
endung(rge,[2,sg,praet,ind], 'etest').
endung(rge,[2,sg,praet,konj],'etest').
```

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
endung(rge,[2,pl,praes,ind], 'et').
endung(rge,[2,pl,praes,konj],'et').
endung(rge,[2,pl,praet,ind], 'etet').
endung(rge,[2,pl,praet,konj],'etet').

endung(rge,[3,sg,praes,ind], 'et').
endung(rge,[3,sg,praes,konj],'e').
endung(rge,[3,sg,praet,ind], 'ete').
endung(rge,[3,sg,praet,konj],'ete').

endung(rge,[3,pl,praes,ind], 'en').
endung(rge,[3,pl,praes,konj],'en').
endung(rge,[3,pl,praet,ind], 'eten').
endung(rge,[3,pl,praet,konj],'eten').
```

Das Flexionsprogramm enthält kein Stammlexikon `lex/4`; erst wenn eins geladen ist, kann man dessen Wörter flektieren:

⟨*Beispiel: regelmäßige Verbflexion*⟩≡

```
?- ['Morphologie/flexion_slex'].
?- flexion_slex:flexion(ablegen).
ERROR: Undefined procedure: flexion_slex:lex/4

?- ['Beispiele/stammformenlexikon'].
?- flexion_slex:flexion(ablegen).

[1, sg, praes, ind]: ablege
[1, sg, praes, konj]: ablege
[1, sg, praet, ind]: ablegte
...
?- flexion_slex:flexion(anlegen).
Das Stammlexikon enthält keinen Eintrag zu: anlegen
```

b. Unregelmäßige Verbflexion

Aus der Stammform und den Merkmalen einer finiten Form bildet man die Vollform, indem man

- von der Stammform die Infinitiv-Endung `en` entfernt, was den Stamm ergibt,
- den Stamm an der aus der Flexionsklasse hervorgehenden Position in den Stammvokal, den Teil davor und den danach aufspaltet,
- je nach der Form den Stammvokal durch einen aus der Ablautreihe ersetzt und die passende Endung anhängt.

Bem. Weitere, manchmal nötige Änderungen am Verbstamm implementieren wir nicht; z.B. die Stammänderungen in `gehen,geht,ging,gegangen`.

Für solche Fälle wäre es einfacher, die Flexionsklasse durch wenige Vollformen darzustellen und die übrigen anderen daraus zu erzeugen.

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
konjugiere(urg(_,(V1,V2,V3,V4),Ablautposition),
           Infinitiv,[Pers,Num,Temp,Mod],Vollform) :-
    concat(Stamm,en,Infinitiv),        % Zerlege z.B.
    LaengeVor is Ablautposition - 1,   % anfangen =
    sub_atom(Stamm,0,LaengeVor,_,Vor), % anf|a | ng |en
    infix(Vor,V1,Nach,Stamm),          % Vor|V1|Nach|en
    verbform((Vor,Nach),(V1,V2,V3,V4),
             [Pers,Num,Temp,Mod],Vollform).

% infix(+Vor,+Mitte,-Nach,+Kette) oder s.u.
% infix(+Vor,+Mitte,+Nach,-Kette)
infix(Vor,Mitte,Nach,Kette) :-
    concat(Vor,Mitte,Praefix),
    concat(Praefix,Nach,Kette).
```

Bildung der finiten Verbformen

Im Präsens Indikativ setzt man bei der 2. und 3. Person Singular den vorgegebenen Ablaut V2 im Stamm ein; dann hängt man die passende Endung an:

$\langle$*Morphologie/flexion_slex.pl*$\rangle{+}{\equiv}$

```
verbform((Vor,Nach),(V1,V2,_V3,_V4),
          [Pers,Num,praes,ind],Vollform) :-
    ( not(member([Pers,Num],[[2,sg],[3,sg]])),
      infix(Vor,V1,Nach,Stamm)
    ; member([Pers,Num],[[2,sg],[3,sg]]),
      infix(Vor,V2,Nach,Stamm)
    ),
    endung(urg,[Pers,Num,praes,ind],Endung),
    concat(Stamm,Endung,Vollform).
```

Im Präteritum Konjunktiv ist V3 durch seinen Umlaut zu ersetzen:

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
verbform((Vor,Nach),(_V1,_V2,V3,_V4),
         [Pers,Num,praet,ind],Vollform) :-
    infix(Vor,V3,Nach,Stamm),
    endung(urg,[Pers,Num,praet,ind],Endung),
    concat(Stamm,Endung,Vollform).
verbform((Vor,Nach),(_V1,_V2,V3,_V4),
         [Pers,Num,praet,konj],Vollform) :-
    (member(V3,[a,o,u]) -> umlaut(V3,U) ; U = V3),
    infix(Vor,U,Nach,Stamm),
    endung(urg,[Pers,Num,praet,konj],Endung),
    concat(Stamm,Endung,Vollform).

umlaut(a,ä).  umlaut(o,ö).  umlaut(u,ü).
```

Endungstabelle für unregelmäßige Konjugation

Auch hier brauchen wir eine Endungstabelle; die nur manchmal nötigen `e`-Erweiterungen sind stets mit aufgeführt:

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
% -- unregelmaessige Konjugation --
endung(urg,[1,sg,praes,ind], 'e').
endung(urg,[1,sg,praes,konj],'e').
endung(urg,[1,sg,praet,ind], '').
endung(urg,[1,sg,praet,konj],'e').

endung(urg,[2,sg,praes,ind], 'st').
endung(urg,[2,sg,praes,ind], 'est').
endung(urg,[2,sg,praes,konj],'est').
endung(urg,[2,sg,praet,ind], 'st').
endung(urg,[2,sg,praet,ind], 'est').
endung(urg,[2,sg,praet,konj],'st').
endung(urg,[2,sg,praet,konj],'est').
```

Übung: Man verbessere `verbform/4` so, daß die `e`-Einfügungen zwischen Stamm und Endung nur dort vorgenommen werden, wo sie nötig sind, sodaß man in `endung/3` nur die echten Endungen braucht.

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
endung(urg,[3,sg,praes,ind], 't').
endung(urg,[3,sg,praes,ind], 'et').
endung(urg,[3,sg,praes,konj],'e').
endung(urg,[3,sg,praet,ind], '').
endung(urg,[3,sg,praet,konj],'e').

endung(urg,[1,pl,praes,ind], 'en').
endung(urg,[1,pl,praes,konj],'en').
endung(urg,[1,pl,praet,ind], 'en').
endung(urg,[1,pl,praet,konj],'en').
```

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
endung(urg,[2,pl,praes,ind], 't').
endung(urg,[2,pl,praes,ind], 'et').
endung(urg,[2,pl,praes,konj],'et').
endung(urg,[2,pl,praet,ind], 't').
endung(urg,[2,pl,praet,ind], 'et').
endung(urg,[2,pl,praet,konj],'t').
endung(urg,[2,pl,praet,konj],'et').

endung(urg,[3,pl,praes,ind], 'en').
endung(urg,[3,pl,praes,konj],'en').
endung(urg,[3,pl,praet,ind], 'en').
endung(urg,[3,pl,praet,konj],'en').
```

⟨*Beispiel: unregelmäßige Verbflexion*⟩≡

```
?- ['Morphologie/flexion_slex',
    'Beispiele/stammformenlexikon'].
?- flexion_slex:flexion(aufgeben).

[1, sg, praes, ind]: aufgebe
[1, sg, praet, ind]: aufgab
[1, sg, praet, konj]: aufgäbe
[1, pl, praes, ind]: aufgeben
[1, pl, praet, ind]: aufgaben
[1, pl, praet, konj]: aufgäben
[2, sg, praes, ind]: aufgibst
[2, sg, praes, ind]: aufgibest
[2, sg, praet, ind]: aufgabst
[2, sg, praet, ind]: aufgabest
[2, sg, praet, konj]: aufgäbst
[2, sg, praet, konj]: aufgäbest
...
```

Prolog: Schreiben von Termen und Regeln

Einen Prolog-Term schreibt man (mit belegten Variablen) mit `write`|`writeq`

⟨*Schreiben eines Terms*⟩≡

```
?- A=a, B=b, write(p(A,'B')).    % Das Atom 'B'
p(a, B)                          % wird die Variable B !
?- A=a, B=b, writeq(p(A,'B')).   % bzw.
p(a, 'B')                        % bleibt das Atom 'B'
```

Ein aus konstanten und variablen, vom Programmablauf abhängenden Teilen zusammengesetztes Atom schreibt man mit `format(+Atom,+Termliste)`:

⟨*Schreiben eines Atoms aus Stücken*⟩≡

```
?- Z=3, format('Rufe p(~w,~w,~w) auf!',[a,'Y',Z]).
Rufe p(a,Y,3) auf!
```

Programmregeln werden mit `write` als Terme geschrieben. Als Regeln (mit `.` und lesbar formatiert) schreibt man sie mit:

⟨*Schreiben von Fakten*⟩≡

```
?- portray_clause(kopf(X,Y)).
kopf(A,B).
```

⟨*Schreiben von Regeln*⟩≡

```
?- portray_clause((kopf(X) :- rumpf(X,Y))).
kopf(A) :-
        rumpf(A, B).
```

Alle Schreibbefehle schreiben auf den jeweils aktuellen *Ausgabestrom*, was normalerweise `user` (der Bildschirm) ist. Mit

⟨*Schreiben auf eine Datei*⟩≡

```
    tell(<Dateiname>),    % Datei öffnen
    <Schreibbefehle>,
    told.                 % Datei schließen
```

kann man die Ausgabe auf eine Datei umlenken.

Vom Stammformen- zum Vollformenlexikon

Für jede Stammform werden die Vollformen gebildet und ein Vollformeintrag in eine Datei geschrieben:

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
erzeuge_vollformenlexikon(Ausgabedatei) :-
    ( tell(Ausgabedatei),
        lex(Stammform,Wortart,Art,Flexionsklasse),
        form(Wortart,Form),
        beuge(Wortart,Flexionsklasse,Stammform,Form,
              Vollform),
        Kategorie =.. [Wortart,Art,Form],
        portray_clause(wort(Stammform,Kategorie,Vollform)),
        fail  % naechste Form oder Stammform
    ; told,
        format(user,'Vollformen erzeugt und nach
                    ''~w'' geschrieben.\n', Ausgabedatei)
    ).
```

Anwendung

Mit dem Flexionsprogramm kann aus einem geladenen Stammformenlexikon ein Vollformenlexikon erzeugt werden:

⟨*Erzeugung des Vollformenlexikons*⟩≡

```
?- ['Beispiele/stammformenlexikon',  % liefert lex/4
    'Morphologie/flexion_slex'].
Warning: ...
% flexion_slex compiled into flexion_slex
Yes

?- erzeuge_vollformenlexikon('vollformen.test.pl').
Vollformen erzeugt und nach
   'vollformen.test.pl' geschrieben
Yes
```

Die erzeugte Datei muß i.a. von Hand nachkorrigiert werden!

Die entstandene Datei enthält die Einträge

⟨*Auzug aus vollformen.test.pl*⟩≡

```
wort(ablegen,
     v([nom,akk],[1,sg,praes,ind]), ablege).
...
wort(aufgeben,
     v([nom,akk],[1,sg,praes,ind]), aufgebe).
wort(aufgeben,
     v([nom,akk],[1,sg,praet,ind]), aufgab).
wort(aufgeben,
     v([nom,akk],[1,sg,praet,konj]), aufgäbe).
...
wort(heben,
     v([nom,akk],[2,pl,praet,konj]), höbet).
wort(heben,
     v([nom,akk],[3,sg,praes,ind]), hebt).
...
```

Vollformenlexikon erneuern

Unter `Anwendungen/Astronomie/` werden wir ein Stammformenlexikon angeben; dasselbe kann man für andere Anwendungen machen. Nach dem Ändern von `Anwendungen/<Anwendung>/stammformen.pl` erneuert das folgende Programm die zugehörigen Vollformen:

⟨*morphologie.pl*⟩≡

```
:- style_check(-discontiguous).


erzeuge_vollformen :-
    ensure_loaded('Morphologie/flexion_slex'),
    write('\nLade Anwendungen/<Verzeichnis>/stammformen.pl au
         <Verzeichnis><Punkt><Return>'),
    nl,read(Verz),
    concat_atom(['Anwendungen/',Verz,'/stammformen.pl'],Stamm
    concat_atom(['Anwendungen/',Verz,'/vollformen.pl'],Vollfo
    [Stammformen],
    erzeuge_vollformenlexikon(Vollformen).
```

Nomenflexion

Das Programm `erzeuge_vollformenlexikon/1` (S.56) bzw. dessen Prädikat `beuge/4` (S.34) braucht noch eine Definition der Nomenflexion.[a]

Zuerst muß die Datei `wortformen.pl` um die Definition der Formen der Nomen, Relationsnomen und Eigennamen erweitert werden:

⟨*Morphologie/wortformen.pl*⟩+≡

```
form(n,[Num,Kas]) :- numerus(Num), kasus(Kas).
form(rn,[Num,Kas]) :- numerus(Num), kasus(Kas).
form(en,[sg,Kas]) :- kasus(Kas).

kasus(X) :- member(X,[nom,gen,dat,akk]).
```

Die Nomenflexion (für `n`, `rn` und `en`) erfolgt durch `dekliniere/4`, das zur Bildung der Vollformen noch passende Endungstabellen braucht.

---

[a]Da wir keine Adjektive benutzen, lassen wir die Adjektivflexion aus.

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
% -------------- Nomenflexion ----------------
% dekliniere(+Flex.klasse,+NomSg,?[Num,Kas],-Vollform)
dekliniere(Deklinationsklasse,Stammform,
           Formmerkmale,Vollform) :-
    endung(Deklinationsklasse,Formmerkmale,Endung),
    concat(Stammform,Endung,Vollform).

nomenflexion(Deklinationsklasse,Stammform) :-
    form(n,Formmerkmale),
    dekliniere(Deklinationsklasse,Stammform,
               Formmerkmale,Vollform),
    write(Formmerkmale),write(': '),
    write(Vollform),nl,
    fail.
nomenflexion(_,_).
```

Deklinationstabellen für Nomina

Nomenflexionsklassen setzen wir aus Endungstabellen für die Deklination im Singular (s1 - s3) und solchen für die Deklination im Plural (p1 - p5) zusammen und nennen sie `(si,pj)`:

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
% endung(+Dekl.klasse,Formmerkmale,-Endung):

endung((Sg,Pl),[Num,Kas],Endung) :-
        ( endung(Sg,[Num,Kas],Endung)
        ; endung(Pl,[Num,Kas],Endung) ).

endung(s1, [sg,nom],'').   % bei mehrsilbigem Stamm mit
endung(s1, [sg,gen],'s').  % unbetonter Endsilbe;
endung(s1, [sg,dat],'').   % bei Vokal(+h)-Auslaut;
endung(s1, [sg,akk],'').   % bei Nominalisierungen
```

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
endung(s1e,[sg,nom],'').   % bei Auslauten s,ß,x,tsch,z;
endung(s1e,[sg,gen],'es'). % bei einsilbigen Nomen;
endung(s1e,[sg,dat],'').   % bevorzugt bei Auslaut sch,st
endung(s1e,[sg,akk],'').

endung(s2e,[sg,nom],'').   % bei Nomina mit
endung(s2e,[sg,gen],'en'). % konson. Auslaut
endung(s2e,[sg,dat],'en').
endung(s2e,[sg,akk],'en').

endung(s3, [sg,nom],'').   % alle Femina
endung(s3, [sg,gen],'').
endung(s3, [sg,dat],'').
endung(s3, [sg,akk],'').
```

⟨*Morphologie/flexion_slex.pl*⟩+≡

```
endung(p1, [pl,nom],'e').
endung(p1, [pl,gen],'e').
endung(p1, [pl,dat],'en').
endung(p1, [pl,akk],'e').

endung(p2, [pl,nom],'').
endung(p2, [pl,gen],'').
endung(p2, [pl,dat],'n').
endung(p2, [pl,akk],'').

endung(p3, [pl,nom],'n').    endung(p3e,[pl,nom],'en').
endung(p3, [pl,gen],'n').    endung(p3e,[pl,gen],'en').
endung(p3, [pl,dat],'n').    endung(p3e,[pl,dat],'en').
endung(p3, [pl,akk],'n').    endung(p3e,[pl,akk],'en').
```

Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon [erledigt]
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing) [ab jetzt]
- Semantik:
  1. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  2. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

Kontextfreie Grammatik

Ein kontextfreie Grammatik (CFG) beschreibt eine Sprache durch Regeln, die die 'Konstituentenstruktur' ausdrücken:

⟨*kontextfreie Regel*⟩≡

```
<Kat> -> <Wortform>                % lexikalische Regel
<Kat> -> <Kat1> <Kat2> ... <KatN> % syntakt. Regel
```

`<Kat>` ist eine syntaktische oder lexikalische (Ausdrucks-)*Kategorie*.

Die syntaktische Regel besagt, daß ein Ausdruck der Kategorie `<Kat>` aus aufeinander folgenden Ausdrücken der Kategorien `<Kat1>` bis `<KatN>` (seinen *Konstituenten*) bestehen kann.

Beispiel einer CFG

⟨*Beispiel einer CFG*⟩≡

```
s -> np vp            det -> der | das
np -> det n           n -> Programmierer | Programm
vp -> v | v np        v -> steht      v -> schrieb
```

Durch | wird eine Alternative angegeben; man kann die Alternativen auch in getrennten Regeln schreiben:

⟨*Gleichwertige Schreibweisen*⟩≡

```
% in zwei Regeln       in einer Regel
  vp -> v
  vp -> v np           vp -> v | v np
```

Ein Ausdruck entsteht durch Anwenden der Regeln:

⟨*Erzeugung von Ausdrücken*⟩≡

```
np -> det n -> der n -> der Programmierer
```

Prolog: Definite Clause Grammars (DCG)

In Prolog kann man eine CFG in einer nur minimal anderen Form angeben: Kategorien sind durch `,` getrennt, Alternativen durch `;`, und Wörter werden einelementige Listen `[<Atom>]`:

⟨*Beispiele/programmierer.pl*⟩≡

```
% DCG-Grammatik (Definite Clause Grammar)
s --> np, vp.         det --> [der] ; [das].
np --> det, n.        n --> ['Programmierer'] ; ['Programm'].
vp --> v ; v, np.     v --> [steht].  v --> [schrieb].
```

Eine DCG darf aber im allgemeinen

1. als Kategorien Prolog-Terme (statt Atome) haben,
2. auf der rechten Regelseite auch Terme der Form `{Prolog-Code}` und `[Atom|Atome]` haben,
3. die Terme der rechten Regelseite mit `(T,S)` und `(T;S)` schachteln.

Man kann z.B. Formmerkmale als Argumente der Kategorienterme angeben, oder eine Liste von Wörtern als eine Konstituente:

⟨*Beispiele/testDCG.pl*⟩≡

```
s(Temp) -->
    np, { Temp = praes ; Temp = praet }, vp(Temp).
vp(praes) --> [steht].
vp(praet) --> [stand,'Kopf'].
```

Prolog: DCG wird zu einem Programm

Prolog übersetzt eine DCG beim Laden in ein Programm.

- aus jeder Kategorie `<Kat>` wird ein Prädikat `<Kat>/2`
- aus jeder Grammatikregel wird eine Programm-Regel.
- das Prädikat `<Kat>(?Atomliste1,?Atomliste2)` trifft zu, wenn die Liste `Atomliste1` von Wörtern mit einem Ausdruck der Kategorie `<Kat>` beginnt und als Rest die Liste `Atomliste2` übrigbleibt.

Die Prädikate werden meist wie `<Kat>(+Eingabe,-Rest)` verwendet.

Genauer:

- aus jeder Kategorie `<Kat>/n` wird ein Prädikat `<Kat>/n+2` usw.

Ein *Text* ist eine Folge von Wörtern; in Prolog eine Liste von Atomen.

Warum werden Kategorien nicht *einstellige* Prädikate?

Wenn natürlichsprachliche Sätze in Prolog als Listen `[Wort1,Wort2,...]` dargestellt werden, sollte man erwarten, daß die grammatischen Kategorien `s`, `np` usw. durch *einstellige Prädikate* über Wortlisten interpretiert werden.

Die Grammatikregel `(s --> np vp)` wird dann ein Programm für `s/1`:

⟨*Grammatikregel als naives Prolog-Programm*⟩≡

```
s(Wortliste) :-
    append(Anfang,Ende,Wortliste),% zerlege Liste; prüfe
    np(Anfang), vp(Ende).         % Eigenschaften der Teile
```

Am Aufwand für `append(-Anfang,-Ende,+Wortliste)` kann man sparen, da die Konstituenten *aufeinanderfolgende* Teillisten der Eingabe sind. Die Kategorien sollten von einer Liste feststellen, ob ein Anfang die gewünschte Eigenschaft hat *und das Endstück weitergeben*:

⟨*Grammatikregel als effizienteres Prolog-Programm*⟩≡

```
s(Wortliste,[]) :-      % Wortliste=(Wortliste-[]) ist S
    np(Wortliste,Ende), % Anfang=(Wortliste-Ende) ist NP
    vp(Ende,[]).        % Ende = (Ende-[]) ist eine VP
```

Anwenden der Beispiel-DCG

⟨*Laden der DCG ergibt ein Programm mit Kategorien als Prolog-Prädikaten*⟩≡

```
?- ['Beispiele/programmierer.pl'].
% liefert: s(?Eingabe,?Rest), np(?Eingabe,?Rest), usw.
```

Das Prädikat `np/2` z.B. erkennt, ob die Eingabe mit einer Nominalphrase beginnt, und gibt den unverbrauchten Rest der Eingabe zurück:

⟨*Kategorien als Prolog-Prädikate*⟩≡

```
?- np(['Der','Programmierer',schrieb,das,'Programm'],
      Rest).
Rest = [schrieb,das,'Programm']
```

Anders gesagt, erkennt das Prädikat `np/2` die Nominalphase `['Der','Programmierer']` als *Differenz zweier Listen*

⟨*NP-Anfang der Liste als Differenz zweier Listen*⟩≡

```
np(['Der','Programmierer',schrieb,das,'Programm'],
                          [schrieb,das,'Programm']).
```

Prolog: Listen

In Prolog ist eine Liste ein Term der Form

```
[&lt;Element1&gt;,&lt;Element2&gt;,...]
```

Falls der Rest einer Liste unbekannt/variabel ist, schreibt man

```
[&lt;Element1&gt;,&lt;Element2&gt; | Restliste]
```

Wenn eine Liste `I` mit den Elementen `a` und `b` beginnt, und die Liste `J` der Rest ist, ist

⟨*Test auf Listengleichheit*⟩≡

```
I = [a,b|J].
```

Aus einer Liste kann man die Anfangselemente holen:

⟨*Entnehmen des ersten und dritten Listenelements*⟩≡

```
?- [a,b,c,d,e] = [X,_,Z|Rest]
X = a
Z = c
Rest = [d, e]
```

Eine Termzerlegung zeigt, daß Listen in Prolog folgende Terme sind:

die leere Liste: `[]`
nichtleere Listen: `'.'(Element,Restliste).`

⟨*Interne Listendarstellung*⟩≡

```
?- [a,b,c,d] =.. [Fun|Args]. % Term =.. [Fun|Args]
Fun = '.'                    % [a,b,c,d] =
Args = [a, [b, c, d]]        % '.'(a,[b,c,d])
```

Listenverkettung

⟨*Verkettung von Listen*⟩≡

```
% append(+Anfangsstück,+Reststück,?Gesamtliste).
% append(?Anfangsstück,?Reststück,+Gesamtliste).

append([a,b,c],[4,5,6,7],[a,b,c,4,5,6,7]).
```

Prolog: Übersetzte DCG

⟨*Prolog-Regeln nach dem Einlesen der DCG Beispiele/programmierer.pl*⟩≡

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

np(A, B) :-
        det(A, C),
        n(C, B).

vp(A, B) :-
        (   v(A, B)
        ;   v(A, C),
            np(C, B)
        ).
```

⟨*Prolog-Regeln nach dem Einlesen der DCG (Forts.)*⟩≡

```
v([steht|A], A).
v([schrieb|A], A).

n(A, B) :-
        (   'C'(A, 'Programmierer', B)
        ;   'C'(A, 'Programm', B)
        ).
det(A, B) :-
        (   'C'(A, der, B)
        ;   'C'(A, das, B)
        ).
```

`'C'(A,X,B)` ist gleichbedeutend mit `A = [X|B]`: Durch Einfügen von `X` in die Liste `B` entsteht die Liste `A`.

(Die Reihenfolge der Regeln zum selben Prädikat bleibt gleich.)

Prolog: Übersetzung von DCG-Regeln in Prolog-Regeln

Wie wird eine Grammatikregel in Prolog-Regeln übersetzt?

Die gleichen Variablen `I,J` für die Eingabe- und Ausgabeliste müssen zur Übersetzung beider Regelseiten benutzt werden:

⟨*Parser/addDifflists.pl*⟩≡

```
:- module(addDifflists,[]).
translate((Links --> Rechts),(Kopf :- Rumpf)) :-
      !, translate(Links, Kopf ,I,J),
         translate(Rechts,Rumpf,I,J).
translate(Term,Term).
```

Durch `!` verhindert man, daß Regeln auch vom sonst-Fall behandelt werden: Terme, die keine Grammatikregeln sind, bleiben unverändert.

Die Seiten einer Regel werden je nach ihrer Form übersetzt; die komplexen Formen sind zuerst zu behandeln:

⟨*Parser/addDifflists.pl*⟩$+\equiv$

```
translate((A,B),(Atr,Btr),I,J) :-    % Konjunktion
    !,translate(A,Atr,I,K),
    translate(B,Btr,K,J).
translate((A;B),(Atr;Btr),I,J) :-    % Disjunktion
    !,translate(A,Atr,I,J),
    translate(B,Btr,I,J).
translate([],(I = J),I,J) :- !.      % leeres Wort
translate([Atom],(I = [Atom|J]),I,J) :- !. % Wort
translate({Prolog},Prolog,I,I) :-    % Anweisungen
    !.
translate(Kategorie,Term,I,J) :-     % sonst: Kategorie
    Kategorie =.. [Funktor|Argumente],
    append(Argumente,[I,J],MehrArgumente),
    Term =.. [Funktor|MehrArgumente].
```

Prolog: Verändern von Ausdrücken beim Einlesen

Prolog verändert eingelesene Terme mit `term_expansion(+Term,-Neu)`.

Wir ändern das Prädikat, sodaß statt der in Prolog eingebauten Übersetzung unser `translate/2` benutzt wird:

⟨*Beispiele/term_expansion0.pl*⟩≡

```
:- use_module('Parser/addDifflists.pl').
term_expansion(Term,Expandiert) :-
    addDifflists:translate(Term,Expandiert).
```

Beispiel: Übersetzung mit `translate/2`

Nach dem Laden von `term_expansion0.pl` werden Terme etwas anders als von Prolog umgeformt. Die Unterschiede sind nur syntaktisch:

⟨*Übersetzung von Beispiele/programmierer.pl*⟩≡

```
% Unterschiede zur Übersetzung durch Prolog:
v(A, B) :-
        A=[steht|B].
v(A, B) :-
        A=[schrieb|B].
n(A, B) :-
        (   A=['Programmierer'|B]
        ;   A=['Programm'|B]
        ).
det(A, B) :-
        (   A=[der|B]
        ;   A=[das|B]
        ).
```

Verwendung von `{ Prolog }` in einer DCG

Um nach dem Parsen auch zu erfahren, was die *syntaktische Strukur* des am Anfang der Eingabe gefundenen Ausdrucks ist, bauen wir in die Kategorien ein Ausgabeargument für den Syntaxbaum ein, in der (vorläufigen)

⟨*Darstellung eines Baums durch einen Term*⟩≡

```
Baum = Wurzel(Teilbaum1,...,TeilbaumN)
```

⟨*Beispiele/baum.term.pl*⟩≡

```
%% np --> det, n.
np(Baum) --> det(BaumDet), n(BaumN),
     { Baum = np(BaumDet,BaumN) }.

%% det --> [der] ; [das].
det(Baum) --> [der], { Baum = det(der) } ;
              [das], { Baum = det(das) }.
```

⟨*Beispiele/baum.term.pl*⟩+≡

```
%% n --> ['Programmierer'] ; ['Programm'].
n(Baum) -->
     ['Programmierer'], { Baum = n('Programmierer') }
     ; ['Programm'], { Baum = n('Programm') }.
```

Ein Aufruf hat die Form `np(-Baum,+Eingabeliste,-Rest)`:

⟨*Beispiel einer Analyse*⟩≡

```
?- np(Baum, [der,'Programmierer',schrieb], Rest).

Baum = np(det(der), n('Programmierer'))
Rest = [schrieb]
```

Baumdarstellung durch Listen

Ist die Wurzel ein komplexer Term, kein Atom, so ist die

⟨*Darstellung eines Baums durch eine Liste*⟩≡

```
Baum = [Wurzel, Teilbaum1,...,TeilbaumN]
```

besser, da die Wurzel(kategorie) ein Teilterm des Baums wird:

⟨*Vergleich der Baumdarstellungen*⟩≡

```
Baum als Liste = [n([fem],[sg,akk]),B1,...,BN] % n/2
Baum als Term  =  n([fem],[sg,akk], B1,...,BN) % n/2+N
```

⟨*Beispiele/baum.liste.pl*⟩≡

```
%% np --> det, n.
np(Baum) --> det(BaumDet), n(BaumN),
    { Baum = [np, BaumDet,BaumN] }.

%% det --> [der] ; [das].
det(Baum) --> [der], { Baum = [det, [der]] } ;
              [das], { Baum = [det, [das]] }.
```

⟨*Beispiele/baum.liste.pl*⟩+≡

```
%% n --> ['Programmierer'] ; ['Programm'].
n(Baum) -->
     ['Programmierer'], { Baum = [n, ['Programmierer']] }
     ; ['Programm'], { Baum = [n, ['Programm']] }.
```

Der Syntaxbaum ist nun eine Liste von Listen:

⟨*Beispiel einer Analyse*⟩+≡

```
?- np(Baum, [der,'Programmierer'], _).
Baum = [np, [det, [der]], [n, ['Programmierer']]]
```

Automatische Ergänzung einer Baumausgabe

Wir können das Anfügen eines Baumausgabearguments durch eine Übersetzung Quell-DCG $\mapsto$ Ziel-DCG automatisieren.

1. Kopf und Rumpf einer Regel werden durch unterschiedliche Übersetzungen in Prolog-Terme oder Ziele übersetzt:

   $\langle$*Parser/addTree.pl*$\rangle\equiv$

   ```
   :- module(addTree,[]).

   translate((L --> R),(Ltr --> Rtr)) :-
       !, translate1(L,Ltr,Baum),
       translate2(R,Rtr,Baumliste),
       Baum = [L|Baumliste].
   translate(Term,Term). % nur Regeln ändern
   ```

   `Ltr` enthält die Variable `Baum`, die im Nachhinein belegt wird.

2. Eine Kategorie (auch mit Merkmalen) erhält ein Argument zur Ausgabe eines Baums:

   ⟨*Parser/addTree.pl*⟩+≡

```
% translate1(+Kategorie,-Term,?Baum)
translate1(Kategorie,Term,Baum) :-
      Kategorie =.. [Funktor|Argumente],
      append(Argumente,[Baum],MehrArgumente),
      Term =.. [Funktor|MehrArgumente].
```

3. Rechte Seiten (und ihre Teile) erhalten ein Argument zur Ausgabe einer Baum*liste*.

   ⟨*Parser/addTree.pl*⟩+≡

```
% translate2(+Rechts,-RechtsTr,-Baumliste)
```

   Da z.B. bei `(B;(C,D))` die Anzahl der Teilbäume von der Alternative beim Parsen abhängt, kann man nur *Variable* für Baumlisten und `{Code}`-Instruktionen zu ihrem Aufbau in die Ziel-DCG einfügen.

⟨*Parser/addTree.pl*⟩+≡

```
translate2((A;B),(Atr;Btr),Baeume) :- % Baeume ist Var!
        !, translate2(A,Atr0,BaeumeA),
           translate2(B,Btr0,BaeumeB),
        Atr = (Atr0,{ Baeume=BaeumeA }),
        Btr = (Btr0,{ Baeume=BaeumeB }).
translate2((A,B),Tr,Baeume) :-
        !, translate2(A,Atr,BaeumeA),
           translate2(B,Btr,BaeumeB),
        (fixed_length(BaeumeA) % s.u.
        -> Tr = (Atr,Btr), % jetzt verketten:
             append(BaeumeA,BaeumeB,Baeume)
        ;  Tr = (Atr,Btr,  % erst beim Parsen verketten:
                    {append(BaeumeA,BaeumeB,Baeume)})
        ).
```

Ist `BaeumeA` von der Form `Var` oder `[B1,...|Var]`, so wird `Var` durch `append` belegt (Fehler); dann darf man erst beim Parsen verketten.

⟨*Parser/addTree.pl*⟩+≡

```
translate2([],[],[]) :- !.
translate2([Atom|As],[Atom|As],[[Atom]|Baeume]) :-
    !, translate2(As,As,Baeume).
translate2({Prolog},{Prolog},[]) :- !.
translate2(Cat,CatTr,[Baum]) :-
    !, translate1(Cat,CatTr,Baum).

fixed_length(L) :-
    var(L),!, fail. % L nicht belegen!
fixed_length([]).
fixed_length([_X|Xs]) :-
    fixed_length(Xs).
```

Man braucht `{Code}` bei `(A,B)` für Regeln wie `e --> (a,b ; c),d.`

Übersetzung: DCG $\mapsto$ DCG mit Baumausgabe

Damit beim Einlesen nur die Baumausgabe in den DCG-Regeln ergänzt wird, definieren wir `term_expansion/2` so:

⟨*Beispiele/term_expansion1.pl*⟩≡

```
:- use_module(['Parser/addTree']).
term_expansion(Term,Expandiert) :-
    addTree:translate(Term,Expandiert).
```

Lädt man zuerst diese Datei und dann eine DCG-Datei, so werden daraus neue DCG-Regeln, in denen auch der Aufbau von Syntaxbäumen definiert wird.

Aus einer Kategorie (ohne Merkmale) wie `np` wird das Prolog-Prädikat `np(?Baum)` mit einem Argument zur Baumausgabe.

Achtung: Die Zielgrammatik ist aber nicht verwendbar, da `-->` nicht als Prolog-Prädikat definiert ist.

Beispiel: Übersetzung in DCG mit Baumausgabe

Wir übersetzen die früher schon benutzte Grammatik:

```
s --> np, vp.
vp --> v ; v, np.

np --> det, n.
det --> [der] ; [das].
n --> ['Programmierer'] ; ['Programm'].

v --> [steht].
v --> [schrieb].
```

⟨*Laden der Übersetzung und der Quell-Grammatik*⟩≡

```
?- ['Beispiele/term_expansion1',
    'Beispiele/programmierer'].
Yes
```

Mit `?- listing.` erhält man dann die übersetzte Grammatik:

⟨*Beispiele/programmierer.baumausgabe.pl*⟩≡

```
s([s, A, B]) --> np(A), vp(B).
det([det|A]) -->  [der], {A=[[der]]}
                ; [das], {A=[[das]]}.
np([np, A, B]) --> det(A), n(B).
n([n|A]) --> ['Programmierer'], {A=[['Programmierer']]}
           ; ['Programm'], {A=[['Programm']]}.
vp([vp|A]) --> v(B), {A=[B]}
             ; (v(C), np(D)), {A=[C, D]}.
v([v, [steht]]) --> [steht].
v([v, [schrieb]]) --> [schrieb].
```

Ihre Kategorien enthalten die auszugebenden Syntaxbäume, soweit sie vor dem Parsen bekannt sind.

Bemerkung: `{append(BäumeA,BäumeB,Bäume)}` in `translate2` braucht man für Regeln, in denen eine Disjunktion unterhalb einer Konjunktion auftritt, z.B.

$\langle$*Übersetzung der Regel (np –> (det ; det, a), n)*$\rangle\equiv$

```
np([np|A]) --> ( det(B), {C=[B]}
               ; (det(D), a(E)), {C=[D, E]}
               ),
               n(F),
               {append(C, [F], A)}.
```

Würde `append(C,[F],A)` bei der Grammatikübersetzung ausgeführt, so würde `C=[],[F]=A` und es entstünden die unlösbaren Bedingungen `{[]=[B]}` und `{[]=[D,E]}`, womit die übersetzte Regel beim Parsen unbrauchbar wäre.

Beispiel: Laden der übersetzten Grammatik

Beim Laden der übersetzten Grammatik ergänzt Prolog noch die Differenzlistenargumente:

⟨*DCG mit Baumausgabe in Prolog-Programm übersetzt:*⟩≡

```
?- ['Beispiele/programmierer.baumausgabe.pl'].
?- listing.

np([np, A, B], C, D) :-
        det(A, C, E),
        n(B, E, D).
det([det|A], B, C) :-
        (   'C'(B, der, D),
            A=[[der]],
            C=D
        ;   'C'(B, das, E),
            A=[[das]],
            C=E
        ).
```

⟨*DCG mit Baumausgabe in Prolog-Programm übersetzt, Forts.*⟩≡

```
s([s, A, B], C, D) :-
        np(A, C, E),
        vp(B, E, D).
vp([vp|A], B, C) :-
        (   v(D, B, E),
            A=[D],
            C=E
        ;   v(F, B, G),
            np(H, G, I),
            A=[F, H],
            C=I
        ).
v([v, [steht]], [steht|A], A).
v([v, [schrieb]], [schrieb|A], A).
```

⟨*DCG mit Baumausgabe in Prolog-Programm übersetzt, Forts.*⟩+≡

```
n([n|A], B, C) :-
        (   'C'(B, 'Programmierer', D),
            A=[['Programmierer']],
            C=D
        ;   'C'(B, 'Programm', E),
            A=[['Programm']],
            C=E
        ).
```

Mit diesem Programm erhält man die Syntaxbäume in Listendarstellung:

⟨*Analyse einer Nominalphrase am Anfang der Eingabe*⟩≡

```
?- np(Baum,
      [der,'Programmierer',schrieb,das,'Programm'],
      Rest).
Baum = [np, [det, [der]], [n, ['Programmierer']]]
Rest = [schrieb, das, 'Programm']
```

Übersetzung: DCG $\mapsto$ Prolog mit Baumausgabe

Brauchbar ist die automatische Ergänzung der Baumausgabe erst, wenn man die Ziel-DCG durch Anfügen der Ein-Ausgabe-Listen in Prolog-Regeln weiterübersetzt:

$\langle$*Parser/term_expansion.pl*$\rangle\equiv$

```
:- use_module([addDifflists,addTree]).

term_expansion(Term,Expandiert) :-
    addTree:translate(Term,ExpTerm),
    addDifflists:translate(ExpTerm,Expandiert).
```

Bem.: Hier werden zwei `translate/2` aus verschiedenen Modulen benutzt.

Lädt man zuerst `Parser/term_expansion.pl` und dann eine DCG-Datei, so wird die Grammatik in Prolog-Regeln übersetzt, die an die Kategorien ein Argument für die Baumausgabe und die Argumente für die Ein- und Ausgabelisten von Wörtern anfügen.

Aus einer Kategorie `np` wird das Prolog-Prädikat

```
np(?Baum,+Eingabewoerter,?Restwoerter).
```

Aus einer Kategorie `np(Genus)` wird das Prolog-Prädikat

```
np(Genus,?Baum,+Eingabewoerter,?Restwoerter).
```

Beispiel: Übersetzung in Prolog+Baumausgabe

Unsere Übersetzung liefert ein leicht einfacheres Ergebnis als das mit der Differenzlistenergänzung durch Prolog:

$\langle$*Ergebnis der Übersetzung DCG* $\mapsto$ *Prolog+Baumausgabe*$\rangle\equiv$

```
?- ['Parser/term_expansion','Beispiele/programmierer'].
?- listing.

s([s, A, B], C, D) :-
        np(A, C, E),
        vp(B, E, D).
np([np, A, B], C, D) :-
        det(A, C, E),
        n(B, E, D).
det([det|A], B, C) :-
        (   B=[der|C], A=[[der]]
        ;   B=[das|C], A=[[das]]
        ).
```

⟨*Ergebnis der Übersetzung DCG ↦ Prolog+Baumausgabe, Forts.*⟩≡

```
vp([vp|A], B, C) :-
        (   v(D, B, C),
            A=[D]
        ;   v(E, B, F),
            np(G, F, C),
            A=[E, G]
        ).
v([v, [steht]], A, B) :-
        A=[steht|B].
v([v, [schrieb]], A, B) :-
        A=[schrieb|B].
n([n|A], B, C) :-
        (   B=['Programmierer'|C],
            A=[['Programmierer']]
        ;   B=['Programm'|C],
            A=[['Programm']]
        ).
```

Beispiel: Grammatik mit Merkmalen

⟨*Beispiele/dcg.merkmale.pl*⟩≡

```
% DCG-Grammatik (Definite Clause Grammar)
% Nur Pers = 3, daher ignoriert.

startsymbol(s([_Temp])).  % für später

s([Temp]) -->
    np([_Gen],[Num,nom]), vp([Temp,Num]).
np([Gen],[Num,Kas]) -->
    det([Gen,Num,Kas]), n([Gen],[Num,Kas]).
vp([Temp,Num]) -->
    ( v([Temp,Num])
    ; v([Temp,Num]), np([_Gen],[_Num,akk])
    ).
```

⟨*Beispiele/dcg.merkmale.pl*⟩+≡

```
det([Gen,sg,Kas]) -->
     ( [der], { Gen = mask, Kas = nom }
     ; [das], { Gen = neut, (Kas = nom; Kas = akk) }
     ).
n([mask],[Num,Kas]) -->
     ['Programmierer'], { (Num = sg ; Num = pl),
                          (Kas = nom; Kas = akk) }.
n([neut],[sg,Kas]) -->
     ['Programm'], { (Kas = nom; Kas = akk) }.

v([praes,sg]) --> [steht].
v([praet,sg]) --> [schrieb].
```

⟨*Übersetzen und Anwenden der Grammatik mit Merkmalen*⟩≡

```
?- ['Parser/term_expansion.pl',
     'Beispiele/dcg.merkmale.pl'], listing.
...
s([A], [s([A]), B, C], D, E) :-
        np([F], [G, nom], B, D, H),
        vp([A, G], C, H, E).


?- s([Temp],Baum,
      [der,'Programmierer',schrieb,das,'Programm'],
      Rest).
Temp = praet
Baum = [s([praet]),
        [np([mask], [sg, nom]),
         [det([mask, sg, nom]), [der]],
         [n([mask], [sg, nom]), ['Programmierer']]],
        [vp([praet, sg]),
         [v([praet, sg]), [schrieb]]]]
Rest = [das, 'Programm'] ;
```

⟨*Übersetzen und Anwenden der Grammatik mit Merkmalen, Forts.*⟩≡

```
Temp = praet
Baum = [s([praet]),
        [np([mask], [sg, nom]),
         [det([mask, sg, nom]), [der]],
         [n([mask], [sg, nom]), ['Programmierer']]],
        [vp([praet, sg]),
         [v([praet, sg]), [schrieb]],
         [np([neut], [sg|...]),
          [det(...)|...], [...|...]]]]
Rest = [] ;

No
?-
```

Formatierte Baumausgabe

Um einen in der Listendarstellung gegebenen Baum lesbar auszugeben, schreibt man die Teilbäume, um drei Leerzeichen gegenüber der Wurzel eingerückt, in neue Zeilen untereinander:

⟨*Parser/showTree.pl*⟩≡

```
showTree([Wurzel|Bs],Einrueckung) :-
    tab(Einrueckung), writeq(Wurzel),
    (Bs = [[Blatt]] -> tab(1), writeq(Blatt)
    ; Einrueckung2 is Einrueckung + 3,
        showTrees(Bs,Einrueckung2)).
showTree([],Einrueckung) :-
    tab(Einrueckung), writeq([]).

showTrees([Baum|Baeume],Einrueckung) :-
    nl,showTree(Baum,Einrueckung),
    showTrees(Baeume,Einrueckung).
showTrees([],_).
```

⟨*Beispiel zur Baumanzeige*⟩≡

```
?- showTree([np,[det,[das]],
                [ap,[grad,[sehr]],[a,[kurze]]],
                [n,['Programm']]],6).
      np
         det das
         ap
            grad sehr
            a kurze
         n 'Programm'
```

Graphische Baumausgabe

Da die Termdarstellung des Syntaxbaums bei größeren Ausdrücken unleserlich ist, fügen wir noch eine graphische Darstellung hinzu.

Sie benutzt das Programm `graphviz/dot`.

Die Moduldatei (nicht auf diesen Folien)

`Parser/dot.syntaxbaum.pl`

exportiert das Prädikat `displayTrees/1`.

Durch `displayTrees(Trees)` werden die Syntaxbäume

1. im `.dot`-Format in die Datei `testbaum.tmp.dot` geschrieben,
2. daraus mit `/bin/dot` eine Postscriptdatei `testbaum.tmp.ps` erzeugt,
3. und diese mit `/usr/X11R6/bin/gv` am Bildschirm gezeigt.

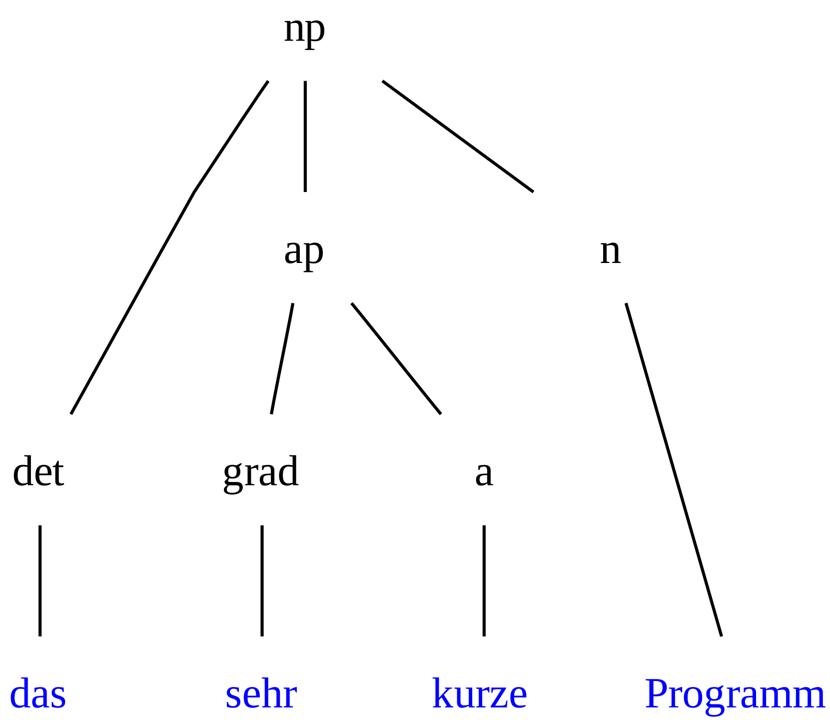
Diese beiden Dateien werden bei der nächsten Analyse überschrieben.

Bem. Das Programm `Parser/dot.syntaxbaum.pl` kann vereinfacht werden; es war für allgemeinere Bäume gedacht.

Anzeige von `testbaum.tmp.ps`

⟨*Graphische Baumausgabe*⟩≡

```
  ?- displayTree([np,[det,[das]],
                     [ap,[grad,[sehr]],[a,[kurze]]],
                     [n,['Programm']]]).
```



Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon [erledigt]
- Tokenisierung und lexikalische Analyse eines Texts [jetzt]
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing) [erledigt]
- Semantik:
  1. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  2. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

Vereinfachung der Satzeingabe

Um Sätze nicht umständlich als Listen von Prolog-Atomen einzugeben, sondern einfach als Zeichenreihen, müssen wir

1. von der Konsole Zeichen bis zu einem Satzende lesen,
2. Zeichenreihen in Listen von Prolog-Atomen umwandeln,
3. die Atomliste der Syntaxanalyse übergeben

können.

Prolog: Zeichenreihen

In Prolog wird ein Zeichen durch seine ASCII-Nummer und eine Zeichenreihe durch die Liste der ASCII-Nummern ihrer Zeichen dargestellt. Zeichenreihen kann man bequem *eingeben*:

⟨*Eingabe von Zeichenreihen*⟩≡

```
?- Codes = "mein Bier".
Codes = [109, 101, 105, 110, 32, 66, 105, 101, 114]
```

Man arbeitet nicht direkt mit den Code-Nummern, sondern wandelt diese um.

Zeichenreihen werden mit `name/2` in Atome umgewandelt:

⟨*Umwandlung einer Liste von Code-Nummern in ein Atom*⟩≡

```
?- name(Atom,[101, 105, 110, 32, 66, 105, 101, 114]).
Atom = 'ein Bier'
?- name('dein Bier',Codes).
Codes = [100, 101, 105, 110, 32, 66, 105, 101, 114]
```

Zeichen werden mit `char_code/2` in Code-Nummern umgewandelt:

⟨*Umwandlung einer Codenummer in ein Atom*⟩≡

```
?- char_code(Atom,97).
Atom = a
?- char_code(b,Code).
Code = 98
```

1. Lesen bis zum Satzende

Wir lesen Zeichen von einem Strom `Stream` (Konsole oder Datei), sammeln die schon gelesenen in einer Liste `Seen`, und geben den ersten Satz als Zeichenreihe `Sentence` aus.

Mit `get0` wird ein Zeichen gelesen und der Strom verkürzt:

⟨*Parser/tokenizer.mini.pl*⟩≡

```
:- module(tokenizer,[read_sentence/3,tokenize/2,
                     satzende/0,parse/0,parsed/0]).

% read_sentence(+Stream,-Sentence,+Seen)
read_sentence(Stream,Sentence,Seen) :-
     get0(Stream,Char),
     read_sentence(Stream,Sentence,Char,Seen).
```

An den beiden letzten Zeichen wird erkannt, ob das Satzende erreicht ist. Wenn ja, ist der Satz die Umkehrung der gesammelten Zeichen; wenn nein, wird weitergelesen:

⟨*Parser/tokenizer.mini.pl*⟩+≡

```
read_sentence(Stream,Sentence,Char,Seen) :-
     (Char = -1  % Dateiende
     -> reverse(Seen,Sentence)
     ; (Seen = [C|Cs], satzende([C,Char]))
     -> reverse(Cs,Sentence)
     ; Char = 10 % return ignorieren
     -> read_sentence(Stream,Sentence,Seen)
     ; read_sentence(Stream,Sentence,[Char|Seen])).
```

Als (nicht ausgegebenes) Satz- bzw. Eingabeende gelte `,,.<Return>‘‘`:

⟨*Parser/tokenizer.mini.pl*⟩+≡

```
satzende([46,10]).  % <Punkt><Return>
```

2. Zerlegung einer Zeichenreihe in Atome

Eine gelesene Zeichenreihe muß nun in eine Folge von *Token* umgewandelt werden, der Eingabe für die Syntaxanalyse.

Die Token sind hier Prolog-Atome. Wir wandeln die Zeichenreihe in ein Atom um und spalten dieses an den (einzelnen!) Leerzeichen:

$\langle$*Parser/tokenizer.mini.pl*$\rangle+\equiv$

```
% tokenize(+String,-Atomlist).
tokenize(String,Atomlist) :-
     name(Atom,String),
     concat_atom(AtomlistK,' ',Atom),
     entferne_komma(AtomlistK,Atomlist).
```

Wenn mehrere Leerzeichen wir ein einziges gelten sollen, muß man doppelte Leerzeichen in `read_sentence/4` ignorieren.

Damit Kommata (ohne vorangehendes Leerzeichen) nicht als Buchstabe des Worts behandelt werden, löschen wir sie:

⟨*Parser/tokenizer.mini.pl*⟩+≡

```
entferne_komma([AtomK|AtomeK],[Atom|Atome]) :-
     (atom_concat(Atom,',',AtomK)
     -> true
     ; Atom = AtomK),
     entferne_komma(AtomeK,Atome).
entferne_komma([],[]).
```

Dann können Relativsätze in der Eingabe mit Komma abgetrennt werden. (In der Grammatik werden keine Kommata berücksichtigt.)

Test zur Erkennung des Satzendes

Als Test fordern wir den Benutzer zu einer Eingabe auf, wandeln diese in eine Zeichenreihe `Sentence`, ein Atom `Satz`, und eine Liste von Atomen `Atomlist` um, und zeigen `Satz` und `Atomlist` am Bildschirm:

⟨*Parser/tokenizer.mini.pl*⟩+≡

```
satzende :-
     write('Beende die Eingabe mit <Punkt><Return>'),
     nl,read_sentence(user,Sentence,[]),
     name(Satz,Sentence),
     nl,write('Eingabe:   '),writeq(Satz),
     tokenize(Sentence,Atomlist),
     nl,write('Atomliste: '),writeq(user,Atomlist).
```

Nach Laden der Datei erhält man z.B.:

⟨*Erkennen des Eingabeendes*⟩≡

```
?- satzende.
Beende die Eingabe mit <Punkt><Return>
|: Kommt Prof. Klug am 3.12. zur Feier ?.

Eingabe:   'Kommt Prof. Klug am 3.12. zur Feier ?'
Atomliste: ['Kommt', 'Prof.', 'Klug', am,
            '3.12.', zur, 'Feier', ?]
```

Satzzeichen (und Zeilenumbrüche) werden als Wortteile behandelt, wenn sie nicht durch Leerzeichen abgetrennt wurden.

Da eine DCG mit den lexikalischen Regeln `Cat --> [Wort]` auch das Lexikon enthält, ist die *lexikalische Analyse* ein Teil der Syntaxanalyse.

3. Übergabe der Atomliste an die Syntaxanalyse

Die Grammatik muß hierfür ein `startsymbol(Kat)` festlegen.

⟨*Parser/tokenizer.mini.pl*⟩+≡

```
parse :-
     write('Beende die Eingabe mit <Punkt><Return>'),
     nl,read_sentence(user,Sentence,[]),
     tokenize(Sentence,Atomlist),
     startsymbol(S),
     addTree:translate1(S,Term,Baum),
     addDifflists:translate(Term,TermExp,Atomlist,[]),
     nl,write('Aufruf: '), portray_clause(TermExp),
     call(TermExp),
     write('Baum:   '), nl,showTree(Baum,8),nl,
     fail.
parse.
```

⟨*Anwendung auf die Beispielgrammatik*⟩≡

```
?- ['Parser/term_expansion','Beispiele/dcg.merkmale',
    'Parser/tokenizer.mini','Parser/showTree'].
?- parse.          % startsymbol(s([_Temp])).
Beende die Eingabe mit <Punkt><Return>
|: das Programm steht.

Aufruf: s([A], B, [das, 'Programm', steht], []).
Baum:
        s([praes])
           np([neut], [sg, nom])
              det([neut, sg, nom]) das
              n([neut], [sg, nom]) 'Programm'
           vp([praes, sg])
              v([praes, sg]) steht
```

Parse-Aufruf mit graphischer Baumdarstellung

Bei größeren Ausdrücken ist die graphische Baumdarstellung übersichtlicher. (Dazu muß das Programm `graphviz/dot` installiert sein.)

$\langle$*Parser/tokenizer.mini.pl*$\rangle{+}{\equiv}$

```
parsed :-
    write('Beende die Eingabe mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    setof(Baum, parse(Atomlist, Baum), Trees),
    displayTrees(Trees).

parse(Atomlist,Baum) :-
    startsymbol(S),
    addTree:translate1(S,Term,Baum),
    addDifflists:translate(Term,Exp,Atomlist,[]),
    call(Exp).
```

Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing)
- Semantik:
  1. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  2. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

Anwendung: Datenbankabfrage

Wir wollen in einer Datenbank mit Hilfe natürlich-sprachlicher Fragen Informationen suchen und die Ergebnisse in natürlicher Sprache formulieren.

Das Anwendungsbeispiel ist eine Datenbank über Astronomen und die von ihnen entdeckten Himmelskörper.

⟨*Form der Datenbankeinträge*⟩≡

```
% db(Himmelskoerper,Art,Durchmesser,
%    Entdecker,UmkreisterHimmelskörper)
```

Wir unterscheiden die Arten Sonne, Planet und Mond.

Fragen wie die folgenden sollen gestellt und beantwortet werden können:

1. Welche Monde entdeckte Galilei?
2. Welcher Astronom, der einen Planeten entdeckte, entdeckte einen Mond?
3. Welchen Planeten, den ein Astronom entdeckte, umkreist ein Mond?
4. Entdeckte jeder Astronom einen Planeten?
5. Ist Uranus ein Planet, der die Sonne umkreist?
6. Ist der Durchmesser von Jupiter kleiner als der Durchmesser von Uranus?
7. Welchen Planten, der einen Durchmesser besitzt, der kleiner als 50000 km ist, entdeckte Herschel?

Semantische Beschränkungen

Die Nomen und Verben der Anwendung erlauben als Argumente nur Objekte `X` bestimmter Typen (geschrieben `X:Typ`), z.B.:

1. `planet von (Y:stern)` $\subseteq$ `stern`
2. `mond von (Y:stern)` $\subseteq$ `stern`
3. `durchmesser von (Y:stern)` $\subseteq$ `groesse`

1. `(X:astronom) entdeckte (Y:stern),`
2. `(X:stern) umkreist (Y:stern),`
3. `(X:name) ist ein (Y:astronom/Y:sternart),`
4. `(X:stern) ist ein (Y:sternart),`
5. `(X:stern) ist ein planet von (Y:stern),`
6. `(X:stern) hat (Y:groesse) als durchmesser,`
7. `(X:groesse) ist kleiner als (Y:groesse).`

Solche semantischen Beschränkungen der Argumente werden in der Grammatik nicht berücksichtigt. Wir prüfen sie erst an der logischen Formel (vgl. `typisiere/2`).

Entwicklung der Grammatik

Welche syntaktischen Kategorien braucht man dafür?

1. verschiedene Arten von Nominalphrasen: `def`inite, interrogative (`qu`), `rel`ativierende und `quant`ifizierte,
2. verschieden Arten von Verben: Hilfsverben und transitive Vollverben,
3. verschiedene Arten von Sätzen: Aussagesätze (`def`) für Antworten, Relativsätze (`rel`) und Fragen (`qu`).

Normierung: die Kategorien haben stets die Form

⟨*Format der Kategorienbezeichnung*⟩≡

```
<Name>([<Artmerkmal>,...],[<Formmerkmal>,...])
```

Zur Fehlervermeidung beim Erstellen der Grammatikregeln ist

- eine genaue Definition der erlaubten Kategorien und
- eine genaue Definition der erlaubten Merkmalwerte

nützlich.

Diese Prolog-Definitionen in `Grammatik/kategorien.pl` werden hier nur als Dokumentation verwendet, nicht für eine Korrektheitsprüfung der Grammatikregeln.

Lexikalische Kategorien

Die Wortarten sollen folgende Namen und Merkmale haben:

⟨*Grammatik/kategorien.pl*⟩≡

```
:- module(kategorien,[kategorie/1]).

:- use_module('Morphologie/wortformen.pl').
genus(Gen) :-
     (Gen = fem ; Gen = mask ; Gen = neut).
kasus(X) :-
     (X = nom ; X = gen ; X = dat ; X = akk).

kategorie(en([Gen],[sg,Kas])) :-
     genus(Gen),
     kasus(Kas).
```

$\langle$*Grammatik/kategorien.pl*$\rangle+\equiv$

```
kategorie(n([Gen],[Num,Kas])) :-
    genus(Gen),
    numerus(Num),  % wortformen:numerus/1
    kasus(Kas).

kategorie(rn([Gen],[Num,Kas])) :-
    genus(Gen),
    numerus(Num),
    kasus(Kas).

kategorie(pron([Def],[Gen,Num,Kas])) :-
    (Def = def ; Def = rel ; Def = qu),
    genus(Gen),          % er, der, wer
    numerus(Num),
    kasus(Kas).
```

⟨*Grammatik/kategorien.pl*⟩+≡

```
kategorie(det([Def],[Gen,Num,Kas])) :-
    member(Def,[indef,def,qu,quant]),
    genus(Gen),           % ein,der,welcher,jeder
    numerus(Num), kasus(Kas).

kategorie(poss([Def],[Gen,Num,Kas])) :-
    member(Def,[def,rel,qu]),
    genus(Gen),           % sein,dessen,wessen
    numerus(Num), kasus(Kas).

kategorie(v([nom,akk],[Pers,Num,Temp,Mod])) :-
    Pers = 3,
    numerus(Num),
    tempus(Temp),  % wortformen:tempus/1
    modus(Mod).    % wortformen:modus/1
```

⟨*Grammatik/kategorien.pl*⟩+≡

```
kategorie(v([H],[Pers,Num,Temp,Mod])) :-
     hilfsverb(H),
     Pers = 3,
     numerus(Num),
     tempus(Temp),
     modus(Mod).
hilfsverb(H) :-
     (H = sein ; H = werden).  % ggf. haben
```

Startsymbole der Grammatik

Die *Startsymbole* einer Grammatik sind ihre Hauptkategorien. Die Syntaxanalyse (vgl. `parse/0`) versucht, die Eingabe (nur) als einen Ausdruck einer solchen Kategorie zu erkennen.

⟨*Grammatik/startsymbole.pl*⟩≡

```
startsymbol(np([_Def,_Pers,_Gen],[_Num,_Kas])).
startsymbol(s([_Def],[_Temp,ind,_Vst])).
```

Diese Hauptkategorien werden unten definiert.

Was muß man laden?

Schrittweise entwickeln wir nun Grammatikregeln für Nominalphrasen und Sätze, und dazu passende Lexkioneinträge.

Durch Laden der Datei

⟨*grammatiktest.pl*⟩≡

```
  :- ['Parser/tokenizer.mini.pl',
      'Parser/term_expansion.pl',
      'Parser/showTree.pl',
      'Parser/dot.syntaxbaum.pl',
      'Grammatik/startsymbole.pl',
      'Grammatik/np.pl',
      'Grammatik/saetze.pl',
      'Beispiele/testlexikon.pl'
      ].
```

werden die erforderlichen (wachsenden) Dateien geladen.

Später ersetzen wir das Testlexikon durch ein endgültiges.

Nominalphrasen

Kategorie NP

⟨*Grammatik/kategorien.pl*⟩+≡

```
kategorie(np([Def,Pers,Gen],[Num,Kas])) :-
    definitheit(np,Def),
    genus(Gen),
    numerus(Num),
    kasus(Kas),
    Pers = 3. % wir behandeln nur dritte Person

definitheit(np,Def) :-
    ( Def = indef ; Def = def ; Def = qu ; Def = quant
    ; Def = rel(Gen,Num), genus(Gen), numerus(Num) ).
```

Nominalphrasen 1

⟨*Grammatik/np.pl*⟩≡

```
np([Def,3,Gen],[Num,Kas]) -->
     det([Def],[Gen,Num,Kas]),
     n([Gen],[Num,Kas]),
     { member(Def,[indef,def,qu,quant]) }.

np([Def,3,Gen],[Num,Kas]) -->
     pron([DefP],[Gen,Num,Kas]),
     { DefP = rel, Def = rel(Gen,Num)
     ; DefP = qu, Def = qu }.

np([def,3,Gen],[Num,Kas]) -->
     en([Gen],[Num,Kas]).
```

Bem.: Aus Effizienzgründen fehlen Nebenbedingungen, die die freien Variablen auf (endliche) Merkmalbereiche beschränken.

Lexikoneinträge (Beisp.)

⟨*Beispiele/testlexikon.pl*⟩≡

```
pron([qu], [mask,sg,nom]) --> [wer].
pron([rel],[mask,sg,nom]) --> [der].
det([def],  [mask,sg,nom]) --> [der].
det([indef],[mask,sg,nom]) --> [ein].
det([qu],   [mask,sg,nom]) --> [welcher].
det([quant],[mask,sg,nom]) --> [jeder].

en([mask],[sg,nom]) --> ['Kepler'].
rn([mask],[sg,nom]) --> ['Mond'].
n([mask],[sg,nom]) --> ['Astronom'].
n([mask],[sg,nom]) --> ['Stern'].
n([Gen],[Num,Kas]) --> rn([Gen],[Num,Kas]). % Umwandlung
```

Die nichtlexikalische Regel `n --> rn` steht hier, damit alle `n/5` Klauseln aus einer Datei kommen.

Beispiele: Nominalphrasen 1

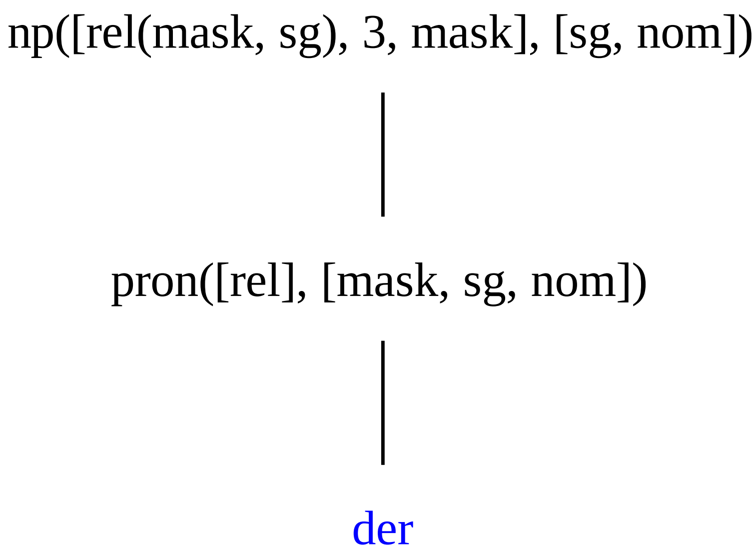
⟨*Nominalphrase 1.a*⟩≡

```
?- [grammatiktest].
?- parse.
|: der Astronom.

Aufruf: np([A,B,C],[D,E],F,[der,'Astronom'],[]).
Baum:
        np([def, 3, mask], [sg, nom])
           det([def], [mask, sg, nom]) der
           n([mask], [sg, nom]) 'Astronom'

Aufruf: s([A],[B,ind,C],D,[der,'Astronom'],[]).
```

Aufrufe für unpassende Startsymbole zeigen wir nicht mehr.

⟨*Nominalphrase 1.b*⟩≡

```
?- parsed. % mit Anzeige durch /usr/bin/dot
|: der.
```



⟨*Nominalphrase 1.c*⟩≡

```
?- parsed.
|: Kepler.
```

np([def, 3, mask], [sg, nom])

en([mask], [sg, nom])

Kepler

(Einfache) Sätze

Es werden nur einfache Sätze, die aus einem Verb und dessen Objekten bestehen, behandelt.

Kategorie der (einfachen) Sätze

$\langle$*Grammatik/kategorien.pl*$\rangle{+}{\equiv}$

```
kategorie(s([Def],[Temp,Mod,Vst])) :-
     definitheit(s,Def),
     tempus(Temp), % ggf. ; Temp = perf
     modus(Mod),
     verbstellung(Vst).
```

Bei den Satzarten unterscheiden wir: definite Sätze (Aussagen, `def`), Interrogativsätze (Fragen, `qu`), und –nach Genus und Numerus des Bezugsnomens unterschiedene– Relativsätze (`rel(Gen,Num)`):

⟨*Grammatik/kategorien.pl*⟩+≡

```
definitheit(s,Def) :-
     (Def = def ; Def = qu
     ; Def = rel(Gen,Num), genus(Gen),numerus(Num)).
```

Bei den Satzformen unterscheiden wir nach dem Satztempus `Temp`, dem Modus `Mod` und der Verbstellung `Vst`, d.h. ob das Verb an erster, zweiter, oder letzter Stelle im Satz steht:

⟨*Grammatik/kategorien.pl*⟩+≡

```
verbstellung(Vst) :-
     (Vst = ve ; Vst = vz ; Vst = vl).
```

Prädikativsätze 1

Betrachten wir zuerst Prädikativsätze mit einem Prädikatsnomen:

⟨*Beispiele für Prädikativsätze*⟩≡

```
Kepler ist ein Astronom.
Ist Triton ein Planet?
(ein Mond,) dessen Durchmesser 30000 km ist.
```

⟨*Grammatik/saetze.pl*⟩≡

```
s([Def],[Temp,Mod,vz]) -->
    np([Def1,3,_Gen1],[Num,nom]),
    { (member(Def1,[def,indef,quant]), Def = def)
    ; (Def1 = qu, Def = qu) },
    v([sein],[3,Num,Temp,Mod]),
    np([Def2,3,_Gen2],[Num,nom]),
    { Def2 = indef ; Def2 = def }.
```

⟨*Grammatik/saetze.pl*⟩+≡

```
s([qu],[Temp,Mod,ve]) -->
     v([sein],[3,Num,Temp,Mod]),
     np([Def1,3,_Gen1],[Num,nom]),
     { Def1 = def ; Def1 = indef ; Def1 = quant },
     np([Def2,3,_Gen2],[Num,nom]), % ,['?'],
     { Def2 = indef ; Def2 = def }.

s([rel(Gen,Num)],[Temp,Mod,vl]) -->
     np([rel(Gen,Num),3,_Gen1],[Num2,nom]),
     np([Def2,3,_Gen2],[Num2,nom]),
     { member(Def2,[def,indef]) }, % quant?
     v([sein],[3,Num2,Temp,Mod]).
```

Testlexikon

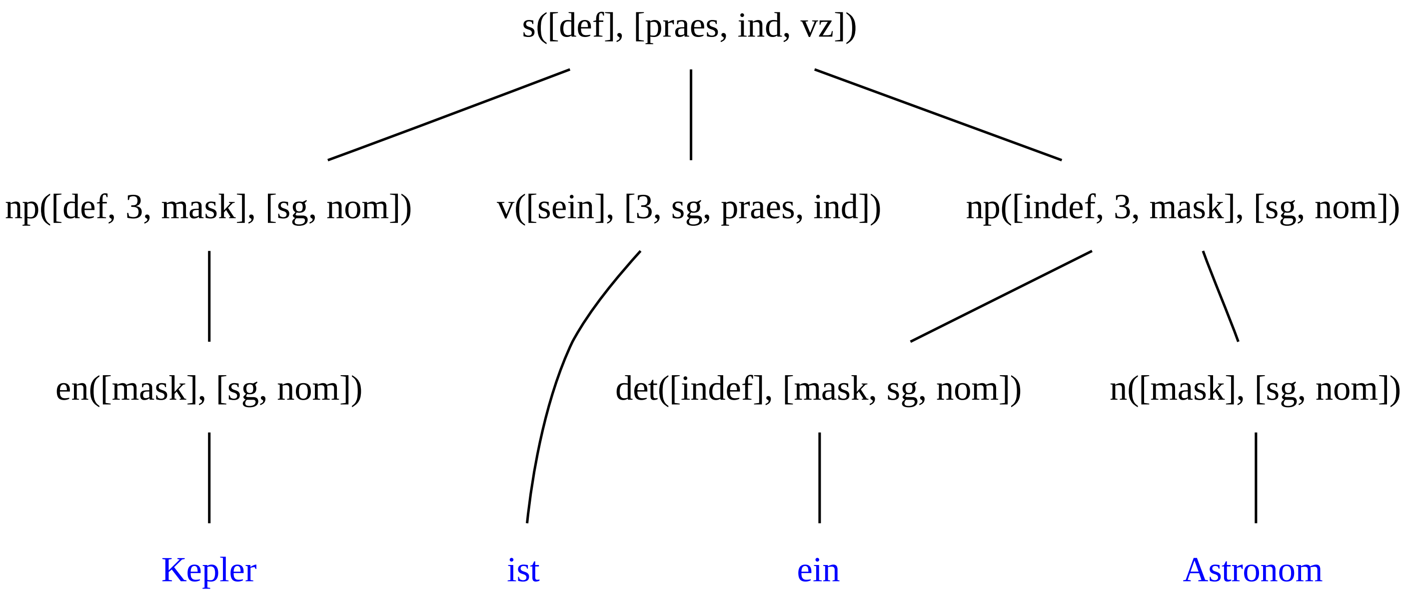
⟨*Beispiele/testlexikon.pl*⟩+≡

```
v([sein],[3,sg,praes,ind]) --> [ist].
v([sein],[3,sg,praet,ind]) --> [war].
v([sein],[3,pl,praes,ind]) --> [sind].
v([sein],[3,pl,praet,ind]) --> [waren].

en([mask],[sg,nom]) --> ['Uranus'].
en([mask],[sg,gen]) --> ['Uranus\''].
en([fem], [sg,nom]) --> ['Venus'].
en([mask],[sg,nom]) --> ['Triton'].
en([mask],[sg,gen]) --> ['Keplers'].
```

Beispiele: Prädikativsätze 1

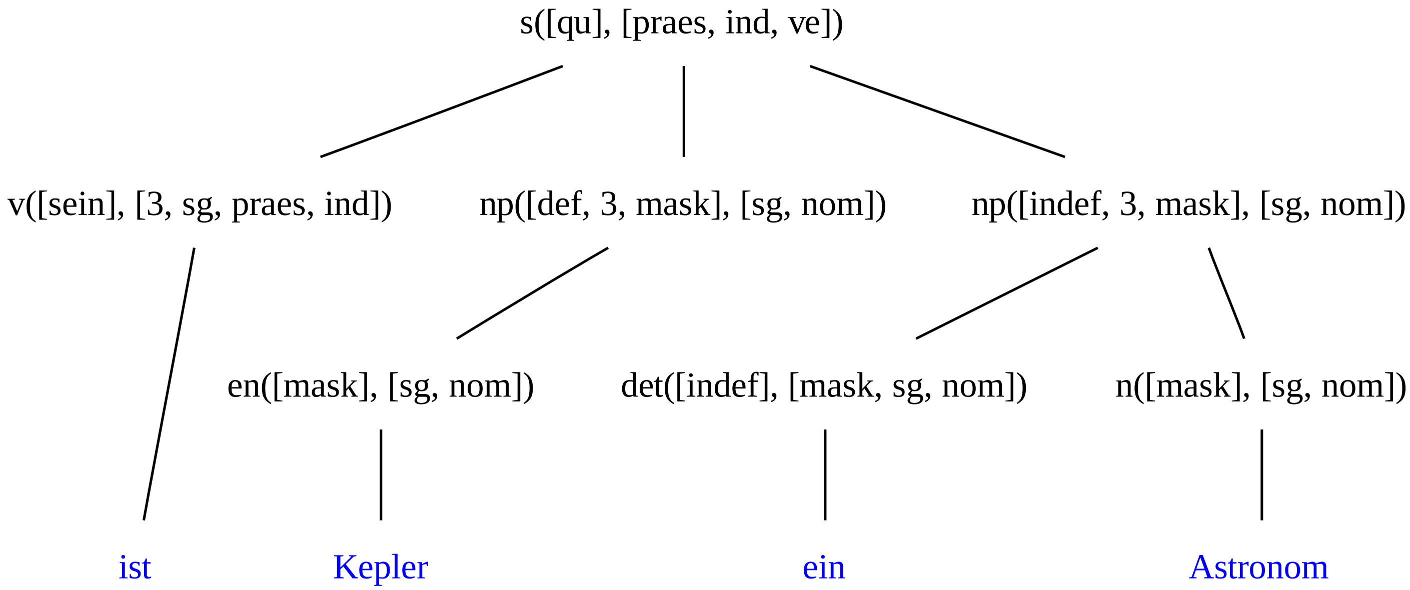
⟨*Prädikativsatz 1.a*⟩≡

```
?- parsed. Kepler ist ein Astronom.
```



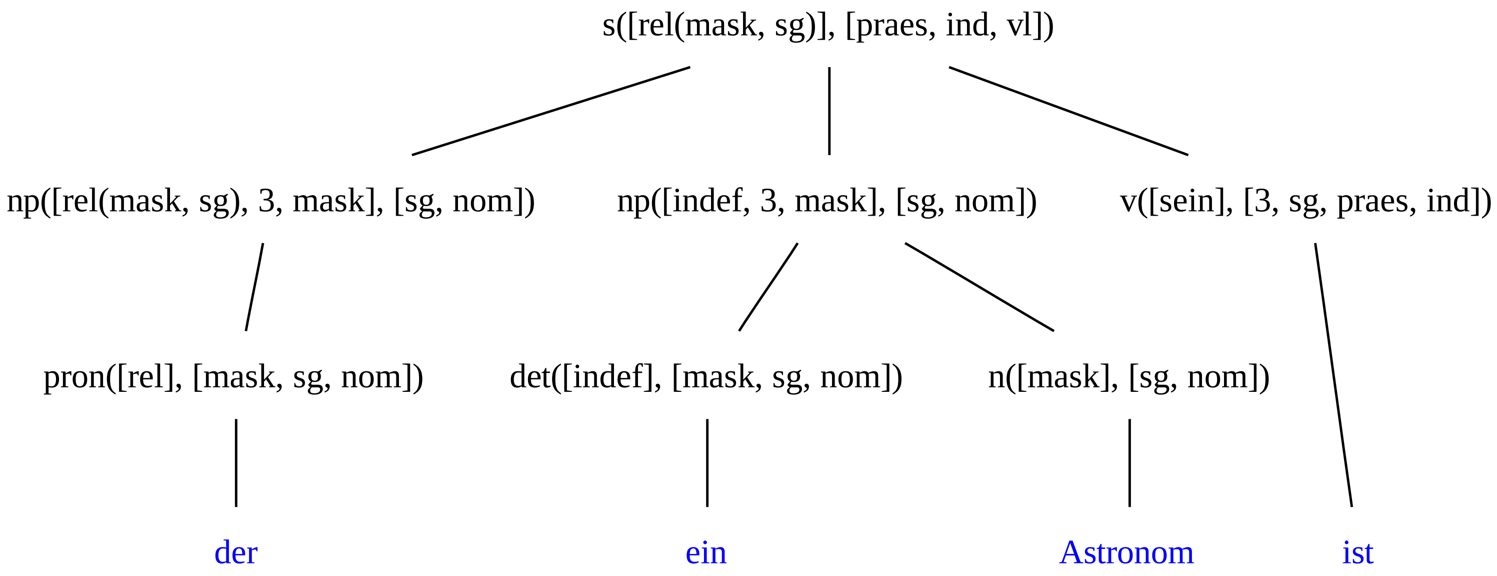
⟨*Prädikativsatz 1.b*⟩≡

```
?- parsed. ist Kepler ein Astronom.
```



⟨*Prädikativsatz 1.c*⟩≡

```
?- parsed. der ein Astronom ist.
```



Nominalphrasen 2

Wir brauchen noch Regeln für Nominalphrasen wie *der Astronom Kepler* und für Nominalphrasen mit einem Relativsatz (Kommata sehen die Regeln nicht vor, da der Tokenizer sie löscht):

⟨*Grammatik/np.pl*⟩+≡

```
np([def,3,Gen],[Num,Kas]) -->
    det([def],[Gen,Num,Kas]),
    ( n([Gen1],[Num,Kas]), {Gen1 = Gen}
    ; [], {Gen2 = Gen} ),
    en([Gen2],[Num,nom]),
    (s([rel(Gen,Num)],[_Temp,_Mod,vl]) ; []).

np([def,3,Gen],[Num,Kas]) -->
    en([Gen],[Num,Kas]),
    s([rel(Gen,Num)],[_Temp,_Mod,vl]).
```

⟨*Grammatik/np.pl*⟩+≡

```
np([Def,3,Gen],[Num,Kas]) -->
     det([Def],[Gen,Num,Kas]),
     n([Gen],[Num,Kas]),
     s([rel(Gen,Num)],[_Temp,_Mod,vl]).
```

Im Plural sind auch Nominalphrasen ohne Artikel üblich:

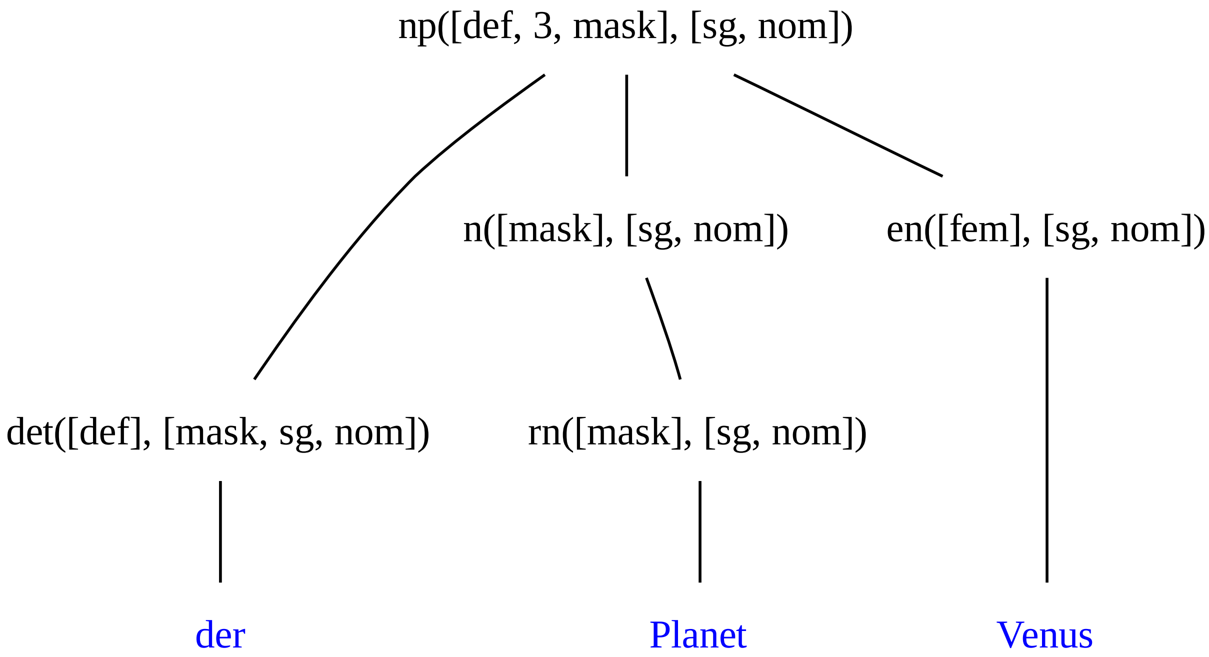
⟨*Grammatik/np.pl*⟩+≡

```
np([indef,3,Gen],[pl,Kas]) -->
     n([Gen],[pl,Kas]), { member(Kas,[nom,dat,akk]) }.
np([indef,3,Gen],[pl,Kas]) -->
     rn([Gen],[pl,Kas]), { member(Kas,[nom,dat,akk]) },
     np([Def2,3,_Gen2],[_Num2,gen]),
     { member(Def2,[def,indef]) }.
```

Beispiele: Nominalphrasen 2

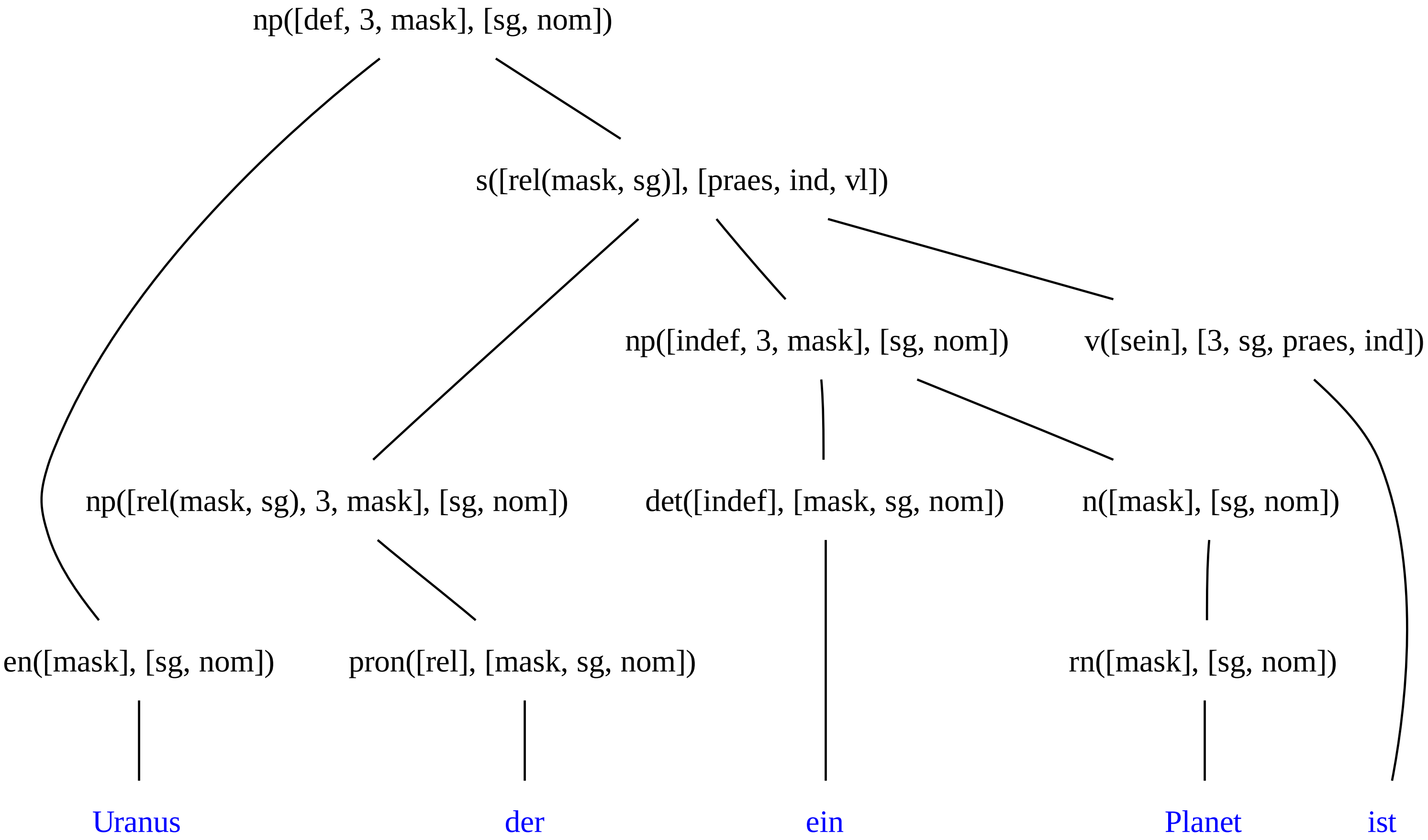
⟨*Nominalphrase 2.a*⟩≡

```
?- parsed. der Planet Venus.  % mask fem !
```



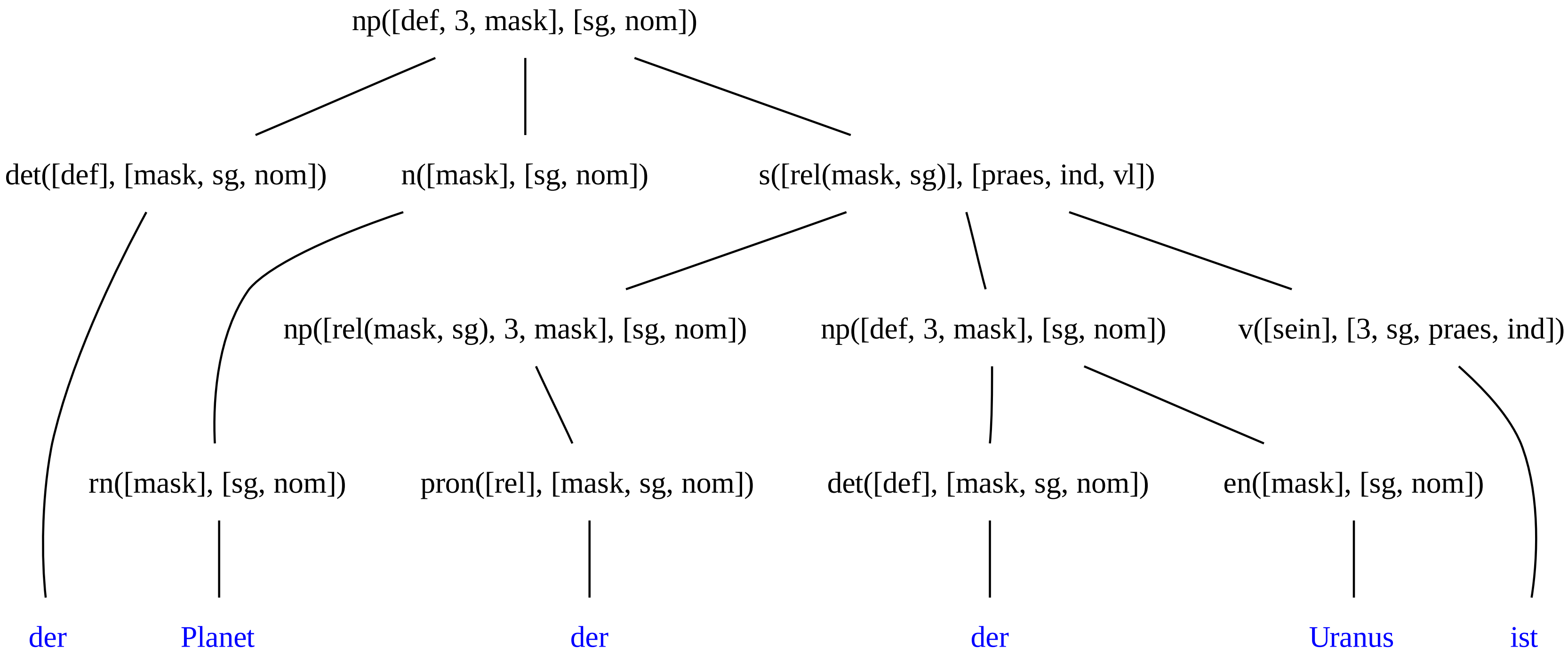
⟨*Nominalphrase 2.b*⟩≡

```
?- parsed. Uranus, der ein Planet ist.
```



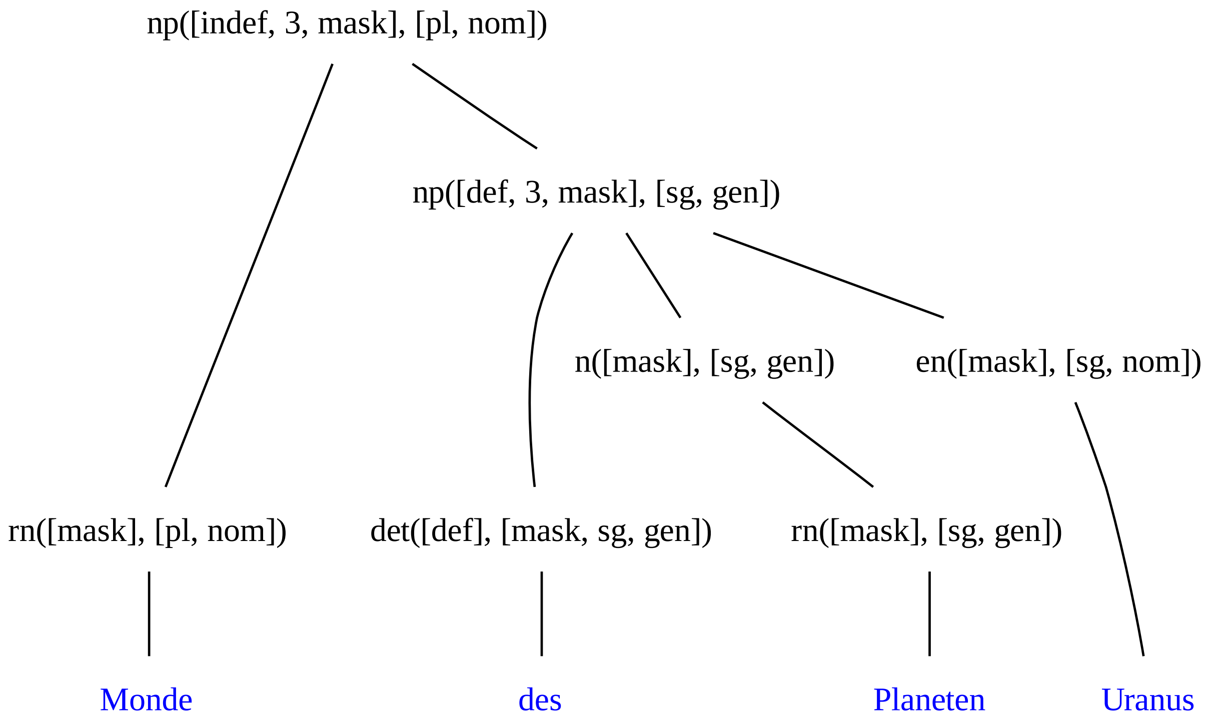
⟨*Nominalphrase 2.c*⟩≡

```
?- parsed. der Planet, der der Uranus ist.
```



⟨*Nominalphrase 2.e*⟩≡

```
?- parsed. Monde des Planeten Uranus.
```



Nominalphrasen 3

Schließlich sollen noch Nominalphrasen wie *der Durchmesser des Uranus* und *dessen Durchmesser* erkannt werden.

⟨*Grammatik/np.pl*⟩+≡

```
np([Def,3,Gen],[Num,Kas]) -->
     det([Def],[Gen,Num,Kas]),
     rn([Gen],[Num,Kas]),
     np([Def2,3,_Gen2],[_Num2,gen]),
       { member(Def2,[def,indef]) },
     { member(Def,[indef,def,qu,quant]) }.

np([rel(Gen2,Num2),3,Gen],[Num,Kas]) -->
     poss([rel],[Gen2,Num2,gen]),
     rn([Gen],[Num,Kas]).
```

Das wird nur für Relationsnomen (Nomen mit Genitivobjekt) erlaubt.

Testlexikon (Forts.)

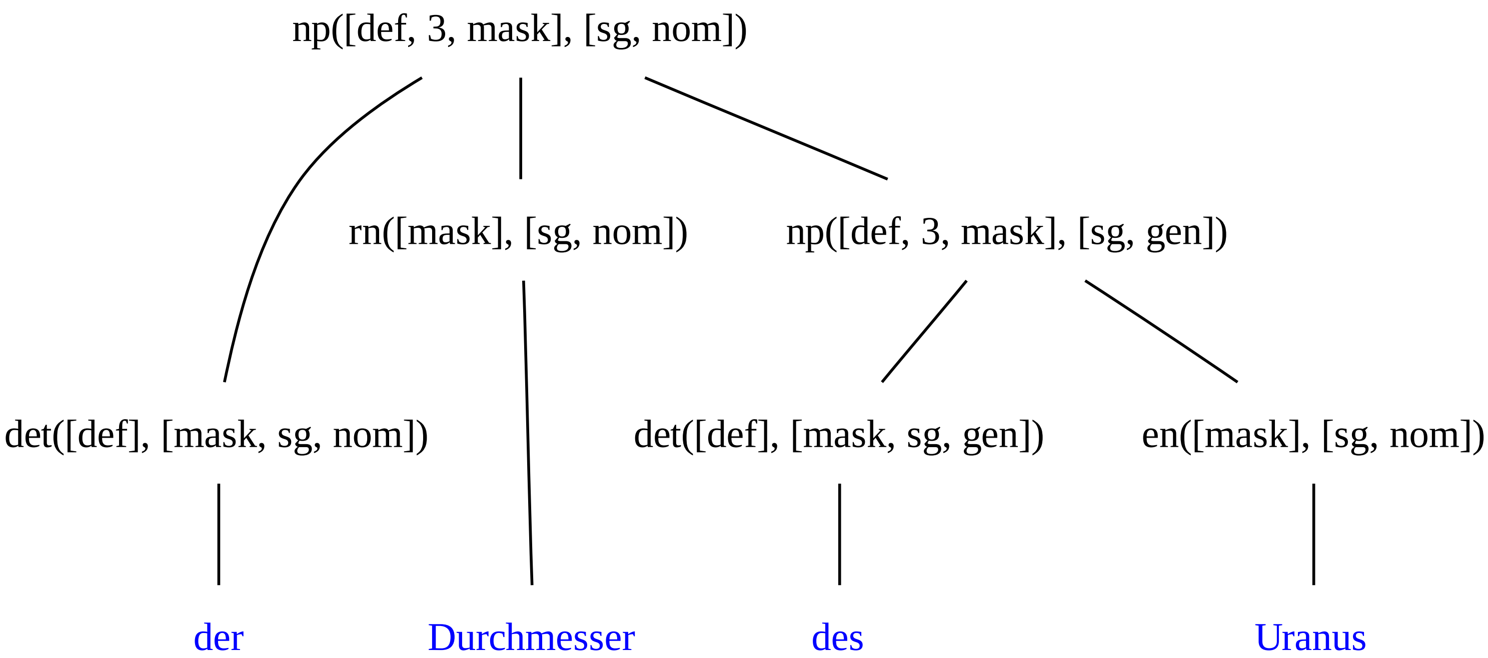
⟨*Beispiele/testlexikon.pl*⟩+≡

```
rn([mask],[sg,nom]) --> ['Durchmesser'].
rn([mask],[sg,nom]) --> ['Planet'].
rn([mask],[pl,nom]) --> ['Monde'].
rn([mask],[sg,gen]) --> ['Planeten'].
rn([mask],[sg,gen]) --> ['Mondes'].

poss([rel],[mask,sg,gen]) --> [dessen].
poss([rel],[fem,sg,gen]) --> [deren].
det([def],[mask,sg,gen]) --> [des].
det([def],[fem,sg,gen]) --> [der].
```

Beispiele: Nominalphrasen 3

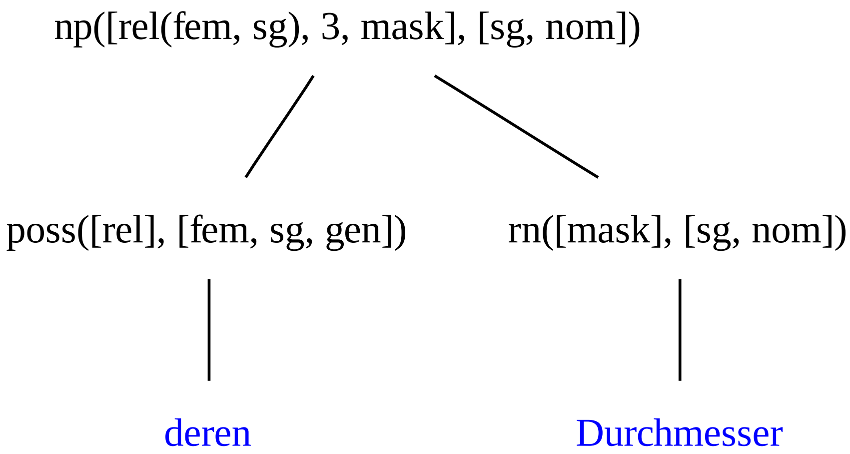
⟨*Nominalphrase 3.a*⟩≡

```
parsed. der Durchmesser des Uranus.
```



⟨*Nominalphrase 3.b*⟩≡

```
?- parsed. deren Durchmesser.
```



Bem.: Muß man *Funktions*- von *Relations*-Nomen trennen?

`kein Durchmesser eines Mondes` = `der(!) D. keines Mondes` vs.
`kein Planet eines Sterns`

Prädikativsätze 2

Es sollen noch Vergleichskonstruktionen wie *der Durchmesser des Uranus ist größer/kleiner als 50000 km* erlaubt werden. Hilfsverb und Vergleichsadjektiv bilden hier das Prädikat.

⟨*Grammatik/saetze.pl*⟩+≡

```
s([Def1],[Temp,Mod,vz]) -->
     np([Def1,3,_Gen1],[Num,nom]),
     { member(Def1,[def,indef,quant,qu]) },
     v([sein],[3,Num,Temp,Mod]),
     ap([def],[komp]).

s([qu],[Temp,Mod,ve]) -->
     v([sein],[3,Num,Temp,Mod]),
     np([Def1,3,_Gen1],[Num,nom]),
     { member(Def1,[def,indef,quant,qu]) },
     ap([def],[komp]).
```

⟨*Grammatik/saetze.pl*⟩+≡

```
s([Def1],[Temp,Mod,vl]) -->
     np([Def1,3,_Gen1],[Num,nom]),
     { member(Def1,[def,indef,quant,qu,rel(_,_)]) },
     ap([def],[komp]),
     v([sein],[3,Num,Temp,Mod]).
```

Adjektivphrasen (kompariert)

⟨*Grammatik/saetze.pl*⟩+≡

```
ap([def],[komp]) -->
     a([als(nom)],[komp]), [als],
     np([Def2,3,_Gen2],[_Num,nom]),
     { Def2 = indef ; Def2 = def ; Def2 = quant}.
```

Interrogative APs wie *wieviel größer als NP* betrachten wir nicht.

Prolog: Zahlatom $\mapsto$ Zahl

Zur Analyse von Größenangaben wie `20500 km` sind Token wie `'20500'` in die jeweiligen Zahlen wie `20500` umzuwandeln.

Atom $\rightleftharpoons$ Code-Nummern

$\langle$*Umwandlung eines Atoms in eine Zeichenreihe*$\rangle\equiv$

```
?- atom_codes('2005',Codes).
Codes = [50, 48, 48, 53]
```

$\langle$*Umwandlung einer Zeichenreihe in ein Atom*$\rangle\equiv$

```
?- atom_codes(Atom,[50,48,48,53]).
Atom = '2005'
```

$\langle$*Vergleich von Atom und Zeichenreihe*$\rangle\equiv$

```
?- atom_codes('Abcd',"Abcd").
Yes
```

Zahl $\rightleftharpoons$ Code-Nummern

⟨*Umwandlung einer Zahl in eine Zeichenreihe*⟩≡

```
?- number_codes(2005,Codes).
Codes = [50, 48, 48, 53]
```

⟨*Umwandlung einer Zeichenreihe in eine Zahl*⟩≡

```
?- number_codes(Zahl,[50,48,48,53]).
Zahl = 2005
```

⟨*Vergleich von Zahl und Zeichenreihe*⟩≡

```
?- number_codes(2005,"2005").
Yes
```

⟨*Nur Zahlzeichenreihen sind in Zahlen umwandelbar*⟩≡

```
number_codes(N,"A340").
ERROR: number_chars/2: Syntax error: Illegal number
```

Maßangaben (als NPs)

Es fehlen noch Nominalphrasen für die Größenangaben wie `50000 km`.
Dazu klassifizieren wir Zahlen n als Artikel im Plural (bei $n \geq 2$):

⟨*Beispiele/testlexikon.pl*⟩+≡

```
det([def],[_Gen,pl,_Kas]) --> [Zahlatom],
        { anzahl(Zahlatom,Zahl), Zahl >= 2 }.

n([mask],[pl,nom]) --> [km].

anzahl(Atom,N) :-
        atom_codes(Atom,Codes),
        number_codes(1234567890,NumCodes),
        subset(Codes,NumCodes), % Atom ist Zahlatom
        number_codes(N,Codes).
```

Ein Atom ist ein Zahlatom, wenn seine Code-Nummern nur Code-Nummern der Ziffern enthalten; sonst kann man es nicht in eine Zahl umwandeln.

Eigentlich sind Zahlen Adjektive, deren Formen bei allen Genus gleich sind. Komparierte Adjektive erfordern (als Prädikative) ein `nom`-'Objekt':

⟨*Beispiele/testlexikon.pl*⟩+≡

```
a([als(nom)],[komp]) --> [größer].
a([als(nom)],[komp]) --> [kleiner].

det([def],[fem,sg,nom]) --> [die].
pron([rel],[fem,sg,nom]) --> [die].
```

⟨*Grammatik/startsymbole.pl*⟩+≡

```
startsymbol(ap([_Def],[_Komp|_]))).
```

Beispiele: Prädikativsätze 2

⟨*Prädikativsatz 2.a*⟩≡

```
?- parse. der Durchmesser ist kleiner als 2000 km.
Baum:
        s([def], [praes, ind, vz])
           np([def, 3, mask], [sg, nom])
              det([def], [mask, sg, nom]) der
              n([mask], [sg, nom])
                 rn([mask], [sg, nom]) 'Durchmesser'
           v([sein], [3, sg, praes, ind]) ist
           ap([def], [komp])
              a([als(nom)], [komp]) kleiner
              als
              np([def, 3, mask], [pl, nom])
                 det([def], [mask, pl, nom]) '2000'
                 n([mask], [pl, nom]) km
```

⟨*Praedikativsatz 2.b*⟩≡

```
?- parse. ist Triton größer als Uranus.
Baum:
        s([qu], [praes, ind, ve])
           v([sein], [3, sg, praes, ind]) ist
           np([def, 3, mask], [sg, nom])
              en([mask], [sg, nom]) 'Triton'
           ap([def], [komp])
              a([als(nom)], [komp]) größer
              als
              np([def, 3, mask], [sg, nom])
                 en([mask], [sg, nom]) 'Uranus'
```

⟨*Praedikativsatz 2.c*⟩≡

```
?- parse. deren Durchmesser kleiner als 13000 km ist.
Baum:
        s([rel(fem, sg)], [praes, ind, vl])
           np([rel(fem, sg), 3, mask], [sg, nom])
              poss([rel], [fem, sg, gen]) deren
              rn([mask], [sg, nom]) 'Durchmesser'
           ap([def], [komp])
              a([als(nom)], [komp]) kleiner
              als
              np([def, 3, mask], [pl, nom])
                 det([def], [mask, pl, nom]) '13000'
                 n([mask], [pl, nom]) km
           v([sein], [3, sg, praes, ind]) ist
```

Sätze mit Vollverben

Für die Anwendung genügen transitive Vollverben: wir kennzeichnen sie durch die Artmerkmale `[nom,akk]`, d.h. daß sie ein Subjekt im Nominativ und ein Objekt im Akkusativ erwarten.

Verbzweitsätze sind Aussagen, es sei denn, sie enthalten eine interrogative Konstituente, dann sind es Fragen. Wir lassen nur am Satzanfang eine interrogative Konstituente zu (u.a. sind also keine Doppelfragen erlaubt):

⟨*Grammatik/saetze.pl*⟩+≡

```
s([Def],[Temp,Mod,vz]) -->
      np([Def1,3,_Gen1],[Num1,Kas1]),
      { member(Def1,[def,indef,quant,qu]) },
      v([nom,akk],[3,Num,Temp,Mod]),
      np([Def2,3,_Gen2],[Num2,Kas2]),
      { member(Def2,[def,indef,quant]) }, % kein qu!
      { ( [Num,nom,akk] = [Num1,Kas1,Kas2]
        ; [Num,nom,akk] = [Num2,Kas2,Kas1]
        ), (qu = Def1 -> Def = qu ; Def = def)
      }.
```

Verberstsätze werden immer als Fragen verstanden, da uneingeleitete Nebensätze in der Anwendung nicht vorkommen:

⟨*Grammatik/saetze.pl*⟩+≡

```
s([qu],[Temp,Mod,ve]) -->
     v([nom,akk],[3,Num,Temp,Mod]),
     { member(Def1,[def,indef,quant]) },
     np([Def1,3,_Gen1],[Num1,Kas1]),
     { member(Def2,[def,indef,quant]) },
     np([Def2,3,_Gen2],[Num2,Kas2]),
     { ( [Num,nom,akk] = [Num1,Kas1,Kas2]
       ; [Num,nom,akk] = [Num2,Kas2,Kas1] )
     }.
```

Verbletztsätze werden immer als Relativsätze verstanden, da die Anwendung keine anderen untergeordneten Sätze erfordert:

⟨*Grammatik/saetze.pl*⟩+≡

```
s([rel(GenR,NumR)],[Temp,Mod,vl]) -->
     np([rel(GenR,NumR),3,_Gen1],[Num1,Kas1]),
     np([Def2,3,_Gen2],[Num2,Kas2]),
     { member(Def2,[def,indef,quant]) },
     v([nom,akk],[3,Num,Temp,Mod]),
     { ( [Num,nom,akk] = [Num1,Kas1,Kas2]
       ; [Num,nom,akk] = [Num2,Kas2,Kas1] )
     }.
```

$\langle$*Beispiele/testlexikon.pl*$\rangle+\equiv$

```
v([nom,akk],[3,sg,praes,ind]) --> [entdeckt].
v([nom,akk],[3,sg,praes,ind]) --> [umkreist].

pron([qu], [mask,sg,akk]) --> [wen].
pron([rel],[mask,sg,akk]) --> [den].

det([def],  [mask,sg,akk]) --> [den].
det([indef],[mask,sg,akk]) --> [einen].
det([qu],   [mask,sg,akk]) --> [welchen].
det([quant],[mask,sg,akk]) --> [jeden].

n([mask],[sg,akk]) --> ['Mond'].
n([mask],[sg,akk]) --> ['Planeten'].
```

Beispiele: Sätze mit Vollverben

⟨*Satz mit Vollverb a (definit,verbzweit)*⟩≡

```
?- parse. jeder Astronom entdeckt einen Planeten.
Baum:
    s([def], [praes, ind, vz])
       np([quant, 3, mask], [sg, nom])
          det([quant], [mask, sg, nom]) jeder
          n([mask], [sg, nom]) 'Astronom'
       v([nom, akk], [3, sg, praes, ind]) entdeckt
       np([indef, 3, mask], [sg, akk])
          det([indef], [mask, sg, akk]) einen
          n([mask], [sg, akk]) 'Planeten'
```

⟨*Satz mit Vollverb a (interrogativ,verbzweit)*⟩≡

```
?- parse. welcher Stern umkreist den Uranus.
Baum:
    s([qu], [praes, ind, vz])
       np([qu, 3, mask], [sg, nom])
          det([qu], [mask, sg, nom]) welcher
          n([mask], [sg, nom]) 'Stern'
       v([nom, akk], [3, sg, praes, ind]) umkreist
       np([def, 3, mask], [sg, akk])
          det([def], [mask, sg, akk]) den
          en([mask], [sg, nom]) 'Uranus'
```

⟨*Satz mit Vollverb b (interrogativ,verberst)*⟩≡

```
?- parse. umkreist den Uranus ein Stern.
Baum:
     s([qu], [praes, ind, ve])
        v([nom, akk], [3, sg, praes, ind]) umkreist
        np([def, 3, mask], [sg, akk])
           det([def], [mask, sg, akk]) den
           en([mask], [sg, nom]) 'Uranus'
        np([indef, 3, mask], [sg, nom])
           det([indef], [mask, sg, nom]) ein
           n([mask], [sg, nom]) 'Stern'
```

⟨*Satz mit Vollverb c (relativ,verbletzt)*⟩≡

```
?- parse. den ein Stern umkreist.
Baum:
    s([rel(mask, sg)], [praes, ind, vl])
       np([rel(mask, sg), 3, mask], [sg, akk])
          pron([rel], [mask, sg, akk]) den
       np([indef, 3, mask], [sg, nom])
          det([indef], [mask, sg, nom]) ein
          n([mask], [sg, nom]) 'Stern'
       v([nom, akk], [3, sg, praes, ind]) umkreist
```

$\langle$*Satz mit Vollverb c (relativ,verbletzt)*$\rangle+\equiv$

```
?- parse. der den Planeten umkreist.
Baum:
    s([rel(mask, sg)], [praes, ind, vl])
       np([rel(mask, sg), 3, mask], [sg, nom])
          pron([rel], [mask, sg, nom]) der
       np([def, 3, mask], [sg, akk])
          det([def], [mask, sg, akk]) den
          n([mask], [sg, akk]) 'Planeten'
       v([nom, akk], [3, sg, praes, ind]) umkreist
```

Beispiel: Weitere Nominalphrasen

Nominalphrasen können jetzt auch die neuen Relativsätze enthalten:

⟨*Nominalphrase, deren Relativsatz ein Vollverb hat*⟩≡

```
?- parse. jeder Planet, den ein Mond umkreist.
Baum:
        np([quant, 3, mask], [sg, nom])
           det([quant], [mask, sg, nom]) jeder
           n([mask], [sg, nom])
              rn([mask], [sg, nom]) 'Planet'
           s([rel(mask, sg)], [praes, ind, vl])
              np([rel(mask, sg), 3, mask], [sg, akk])
                 pron([rel], [mask, sg, akk]) den
              np([indef, 3, mask], [sg, nom])
                 det([indef], [mask, sg, nom]) ein
                 n([mask], [sg, nom])
                    rn([mask], [sg, nom]) 'Mond'
              v([nom, akk], [3, sg, praes, ind]) umkreist
```

Lexikalische Regeln der DCG

Es ist Zeit, das Lexikon für die Beispielanwendung vollständig aufzubauen, damit man systematischer testen kann.

Artikel und Quantoren

Zuerst die Artikel und Quantoren im Singular:

⟨*Grammatik/lexikon_detpron.pl*⟩≡

```
det([indef],[mask,sg,nom]) --> [ein].
det([def],  [mask,sg,nom]) --> [der].
det([qu],   [mask,sg,nom]) --> [welcher].
det([quant],[mask,sg,nom]) --> [jeder].

det([indef],[mask,sg,gen]) --> [eines].
det([def],  [mask,sg,gen]) --> [des].
det([qu],   [mask,sg,gen]) --> [welches].
det([quant],[mask,sg,gen]) --> [jedes].
```

⟨*Grammatik/lexikon_detpron.pl*⟩+≡

```
det([indef],[mask,sg,dat]) --> [einem].
det([def],  [mask,sg,dat]) --> [dem].
det([qu],   [mask,sg,dat]) --> [welchem].
det([quant],[mask,sg,dat]) --> [jedem].

det([indef],[mask,sg,akk]) --> [einen].
det([def],  [mask,sg,akk]) --> [den].
det([qu],   [mask,sg,akk]) --> [welchen].
det([quant],[mask,sg,akk]) --> [jeden].

det([indef],[fem, sg,nom]) --> [eine].
det([def],  [fem, sg,nom]) --> [die].
det([qu],   [fem, sg,nom]) --> [welche].
det([quant],[fem, sg,nom]) --> [jede].
```

⟨*Grammatik/lexikon_detpron.pl*⟩+≡

```
det([indef],[fem, sg,gen]) --> [einer].
det([def],  [fem, sg,gen]) --> [der].
det([qu],   [fem, sg,gen]) --> [welcher].
det([quant],[fem, sg,gen]) --> [jeder].

det([indef],[fem, sg,dat]) --> [einer].
det([def],  [fem, sg,dat]) --> [der].
det([qu],   [fem, sg,dat]) --> [welcher].
det([quant],[fem, sg,dat]) --> [jeder].

det([indef],[fem, sg,akk]) --> [eine].
det([def],  [fem, sg,akk]) --> [die].
det([qu],   [fem, sg,akk]) --> [welche].
det([quant],[fem, sg,akk]) --> [jede].
```

Dann die Artikel und Quantoren im Plural:

⟨*Grammatik/lexikon_detpron.pl*⟩+≡

```
det([indef],[mask,pl,nom]) --> [einige].
det([def],  [mask,pl,nom]) --> [die].
det([qu],   [mask,pl,nom]) --> [welche].
det([quant],[mask,pl,nom]) --> [alle].

det([indef],[mask,pl,gen]) --> [einiger].
det([def],  [mask,pl,gen]) --> [der].
det([qu],   [mask,pl,gen]) --> [welcher].
det([quant],[mask,pl,gen]) --> [aller].

det([indef],[mask,pl,dat]) --> [einigen].
det([def],  [mask,pl,dat]) --> [den].
det([qu],   [mask,pl,dat]) --> [welchen].
det([quant],[mask,pl,dat]) --> [allen].
```

⟨*Grammatik/lexikon_detpron.pl*⟩+≡

```
det([indef],[mask,pl,akk]) --> [einige].
det([def],  [mask,pl,akk]) --> [die].
det([qu],   [mask,pl,akk]) --> [welche].
det([quant],[mask,pl,akk]) --> [alle].

det([indef],[fem, pl,nom]) --> [einige].
det([def],  [fem, pl,nom]) --> [die].
det([qu],   [fem, pl,nom]) --> [welche].
det([quant],[fem, pl,nom]) --> [alle].

det([indef],[fem, pl,gen]) --> [einiger].
det([def],  [fem, pl,gen]) --> [der].
det([qu],   [fem, pl,gen]) --> [welcher].
det([quant],[fem, pl,gen]) --> [aller].
```

⟨*Grammatik/lexikon_detpron.pl*⟩$+\equiv$

```
det([indef],[fem, pl,dat]) --> [einigen].
det([def],  [fem, pl,dat]) --> [den].
det([qu],   [fem, pl,dat]) --> [welchen].
det([quant],[fem, pl,dat]) --> [allen].

det([indef],[fem, pl,akk]) --> [einige].
det([def],  [fem, pl,akk]) --> [die].
det([qu],   [fem, pl,akk]) --> [welche].
det([quant],[fem, pl,akk]) --> [alle].
```

Artikel im Neutrum werden nicht angegeben, da die Anwendung keine Nomina im Neutrum hat.

Zahlquantoren

Die Anzahlen werden nicht wie Adjektive, sondern wie Quantoren behandelt.

⟨*Grammatik/lexikon_detpron.pl*⟩+≡

```
det([def],[_Gen,pl,_Kas]) --> [Zahlatom],
     { anzahl(Zahlatom,Zahl), Zahl >= 2 }.

anzahl(Atom,N) :-
     atom_codes(Atom,Codes),
     number_codes(1234567890,NumCodes),
     subset(Codes,NumCodes), % Atom ist Zahlatom
     number_codes(N,Codes).
```

(Bem. Sollte die Definitheit nicht `quant` sein?)

Possessiv-, Interrogativ- und Relativpronomen

⟨*Grammatik/lexikon_detpron.pl*⟩$+\equiv$

```
poss([rel],[mask,sg,gen]) --> [dessen].
poss([rel],[fem, sg,gen]) --> [deren].
poss([rel],[mask,pl,gen]) --> [deren].
poss([rel],[fem, pl,gen]) --> [deren].

pron([qu], [mask,sg,nom]) --> [wer].
pron([qu], [mask,sg,akk]) --> [wen].

pron([rel],[mask,sg,nom]) --> [der].
pron([rel],[mask,sg,gen]) --> [dessen].
pron([rel],[mask,sg,dat]) --> [dem].
pron([rel],[mask,sg,akk]) --> [den].
```

Bem. Es werden nur relativierende Possessivpronomen behandelt, und nur relativierende und interrogative Personalpronomina.

⟨*Grammatik/lexikon_detpron.pl*⟩+≡

```
pron([rel],[fem, sg,nom]) --> [die].
pron([rel],[fem, sg,gen]) --> [derer].
pron([rel],[fem, sg,dat]) --> [der].
pron([rel],[fem, sg,akk]) --> [die].

pron([rel],[mask,pl,nom]) --> [die].
pron([rel],[mask,pl,gen]) --> [deren].
pron([rel],[mask,pl,dat]) --> [denen].
pron([rel],[mask,pl,akk]) --> [die].

pron([rel],[fem, pl,nom]) --> [die].
pron([rel],[fem, pl,gen]) --> [derer].
pron([rel],[fem, pl,dat]) --> [denen].
pron([rel],[fem, pl,akk]) --> [die].
```

Adjektive, Verben, Nomen, Eigennamen

Zuerst ein paar Sonderfälle:

⟨*Grammatik/lexikon_nomenverb.pl*⟩≡

```
a([als(nom)],[komp]) --> [größer].
a([als(nom)],[komp]) --> [kleiner].

n([mask],[pl,nom]) --> [km].

n([Gen],[Num,Kas]) --> rn([Gen],[Num,Kas]).

v([sein],[3,sg,praes,ind]) --> [ist].
v([sein],[3,sg,praet,ind]) --> [war].
v([sein],[3,pl,praes,ind]) --> [sind].
v([sein],[3,pl,praet,ind]) --> [waren].
```

Die lexikalischen Regeln zu den Kategorien `v,n,rn,en` setzen voraus, daß ein Vollformenlexikon `wort/3` geladen ist.

⟨*Grammatik/lexikon_nomenverb.pl*⟩+≡

```
v(Art,Form) --> [Vollform],
    { wort(_Stamm,v(Art,Form),Vollform) }.
n(Art,Form) --> [Vollform],
    { wort(_Stamm,n(Art,Form),Vollform) }.
rn(Art,Form) --> [Vollform],
    { wort(_Stamm,rn(Art,Form),Vollform) }.
en(Art,Form) --> [Vollform],
    { wort(_Stamm,en(Art,Form),Vollform) }.
a(Art,Form) --> [Vollform],
    { wort(_Stamm,a(Art,Form),Vollform) }.
```

Ein solches Vollformenlexikon wird nun aus einem Stammlexikon erzeugt.

Nomen-, Eigennamen- und Verblexikon

Die Vollformen für Nomen, Eigennamen und Verben erzeugen wir mit Flexionsprogrammen[a] aus Stammformen.

In das Stammformenlexikon nehmen wir die Wortstämme von Verben und Nomen auf, die in der Anwendung vorkommen:

⟨*Anwendungen/Astronomie/stammformen.pl*⟩≡

```
:- module(stammformenlexikon,[lex/4]).

% lex(?Stammform,?Wortart,?Artmerkmale,?Flexionskl.)

lex(entdecken,       v,[nom,akk],rg(0)).
lex(umkreisen,       v,[nom,akk],rg(0)).
lex('Himmelskörper',n,[mask],(s1, p2 )).
lex('Stern',         n,[mask],(s1, p1 )).
lex('Sonne',         n,[fem], (s3, p3 )).
lex('Durchmesser', rn,[mask],(s1, p2 )).
```

[a] Nomenflexion: siehe Nachtrag unten

⟨*Anwendungen/Astronomie/stammformen.pl*⟩+≡

```
lex('Astronom',     n,[mask],(s2e,p3e)).

lex('Bond',        en,[mask],(s1,-)).
lex('Cassini',     en,[mask],(s1,-)).
lex('Galilei',     en,[mask],(s1,-)).
lex('Hall',        en,[mask],(s1,-)).
lex('Herschel',    en,[mask],(s1,-)).
lex('Huyghens',    en,[mask],(s1,-)).
lex('Kepler',      en,[mask],(s1,-)).
lex('Lassell',     en,[mask],(s1,-)).
lex('Melotte',     en,[mask],(s1,-)).
lex('Nicholson',   en,[mask],(s1,-)).
lex('Perrine',     en,[mask],(s1,-)).
lex('Tombaugh',    en,[mask],(s1,-)).
```

$\langle$*Anwendungen/Astronomie/stammformen.pl*$\rangle+\equiv$

```
lex('Planet',        rn,[mask],(s2e,p3e)).

lex('Erde',          en,[fem], (s1,-)).
lex('Jupiter',       en,[mask],(s1,-)).
lex('Mars',          en,[mask],(s1,-)).
lex('Merkur',        en,[mask],(s1,-)).
lex('Neptun',        en,[mask],(s1,-)).
lex('Pluto',         en,[mask],(s1,-)).
lex('Saturn',        en,[mask],(s1,-)).
lex('Uranus',        en,[mask],(s1,-)).
lex('Venus',         en,[fem], (s3,-)).
```

⟨*Anwendungen/Astronomie/stammformen.pl*⟩+≡

```
lex('Mond',         rn,[mask],(s1e,p1 )).

lex('Dione',        en,[fem], (s1,-)).
lex('Europa',       en,[fem], (s3,-)).
lex('Ganymed',      en,[mask],(s1,-)).
lex('Io',           en,[fem], (s3,-)).
lex('Kallisto',     en,[fem], (s3,-)).
lex('Mimas',        en,[mask],(s1,-)).
lex('Titan',        en,[mask],(s1,-)).
lex('Triton',       en,[mask],(s1,-)).
% Merkmale geraten:
lex('Adrastea',     en,[fem], (s3,-)).
lex('Amalthea',     en,[fem], (s3,-)).
```

$\langle$*Anwendungen/Astronomie/stammformen.pl*$\rangle+\equiv$

```
lex('Ananke',      en,[fem], (s3,-)).
lex('Ariel',       en,[mask],(s1,-)).
lex('Carme',       en,[fem], (s3,-)).
lex('Charon',      en,[mask],(s1,-)).
lex('Deimos',      en,[mask],(s1,-)).
lex('Diana',       en,[fem], (s3,-)).
lex('Elara',       en,[fem], (s3,-)).
lex('Enkeladus',   en,[mask],(s1,-)).
lex('Himalia',     en,[fem], (s3,-)).
lex('Hyperion',    en,[mask],(s1,-)).
lex('Iapetus',     en,[mask],(s1,-)).
```

Erzeugung der Nomen- und Verbvollformen

Aus `Anwendungen/Astronomie/stammformen.pl` muß man mit `erzeuge_vollformen/0` das Vollformenlexikon zu Nomen, Eigennamen und Verben bilden:

⟨*Erstellung der Nomen- und Verbvollformen*⟩≡

```
?- [morphologie].
?- erzeuge_vollformen.
Lade stammformen.pl aus
     Anwendungen/<Verzeichnis><Punkt><Return>
|: 'Astronomie'.
...
Vollformen erzeugt und nach
   'Anwendungen/Astronomie/vollformen.pl' geschrieben.
Yes
```

Nach Änderungen im Stammformenlexikon muß man das Vollformenlexikon neu erstellen.

Laden der Grammatik

Mit `?- [grammatik,astronomie].` lädt man Grammatik und Lexikon:

⟨*grammatik.pl*⟩≡

```
:- ['Parser/tokenizer.mini.pl'],
   ['Parser/term_expansion.pl',
    'Grammatik/startsymbole.pl',
    'Grammatik/np.pl',
    'Grammatik/saetze.pl',
    'Grammatik/lexikon_detpron.pl',
    'Grammatik/lexikon_nomenverb.pl',
    'Parser/showTree.pl',
    'Parser/dot.syntaxbaum.pl'].
```

⟨*astronomie.pl*⟩≡

```
:- ['Anwendungen/Astronomie/vollformen.pl'].
```

Für andere Anwendungen muß man andere Vollformenlexika erstellen und laden; ggf. auch `Grammatik/lexikon_detpron.pl` erweitern.

Testsätze von einer Datei lesen

Zum Testen der Grammatik ist es nützlich, wenn man Beispiele aus einer Datei lesen (und ggf. die Analysen auf eine Ausgabedatei schreiben) kann. Dazu dienen `parse/1` und `parsen/1`:

⟨*Parser/tokenizer.mini.pl*⟩+≡

```
parse(Dateiname) :-
    % atom_concat(Dateiname,'.aus',Ausgabe),
    % tell(Ausgabe), % ggf. Ausgabedatei öffnen
    open(Dateiname,read,Strom),
    parsen(Strom),
    close(Strom),
    % told,          % ggf. Ausgabedatei schließen
    nl,write('Fertig.').
```

Man muß `parse/1` mit “langem Namen” aufrufen:

⟨*Parsen aller Sätze einer Datei*⟩≡

```
?- tokenizer:parse('Grammatik/testsaetze.txt').
```

⟨*Parser/tokenizer.mini.pl*⟩+≡

```
parsen(Strom) :-
    read_sentence(Strom, Satz, []),
    (Satz = [] -> true  % Dateiende
    ; tokenize(Satz,Atome),
        nl,writeq(Atome),nl,
        (setof(Baum,parse(Atome,Baum),Trees)
        -> showTrees(Trees,3), nl
        ; nl, tab(3),
            write('* keine Analysen gefunden *'),
            nl),
        parsen(Strom)   % weitere Sätze
    ).
```

Bem. Wenn `Goal` keine Lösungen hat, gibt `setof(+Term,+Goal,-Liste)` nicht die leere Liste aus, sondern scheitert!

Testsätze

Die "Sätze" einer Datei sind mit `<Punkt><Zeilenumbruch>` von einander zu trennen, damit `read_sentence/3` sie erkennt.

Folgende Beispiele werden erkannt und sollten nach einer Grammatikänderung wieder erkannt werden:

⟨*Grammatik/testsaetze.txt*⟩≡

```
der Uranus ist ein Planet.
größer als der Durchmesser der Venus.
der Durchmesser des Planeten.
der Planet Venus, der ein Planet ist.
deren Planet die Venus ist.
dessen Planet die Venus ist.
der Venus, die ein Planet ist.
der Durchmesser der Venus ist kleiner
 als der Durchmesser des Uranus.
ist der Planet Venus ein Planet.
ist die Venus ein Planet.
```

⟨*Grammatik/testsaetze.txt*⟩+≡

```
Kepler entdeckte einen Mond.
welcher Astronom entdeckte einen Mond.
welchen Mond entdeckte der Astronom.

welcher Mond des Uranus.
welcher Mond des Uranus umkreist die Sonne.

ein Stern, den ein Astronom entdeckte.
ein Stern, den einige Astronomen entdeckten.
ein Astronom, der 230 Sterne entdeckte.
ein Himmelskörper, dessen Monde Planeten sind.

sind die Planeten die Monde der Venus.
sind die Planeten Monde der Sonne.
ist der Planet Venus die Venus.
ist die Venus ein Mond des Uranus.
```

⟨*Grammatik/testsaetze.txt*⟩+≡

```
entdeckte Kepler den Mond eines Planeten.
entdeckte Kepler 3 Monde des Uranus.
entdeckte Kepler alle Planeten der Sonne.
entdeckten alle Astronomen einen Planeten.
der den Mond eines Planeten entdeckte.
der die Monde einiger Planeten entdeckte.

ist der Durchmesser der Venus kleiner als der
 Durchmesser der Sonne.
welcher Mond eines Planeten ist kleiner als der
 Durchmesser des Uranus.
ein Himmelskörper, dessen Durchmesser kleiner
 als 50000 km ist.
```

Nicht erkannte Eingaben

Korrekterweise werden nicht erkannt:

- Elliptische Ausdrücke,
- Eigennamen mit nicht-definitem Artikel,
- Possessivum und Genitiv-Attribut bei absoluten Nomen,
- untergeordnete Fragesätze,
- verschobene Relativsätze

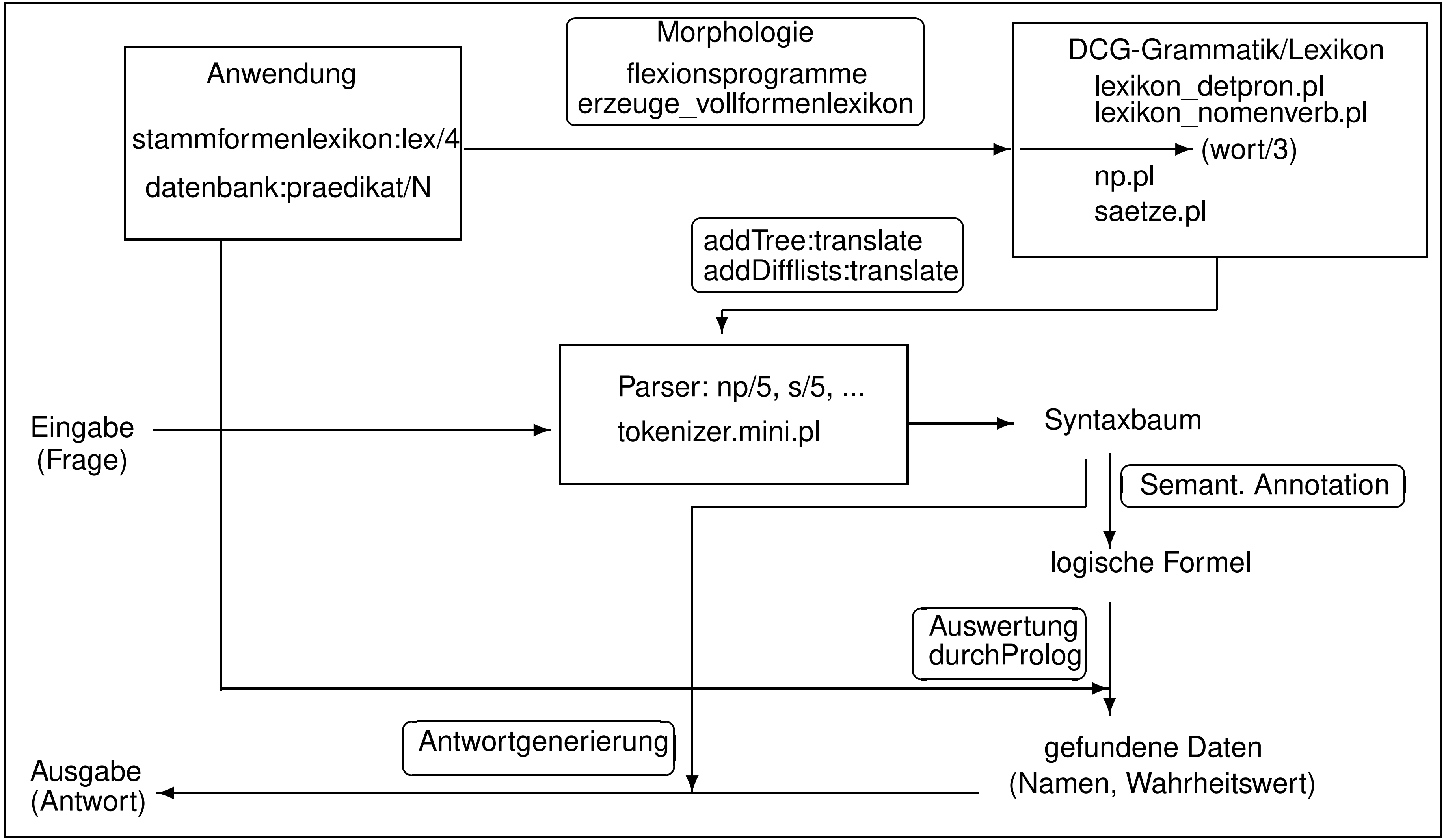
⟨*Grammatik/testsaetze.txt*⟩$+\equiv$

```
ist der Durchmesser der Venus kleiner als der der Sonne.
eine Venus.
jede Venus.
welche Venus.
dessen Astronom.
der Astronom der Venus.
welcher Mond der Uranus ist.
der einen Mond entdeckte, der den Uranus umkreist.
```

Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing)
- Semantik:
  1. Datenbank (Objekte und Grundrelationen)
  2. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  3. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

Überblick


Morphologie
flexionsprogramme
erzeuge_vollformenlexikon
Anwendung
stammformenlexikon:lex/4
datenbank:praedikat/N
DCG-Grammatik/Lexikon
lexikon_detpron.pl
lexikon_nomenverb.pl
(wort/3)
np.pl
saetze.pl
addTree:translate
addDifflists:translate
Parser: np/5, s/5, ...
tokenizer.mini.pl
Eingabe
(Frage)
Syntaxbaum
Semant. Annotation
logische Formel
Auswertung
durchProlog
Antwortgenerierung
Ausgabe
(Antwort)
gefundene Daten
(Namen, Wahrheitswert)

Semantik: Datenbank

Unser Ziel ist, eine Datenbankabfrage in natürlicher Sprache zu programmieren. Dazu müssen wir:

1. Fragen in natürlicher Sprache in Prolog-Ziele übersetzen,
2. mit Prolog in der Datenbank nach Informationen suchen,
3. die Treffer als Antwort in natürlicher Sprache ausgeben.

Datenbank

Die Datenbank enthält Informationen über Sterne und Astronomen. Sie exportiert Prädikate, mit denen man auf Datensätze einer relationalen Datenbank `db/5` zugreift:

⟨*Anwendungen/Astronomie/datenbank.pl*⟩≡

```
:- module(datenbank,[astronom/1,stern/1,sonne/1,
                     planet/2,mond/2,durchmesser/2,
                     objekt/1,mond/1,planet/1,
                     umkreisen/2,entdecken/2]).
```

Datenbank: Datensätze

⟨*Anwendungen/Astronomie/datenbank.pl*⟩+≡

```
% Datensaetze: db(S,A,D,E,Z) für
% db(Stern,Art,Durchmesser,Entdecker,Zentralstern)

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).
```

⟨*Anwendungen/Astronomie/datenbank.pl*⟩+≡

```
db( adrastea, mond,      24,         _, jupiter).
db( amalthea, mond,     135,         _, jupiter).
db(   ananke, mond,      30,nicholson, jupiter).
db(    ariel, mond,    1158,  lassell,  uranus).
db(    carme, mond,      40,  melotte, jupiter).
db(   charon, mond,    1000,         _,   pluto).
db(   deimos, mond,       8,     hall,    mars).
db(    diana, mond,    1100,  cassini,  saturn).
db(    dione, mond,    1120, herschel,  saturn).
db(    elara, mond,      76,nicholson, jupiter).
db(enkeladus, mond,     500, herschel,  saturn).
db(   europa, mond,    3138,  galilei, jupiter).
db(  ganymed, mond,    5262,  galilei, jupiter).
```

⟨*Anwendungen/Astronomie/datenbank.pl*⟩+≡

```
db(  himalia, mond,     186,  perrine, jupiter).
db( hyperion, mond,     205,     bond,  saturn).
db(  iapetus, mond,    1460,  cassini,  saturn).
db(       io, mond,    3630,  galilei, jupiter).
db( kallisto, mond,    4800,  galilei, jupiter).
db(    mimas, mond,     392, herschel,  saturn).
db(     mond, mond,    3476,        _,    erde).

db(    titan, mond,    5150, huyghens,  saturn).
db(   triton, mond,    2700,  lassell,  neptun).
```

Für einige Monde im Planetensystem fehlen analoge Einträge.

Datenbank: Zugriffsprädikate

Neben den Datensätzen soll die Datenbank auch die Prolog-Prädikate definieren, mit denen eine Suchanfrage in Prolog ausgedrückt werden kann:

⟨*Anwendungen/Astronomie/datenbank.pl*⟩+≡

```
% Zugriffspraedikate:

stern(S) :-    db(S,_,_,_,_).
sonne(S) :-    db(S,sonne,_,_,_).
astronom(X) :-
    setof(E,S^A^D^Z^(db(S,A,D,E,Z), atom(E)),Es),
    member(X,Es).  % vermeidet Mehrfachnennungen

mond(S,Z) :-   db(S,mond,  _,_,Z).
planet(S,Z) :- db(S,planet,_,_,Z).
umkreisen(S,Z) :- db(S,_,_,_,U), atom(U), Z = U.
entdecken(E,S) :- db(S,_,_,A,_), atom(A), E = A.
durchmesser(D,S) :- db(S,_T,D,_E,_Z).
```

⟨*Anwendungen/Astronomie/datenbank.pl*⟩+≡

```
mond(S) :- mond(S,_).
planet(S) :- planet(S,_).

objekt(X) :-  atomic(X), !,
     (stern(X) ; astronom(X) ; durchmesser(X,_)).
objekt(X:T) :- type(X/0,T).
```

Mit `objekt(X:T)` werden getypte Variable auf Eigennamen beschränkt.

Damit die Zugriffsprädikate nur dann eine Antwort liefern, wenn `db/5` in der jeweiligen Komponente keine Variable _ hat, wird diese mit `atom/1` geprüft.

Typisierung der Grundprädikate

Damit die Suche in der Datenbank auf kleinere Teilbereiche beschränkt werden kann, geben wir den Prädikaten einen semantischen Typ:

⟨*Anwendungen/Astronomie/datenbank.pl*⟩+≡

```
% type(+Praedikat/Stelligkeit,-Typ)
%    m=Mensch, s=Gestirn, n=Zahl, e=Entität, t=Wahrheitswert
type(N/0,m) :- astronom(N).
type(N/0,s) :- stern(N).
type(N/0,n) :- durchmesser(N,_).
type(objekt/1,(e -> t)).    type(eq/2,(T*T -> t)).
type(astronom/1,(m -> t)).
type(sonne/1,(s -> t)).     type(stern/1,(s -> t)).
type(planet/1,(s -> t)).    type(mond/1,(s -> t)).
type(planet/2,(s*s -> t)). type(mond/2,(s*s -> t)).
type(entdecken/2,(m*s -> t)). type('=<'/2,(n*n -> t)).
type(umkreisen/2,(s*s -> t)). type('<'/2,(n*n -> t)).
type(durchmesser/2,(n*s -> t)).
```

Typen und Untertypen werden erst bei der Auswertung benutzt.

$\langle$*Anwendungen/Astronomie/datenbank.pl*$\rangle{+}{\equiv}$

```
subtype(X,Y) :-
     (var(X) ; var(Y)), !, X = Y.
subtype(s,e).
subtype(m,e).
subtype(n,e).
subtype(X*Y,U*V) :-
     subtype(X,U), subtype(Y,V).
subtype((A -> B),(C -> D)) :-
    subtype(C,A), subtype(B,D).
subtype(X,X).
```

Klammerregeln:

```
 (a * b * c)   =   ((a * b) * c),
(a * b -> c)   =   ((a * b) -> c),
(a -> b * c)   =   (a -> (b * c))
(a -> b -> c)  =   (a -> (b -> c))
```

Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing)
- Semantik:
  1. Datenbank (Objekte und Grundrelationen)
  2. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  3. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

Prädikatenlogik (PL) als Datenbanksprache

Wir wollen als Anfragesprache logische Formeln benutzen, für die Antworten in der Form von Belegungen der Variablen der Anfrage gesucht werden.

⟨*Terme und Formeln*⟩≡

```
Term := Variable          % Prolog-Variable
     | Zahl | Atom        % Atom der Datenbank: mars,...
Formel :=
     datenbank:Grundpraedikat(Terme)
   | Zahl < Zahl                | Zahl =< Zahl
   | eq(Term,Term)              | neg(Formel)
   | (Formel & Formel)          | (Formel \/ Formel)
   | (Formel => Formel)         | (Formel <=> Formel)
   | ex(Variable,Formel)        | all(Variable,Formel)
   | anzahl(Variable,Formel,Zahl)
Frage := Formel | qu(Variable,Formel)
```

Voraussetzung: Alle Variablen `V` bei den Teilformeln `ex(V,F)`, `all(V,F)` und `anzahl(V,F,N)` einer Formel sind verschieden.

Ob ein Prolog-Term eine Formel in diesem Sinne ist, sollte mit einem Testprädikat geprüft werden, damit man nur syntaktisch korrekte Formeln auszuwerten versucht. (Übungsaufgabe!).

Wir erklären `&`, `\/`, `=>` als Infix-Operatoren für *und*, *oder*, und *wenn-dann*, wobei `&` enger binden soll als `\/` und `=>`:

⟨*Semantik/auswertung.pl*⟩≡

```
:- op(500,yfx,&), op(600,yfx,'\/'),
     op(600,xfx,'=>'), op(600,xfx,'<=>').
```

Trotz der Bindungsstärken: Klammern besser hinschreiben!

Anfragen können auf zwei Weisen gestellt werden:

⟨*Ja/Nein-Frage und W-Frage*⟩≡

```
?- Formel.             % Gilt Formel?
?- qu(Var,Formel).   % Welche Var erfüllen Formel?
```

Auswertung von Aussagen der PL

Zur Auswertung von Formeln benutzen wir Prolog. Wir werten zuerst nur *Aussagen*, d.h. Formeln *ohne freie Variablen*, aus.

1. Bei aussagenlogischen Verbindungen mit *und* `&`, *oder* `\/`, *wenn-dann* `=>` und *nicht* `neg` benutzen wir Prolog:

   ⟨*Semantik/auswertung.pl*⟩+≡

   ```
   wahr((F & G)) :- !, wahr(F) , wahr(G).
   wahr((F \/ G)) :- !, (wahr(F), ! ; wahr(G)).
   wahr((F => G)) :-  !, wahr(neg(F) \/ G).
   wahr((F <=> G)) :- !, wahr((F => G) & (G => F)).
   wahr(neg(F)) :- !, (wahr(F) -> fail ; true).
   ```

2. Bei Aussagen `all(Var,Formel)` wird eine gleichbedeutende negierte `ex`-quantifizierte Aussage ausgewertet:

   ⟨*Semantik/auswertung.pl*⟩+≡

   ```
   wahr(all(Var,F)) :- !, wahr(neg(ex(Var,neg(F)))).
   ```

3. Bei Aussagen `ex(Var,Formel)` wird nach einem (`!`: dem ersten) Objekt gesucht, das die Eigenschaft `Formel` hat:

⟨*Semantik/auswertung.pl*⟩+≡

```
wahr(ex(Var,Formel)) :-
    !, (fail, % debugging(typisieren), % aktivieren mit ?-
          current_predicate(typisiere,typisiere(_,_,_)),
          current_predicate(type,datenbank:type(_,_))
      -> ( var(Var) ->
              typisiere([],ex(Var,Formel),Getypt),
              wahr(Getypt)
         ; Var = V:Typ,
              objekt(V:Typ), wahr(Formel), !
         )
      ; objekt(Var:_), wahr(Formel), !
      ).
```

Die Suche in der endlichen Datenbank terminiert; die Typisierung (s.u.) schränkt sie auf einen Teilbereich ein.

4. Bei den arithmetischen Grundprädikaten `<`, `=<` und der Gleichheit `eq` wird Prolog aufgerufen:

   ⟨*Semantik/auswertung.pl*⟩+≡

   ```
   wahr((X < Y))  :- !, X < Y.
   wahr((X =< Y)) :- !, X =< Y.
   wahr(eq(X,Y))  :- !, X == Y.
   ```

5. Bei Anzahlaussagen werden die möglichen Lösungen gesucht (s.u.) und mit der vorgegebenen Zahl verglichen:

   ⟨*Semantik/auswertung.pl*⟩+≡

   ```
   wahr(anzahl(Var,Formel,Zahl)) :-
       !, antworten(qu(Var,Formel),Objekte),
       length(Objekte,N),
       Zahl =< N.
   ```

6. Bei Grundprädikaten wird in der Datenbank nach dem entsprechenden Faktum gesucht; falls es vorhanden ist, wird mit `!` die weitere Suche gestoppt:

⟨*Semantik/auswertung.pl*⟩+≡

```
wahr(Formel) :-
    functor(Formel,_Fn,_Arity),
%     predicate_property(Formel,  % unguenstig, wenn man M
%                        imported_from(datenbank)),
    call(Formel), !.
```

Der Wert wird stets nur auf *eine* Weise berechnet, wegen der `!` und da es *nicht* `wahr(F \/ G) :- wahr(F);wahr(G).` heißt.

Beispiele: Ja/Nein-Fragen als Aussagen in PL

Frage: *Entdeckte Herschel einen Planeten der Sonne?*

$\langle$*Anfrage in Prolog*$\rangle\equiv$

```
:- ['Anwendungen/Astronomie/datenbank',
    'Semantik/auswertung'].
?- wahr(ex(X,planet(X,sonne)
              & entdecken(herschel,X))).
```

Frage: *Entdeckte Herschel jeden Planeten der Sonne?*

$\langle$*Anfrage in Prolog*$\rangle+\equiv$

```
?- wahr(all(X,(planet(X,sonne)
               => entdecken(herschel,X)))).
```

Frage: *Hat Uranus einen größeren Durchmesser als die Erde?*

$\langle$*Anfrage in Prolog*$\rangle{+}{\equiv}$

```
?- wahr(ex(DE,ex(DU,durchmesser(DE,erde)
                   & durchmesser(DU,uranus)
                   & (DE < DU)))).
```

Bei dieser Formulierung bleibt offen, wieviele Durchmesser ein Stern hat.
Beachte: `durchmesser(D,S)` ist kein *Name*.

Ausgabe des Wahrheitswerts

Die Prädikate `wahr/1` und `falsch/1` geben aber den berechneten Wert nicht so aus, wie man vielleicht erwartet:

⟨*Beispiele* ⟩≡

```
?- wahr(ex(X,planet(X,sonne) & entdecken(galilei,X))).
No
?- wahr(ex(X,mond(X,jupiter) & entdecken(galilei,X))).
X = europa
Yes
```

Da auch *gebundene* prädikatenlogische Variable durch *freie* Prolog-Variable dargestellt werden, wird deren Belegung ausgegeben!

Zum Weiterrechnen geben wir die Wahrheitswerte als Antworten aus:

⟨*Semantik/auswertung.pl*⟩+≡

```
beantworte(PL_Aussage,Wert) :-
    wahr(PL_Aussage) -> Wert = ja ; Wert = nein.
```

Auswertung von Formeln der PL

Beim Auswerten einer Formel *mit freien Variablen* sollen die Belegungen, die die Formel wahr machen, als Ergebnis dienen.

Damit lassen sich die Formeln als Konstituentenfragen verstehen und die Belegungen als deren Antworten:

$$\textit{Welche}\ x\ \varphi\,? \quad \mapsto \quad \{a \mid \varphi[a/x]\ \textit{ist wahr}\} \subseteq \textit{Objekte}.$$

Aber: das Wahrheitsprädikat `wahr/1` nützt uns dabei nur, wenn wir die freien PL-Variablen belegen, *bevor* wir es aufrufen:

$\langle$*Test, ob die Belegung die Formel erfüllt*$\rangle \equiv$

```
?- astronom(E), % Prolog belegt E
   wahr(ex(Y,entdecken(E,Y) & mond(Y,jupiter))).

E = nicholson
Y = ananke ;   ...
```

Wie sammeln wir die möglichen Werte?

Prolog: Sammeln aller Lösungen eines Ziels

setof(+Muster,+Ziel,-Liste)

sucht alle Beweise von `Ziel`, wobei freie Prolog-Variablen belegt werden, und sammelt *pro Belegung der Variablen von* `Ziel`*, die nicht in* `Muster` *vorkommen*, die verschiedenen Instanzen von `Muster` der Beweise von `Ziel` in der `Liste`:

⟨*Lösungen mit* `setof`⟩≡

```
?- setof(E,(astronom(E),
            wahr(ex(Y,entdecken(E,Y)
                      & mond(Y,jupiter)))),
         Es).

E = _G147
Y = carme              % frei im Ziel, nicht im Muster
Es = [melotte] ;
```

⟨*Lösungen mit* `setof`⟩+≡

```
E = _G147
Y = ananke
Es = [nicholson] ;

E = _G147
Y = himalia
Es = [perrine] ;

E = _G147
Y = europa
Es = [galilei] ; No
```

Schlecht: Da `Y` eine im Muster `E` nicht vorkommende Prolog-Variable ist, werden für jeden Astronomen und den laut `db/5` *ersten* von ihm entdeckten Jupitermond seine Entdecker gesammelt.

setof(+Muster,+Variable^Ziel,-Liste)

Sollen die Ergebnisse unabhängig vom Wert der freien Prolog-Variablen gesammelt werden, muß man diese Variablen im Ziel durch `X^Y^Z^Ziel` „existentiell quantifizieren“:

⟨*Lösungen nach Prolog-Quantifizierung freier Variablen*⟩≡

```
?- setof(E,(astronom(E),
               Y^wahr(ex(Y,entdecken(E,Y)
                            & mond(Y,jupiter)))),
            Es).


E = _G147
Y = _G150  % Ex-quanfizierte Prolog-Variable
Es = [galilei, melotte, nicholson, perrine]
```

Dieses Ergebnis wollen wir, aber die Prolog-Quantifizierung ist (aus einem NL-Syntaxbaum) zu umständlich zu konstruieren.

Bem. Diese Prolog-$\exists$-Quantifizierung kann nicht in normalen Zielen verwendet werden, nur innerhalb `setof/3`, `bagof/3`.

findall(+Muster,+Ziel,-Liste)

gibt alle alle Spezialisierungen des Musterterms, für die Prolog das `Ziel` beweisen kann, in `Liste` aus, wobei Variable des Ziels, die im Muster nicht auftreten, implizit Prolog-Quantifiziert werden. Wenn sie verschieden belegt werden können, kann es zu wiederholten Instanzen von `Muster` kommen:

⟨*Suche mit* `findall`⟩≡

```
?-  findall(E,(astronom(E),
               wahr(ex(Y,entdecken(E,Y)
                         & mond(Y,jupiter)))),
            Es).


E = _G147
Y = _G150
Es = [nicholson, melotte, nicholson, galilei,
      galilei, perrine, galilei, galilei]
```

Da Nicholson zwei und Galilei vier Jupitermonde entdeckt haben, treten sie mehrfach in der Ergebnisliste auf.

Wir brauchen uns hier um die Prolog-Quantifizierung der gebundenen PL-Variablen nicht zu kümmern, müssen aber das Ergebnis in eine Menge zusammenfassen: das geht mit `sort/2`:

⟨*Suche mit* `findall` *und* `sort`⟩≡

```
?-  findall(E,(astronom(E),
               wahr(ex(Y,entdecken(E,Y)
                         & mond(Y,jupiter)))),
             Es),
    sort(Es,Ergebnis).

E = _G147 ...
Ergebnis = [galilei, melotte, nicholson, perrine]
```

Wert einer PL-Formel mit freien Variablen

Als Wert einer PL-Formel berechnen wir die Antwortmenge

$$\textit{Welche } x\, \varphi\, ? \quad \mapsto \quad \{a \mid \varphi[a/x] \textit{ ist wahr}\} \subseteq \textit{Objekte}.$$

also durch `findall` und `sort`:

$\langle$*Wert einer Formel: Antwortmenge*$\rangle\equiv$

```
?- findall(E,(astronom(E), % Prolog belegt E
              wahr(ex(Y,entdecken(E,Y)
                         & mond(Y,jupiter)))),Es),
   sort(Es,Ergebnis).

E = _G147
Y = _G150
Es = [nicholson, melotte, nicholson,
      galilei, galilei, perrine, galilei, galilei]
Ergebnis = [galilei, melotte, nicholson, perrine]
```

Eine W-Frage kann *beschränkt* sein, wie bei `welcher Astronom`, oder *unbeschränkt*, wie bei `wer`,`was`.

Wir typisieren die Formel, um die Suche auf den Teilbereich zu beschränken, der dem Typ der Suchvariablen entspricht:

⟨*Semantik/auswertung.pl*⟩+≡

```
antworten(qu(Var,Formel), As) :-
    (debugging(typisieren), % aktivieren mit ?- debug(typisie
        current_predicate(typisiere,typisiere(_,_,_))
    -> ( var(Var)
        -> typisiere([],qu(Var,Formel),qu(Var:Typ,Fml)),
            antworten(qu(Var:Typ,Fml), As)
        ; Var = V:Typ,
            findall(V,(objekt(V:Typ), wahr(Formel)),Ws),
            sort(Ws,As)
        )
    ; findall(Var,(objekt(Var:_), wahr(Formel)),Ws),
        sort(Ws,As)
    ).
```

Beispiel: W-Frage

Frage: *Welche Astronomen haben einen Mond des Jupiter entdeckt?*

⟨*Anfrage in Prolog*⟩+≡

```
?- antworten(qu(X,astronom(X) & ex(Y,(mond(Y,jupiter)
                                      & entdecken(X,Y)))),
             As).


X = _G147
Y = _G149
As = [galilei, melotte, nicholson, perrine]
```

Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing)
- Semantik:
  1. Datenbank (Objekte und Grundrelationen)
  2. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  3. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

Syntaxbaum $t \mapsto$ PL-Formel $\varphi_t(x, \ldots)$

Zur Berechnung der Bedeutung von Aussagen und Fragen fehlt uns von den drei Schritten noch der mittlere:

1. NL-Aussage $\alpha \quad \mapsto \quad$ Syntaxbaum t,
2. Syntaxbaum $t \quad \mapsto \quad$ logische Formel φ,
3. PL-Formel $\varphi(x) \quad \mapsto \quad$ Wert $\{a \in D \mid \textit{wahr}(\varphi[a/x])\}$

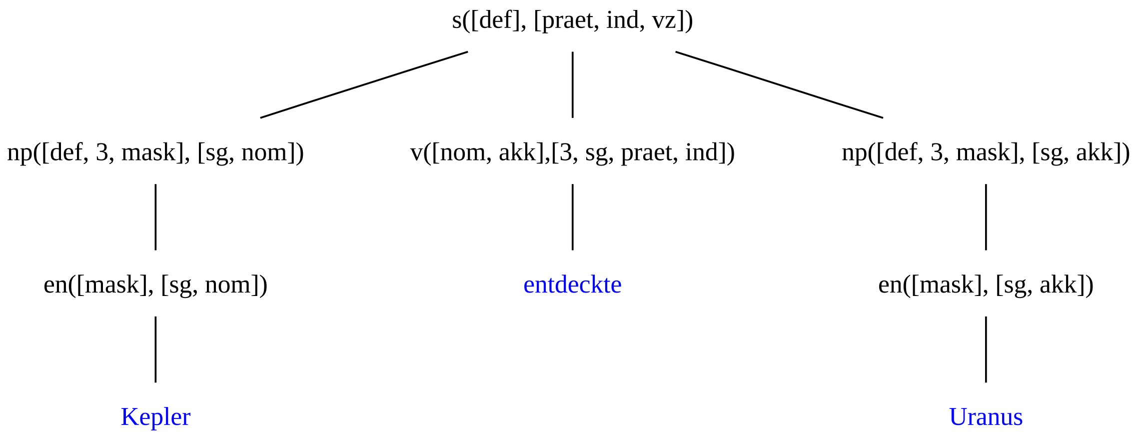
Wie berechnet man aus dem Syntaxbaum die passende Formel?

Was ist „die" passende Formel?

Atomare Aussage:

Aussage: `Kepler entdeckte Uranus.`

Baum:



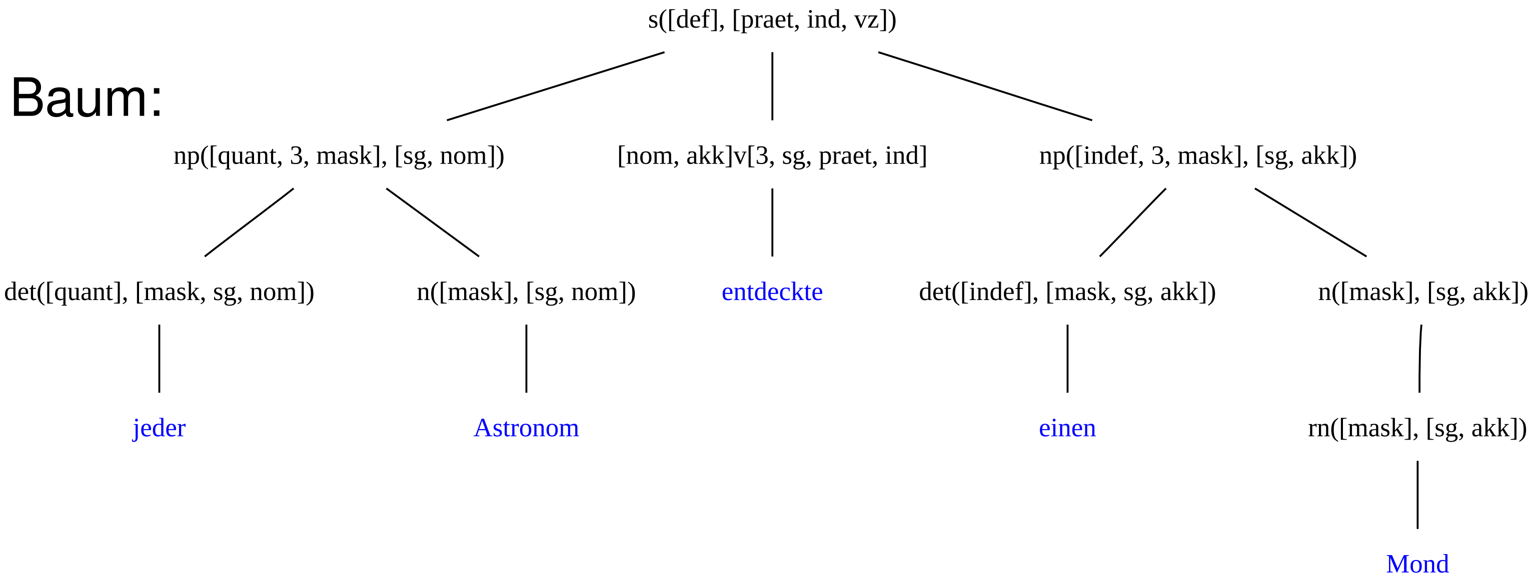
Formel: `entdecken(kepler,uranus).`

Dasselbe sollte bei anderen Wortstellungen herauskommen.

Quantifizierte Nominalphrasen:

Aussage: `Jeder Astronom entdeckte einen Mond.`

Baum:



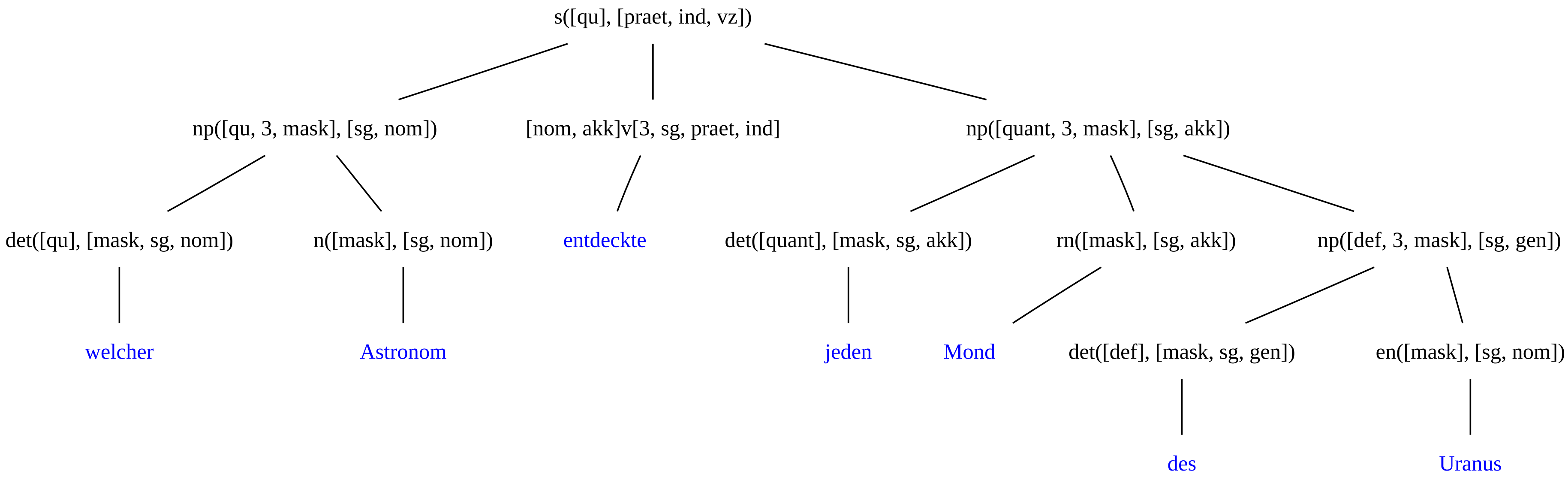
2 Formeln, da der Satz zweideutig ist:

```
all(X,astronom(X) => ex(Y,mond(Y) &
                              entdecken(X,Y))).

ex(Y,mond(Y) & all(X,astronom(X) =>
                              entdecken(X,Y))).
```

W-Fragen und NP-Attribute:

Aussage: `Welcher Astronom entdeckte jeden Mond des Uranus?`

Baum:



Formel:
```
qu(X, astronom(X) &
      all(Y,mond(Y,uranus) => entdecken(X,Y)))
```

Was ist der schwierige Teil?

Natürliche Sprache:

- Verben bedeuten eine Relation zwischen Individuen (oder Sachverhalten),
- Verbargumente sind Nominalphrasen (oder Sätze),
- Quantoren sind Teil der Nominalphrasen (NPs),
- quantifizierte NPs bedeuten *nicht* Individuen,
- der Wirkungsbereich der Quantoren ist nicht eindeutig.

Prädikatenlogik (erster Stufe):

- Prädikate bedeuten Relationen zwischen Individuen,
- alle Prädikatargumente (Terme) bedeuten Individuen,
- Quantoren sind Teil der Formeln (nicht der Terme),
- Wirkungsbereich eines Quantors ist die folgende Formel.

Bedeutung einer quantifizierten NP?

Aristoteles: NPs haben keine Bedeutung; ihre Quantoren „bedeuten" Beziehungen zwischen den Nomenbedeutungen

$$\begin{array}{rcccl}
(\textit{jeder } M \textit{ ist ein } L) & \mapsto & M\ a\ L & \equiv & M \subseteq L, \\
(\textit{eín } M \textit{ ist ein } L) & \mapsto & M\ i\ L & \equiv & M \cap L \neq \emptyset, \\
(\textit{eín } M \textit{ ist kein } L) & \mapsto & M\ o\ L & \equiv & M \not\subseteq L, \\
(\textit{kein } M \textit{ ist ein } L) & \mapsto & M\ e\ L & \equiv & M \cap L = \emptyset.
\end{array}$$

Aber das ist unvollständig: Quantoren in Objektposition, andere Verben?

Montague: NPs bedeuten Funktionen, die „einem Satz mit einem fehlenden *Individuen*namen" einen Wahrheitswert geben.

Für einfache Aussagen (Verb mit Komplementen):

$$(\ldots(\textit{Quantor } N)\ldots) \quad \mapsto \quad \textit{Quantor } x \in N\ (\ldots x \ldots)$$

Bei mehreren NP's muß man das wiederholt anwenden, z.B.:

$$\begin{array}{rcl}
((\textit{jeder } M)\ \textit{singt}\ (\textit{ein } L)) & \mapsto & \forall x \in M\ (x \textit{ singt ein } L) \\
 & \mapsto & \forall x \in M\ (\exists y \in L\ (x \textit{ singt } y))
\end{array}$$

Mathematisch ausgedrückt:

$$P(Q\ N) \mapsto Qx \in N\ P(x) := \begin{cases} \forall x(N(x) \to P(x)), & \text{falls } Q = \forall \\ \exists x(N(x) \wedge P(x)), & \text{falls } Q = \exists \end{cases}$$

Die Bedeutung von $(Q\ N)$ ist die Funktion, die jeder einfachen Aussage $P(x)$ über Individuen x den Wahrheitswert von $Qx \in N\ P(x)$ zuordnet.

Daher wollen wir übersetzen:

$$\begin{array}{rcl} (\textit{alle}\ N) & \mapsto & (P \mapsto \forall x(N(x) \to P(x))) \\ (\textit{ein}\ N) & \mapsto & (P \mapsto \exists x(N(x) \wedge P(x))) \end{array}$$

Problem: P ist i.a. kein Grundprädikat, also $P(x)$ keine Formel

Wir brauchen eine flexiblere Sprache, in der man auch unbekannte oder komplexe Prädikate auf Objektbezeichnungen anwenden kann.

λ-Terme

λ-Terme sind eine Schreibweise für Funktionen und Daten:

$$
\begin{array}{rcll}
s, t & := & x & \text{Variable} \\
 & \mid & c & \text{Konstante} \\
 & \mid & (t \cdot s) & \text{Anwendung von } t \text{ auf } s \\
 & \mid & \lambda x\, t & \text{Funktionsabstraktion von } t \text{ bzgl. } x
\end{array}
$$

Beachte: bei der „Anwendung“ $(t \cdot s)$ sind Funktion und Argument gleichrangig, man kann bei beiden eine Variable haben – anders als bei $f(s)$ in Prolog und in der Prädikatenlogik.

Bei jeder Interpretation $\mathcal{D} = (D, \cdot^{\mathcal{D}}, c^{\mathcal{D}}, \ldots)$ sollte u.a. gelten:

$$(\lambda x\, t \cdot s)^{\mathcal{D}} = (\lambda x\, t)^{\mathcal{D}} \cdot^{\mathcal{D}} s^{\mathcal{D}} = t^{\mathcal{D}}[x/s^{\mathcal{D}}] = (t[x/s])^{\mathcal{D}}.$$

Das will man durch *Termvereinfachung*, $s \to t$, ausrechnen können, mit $s^{\mathcal{D}} = t^{\mathcal{D}}$ bei allen Interpretationen $\mathcal{D}$.

Termvereinfachung $s \longrightarrow t$

$$\frac{r \longrightarrow t}{(r \cdot s) \longrightarrow (t \cdot s)}\,(=_1) \qquad \frac{s \longrightarrow u}{(r \cdot s) \longrightarrow (r \cdot u)}\,(=_2) \qquad \frac{r \longrightarrow s}{\lambda x\, r \longrightarrow \lambda x\, s}\,(=_3)$$

$$\frac{y \notin frei(t)}{\lambda x\, t \longrightarrow \lambda y\; t[x/y]}\,(\alpha) \qquad \frac{}{(\lambda x\, t \cdot s) \longrightarrow t[x/s]}\,(\beta) \qquad \frac{x \notin frei(t)}{\lambda x\,(t \cdot x) \longrightarrow t}\,(\eta)$$

Weitere Regeln legen den Umgang mit Konstanten c fest.

Die syntaktische Einsetzung $t[x/s]$ ist so zu definieren, daß gebundene Variablen in t umbenannt werden, damit kein $y \in frei(s)$ in den Wirkungsbereich eines λy von t gerät.

Mit $s \longrightarrow^* t$ ist gemeint, daß man von s mit diesen Regeln (und gebundener Umbenennung) zu t kommen kann.

Syntaktische Einsetzung $t[x/s]$

Die *frei* in einem λ-Term vorkommenden Variablen sind:

$$
\begin{aligned}
\mathit{frei}(y) &:= \{y\},\\
\mathit{frei}(c) &:= \emptyset,\\
\mathit{frei}((s \cdot t)) &:= \mathit{frei}(s) \cup \mathit{frei}(t),\\
\mathit{frei}(\lambda x\, t) &:= \mathit{frei}(t) \setminus \{x\}.
\end{aligned}
$$

Die *Ersetzung (der freien Vorkommen) von x in t durch s* definiert man induktiv über den Aufbau von t:

$$
\begin{aligned}
y[x/s] &:= \begin{cases} s, & \text{falls } y \equiv x,\\ y, & \text{falls } y \not\equiv x \end{cases}\\
c[x/s] &:= c\\
(r \cdot t)[x/s] &:= (r[x/s] \cdot t[x/s])\\
\lambda y\, t\, [x/s] &:= \begin{cases} \lambda y\, t, & \text{falls } y \equiv x,\\ \lambda y\, (t[x/s]), & \text{sonst, falls } y \notin \mathit{frei}(s),\\ \lambda z\, (t[y/z][x/s]), & \text{sonst, mit } z \notin \mathit{frei}(\lambda y\, t \cdot s) \end{cases}
\end{aligned}
$$

Beispiele für $t[x/s]$

$$\begin{array}{rcl}
\lambda x(y\cdot x)[x/\lambda z(c\cdot z)] & = & \lambda x(y\cdot x) \\
\lambda x(y\cdot x)[y/\lambda z(c\cdot z)] & = & \lambda x(\lambda z(c\cdot z)\cdot x) \quad \text{da } x\notin \mathit{frei}(s) \\
 & & t[x/s]=t, \text{ da } x\notin \mathit{frei}(t) \\
\lambda x(y\cdot x)[y/\lambda z(x\cdot z)] & = & \lambda z((y\cdot x)[x/z][y/\lambda z(x\cdot z)]) \\
 & = & \lambda z((y\cdot z)[y/\lambda z(x\cdot z)]) \\
 & = & \lambda z(\lambda z(x\cdot z)\cdot z) \\
 & =_\alpha & \lambda u(\lambda z(x\cdot z)\cdot u) \\
\lambda y\,\lambda x\,(y\cdot x)\cdot\lambda z(x\cdot z) & \to^* & \lambda u(\lambda z(x\cdot z)\cdot u)
\end{array}$$

Die *Redexe* eines Terms sind die Teilterme, auf die die jeweiligen Reduktionsregeln angewendet werden können.

Durch Anwenden der Reduktionsregeln können neue Redexe entstehen: in

$$
\begin{array}{rcl}
s & = & \lambda x((x \cdot y) \cdot (x \cdot y)) \cdot \lambda v\, v \\
 & \to_\beta & ((x \cdot y) \cdot (x \cdot y))[x/\lambda v\, v] \\
 & = & ((\lambda v\, v \cdot y) \cdot (\lambda v\, v \cdot y)) \ =: t \\
 & \to_\beta & (y \cdot (\lambda v\, v \cdot y)), \\
 & \to_\beta & (y \cdot y),
\end{array}
$$

enthält der Ausgangsterm s *einen* β-Redex und der durch die Reduktion entstandene Term t *zwei* β-Redexe.

Die „Vereinfachung“ kann sogar divergieren!

Aber wenn man Variablen mit *Typen*

$$\sigma, \tau := \alpha \mid \textit{bool} \mid \textit{int} \mid (\sigma \to \tau)$$

versieht und nur „typkorrekte“ Anwendung $(t^{\sigma\to\tau} \cdot s^{\sigma})^{\tau}$ zuläßt, terminiert die Vereinfachung immer (in einer *Normalform*).

Zurück: Jetzt können wir –mit $P \cdot x$ statt $P(x)$– übersetzen:

$$\begin{array}{rcl} (\textit{alle}\ N) & \mapsto & \lambda P\, \forall x(N(x) \Rightarrow (P \cdot x)) \\ \textit{alle} & \mapsto & \lambda N\, \lambda P\, \forall x((N \cdot x) \Rightarrow (P \cdot x)) \end{array}$$

unabhängig davon, ob P, N Prädikate oder Formeln sind. Junktoren und Quantoren könnten wir als Konstante verstehen:

$$(\varphi \Rightarrow \psi) := ((\Rightarrow \cdot \varphi) \cdot \psi), \qquad \forall x \varphi := \forall \cdot \lambda x\, \varphi.$$

Reduktion von λ-Termen

Die Vereinfachung von λ-Termen erfolgt nach der Strategie:

1. Versuche eine β-Reduktion, $t \to_\beta s$, und wenn das ging, reduziere s weiter; sonst ist t das Ergebnis (Normalform).

2. In der β-Reduktion wird bei Anwendungen $t \cdot s$ der Form $\lambda x\, r \cdot s$ zu $r[x/s]$ und dann dies, sonst erst t, dann s vereinfacht. Bei Abstraktionen $\lambda x\, r$ wird r vereinfacht, bei Termen $f(t_1 \ldots, t_n)$ nacheinander $t_1, \ldots, t_n$.

3. Bei $r[x/s]$ werden erst die in r gebundenen Variablen umbenannt, bevor x durch s ersetzt wird. (α-Reduktion)

⟨*Semantik/lambdaTerme.pl*⟩≡

```
% :- module(lambda,[normalize/2]).

% Term := Var | Atom | (Term * Term) | lam(X,Term)
%         | Atom(Term,...,Term) (!)

normalize(T,Nf) :-
    beta(T,ConT,Tag),      % Einen Vereinfachungsschritt
    ( Tag = chd            % machen; falls der zu ConT =/= T
    -> normalize(ConT,Nf) % fuehrt, das weiter vereinfachen,
    ; Nf = ConT ).         % sonst ist ConT = T die Normalform

normalize_seq([T|Ts],[Nf|Nfs]) :-
    normalize(T,Nf),
    normalize_seq(Ts,Nfs).
normalize_seq([],[]).
```

β-Reduktion

$\langle$*Semantik/lambdaTerme.pl*$\rangle{+}{\equiv}$

```
% Beta-Reduktion beta(+Term,-Reduziert,-geändert?)

beta(X, X, not) :-  var(X), !.

beta(T*S, T*ConS, Tag) :-
    var(T),
    !, beta(S,ConS,Tag).

beta(lam(X,R)*S, RDup, chd) :-
    !, alpha(lam(X,R),lam(XDup,RDup)),
    XDup = S.          % simuliert R[X/S]
```

⟨*Semantik/lambdaTerme.pl*⟩+≡

```
beta(T*S, U, Tag) :-
     !, beta(T,ConT,TagT),
     ( TagT = chd -> U = ConT*S, Tag = chd
     ; beta(S,ConS,Tag), U = T*ConS ).

beta(lam(X,R), lam(X,ConR), Tag) :-
     !, beta(R,ConR,Tag).

beta(T, ConT, not) :- % für Atom(Term,..,Term)
     compound(T),
     !, T =.. [F|Args],
     normalize_seq(Args,Nfs),
     ConT =..[F|Nfs].

beta(T, T, not) :- !. % Konstante
```

α-Reduktion: Umbenennung gebundener Variablen

⟨*Semantik/lambdaTerme.pl*⟩+≡

```
% Alpha-Reduktion: alpha(+Term,-Umbenannt)

alpha(T, TDup) :-
    alpha_list(T,[],TDup), !.

alpha_list(V, L, VDup) :-
    var(V), !, rename(V,L,VDup).

alpha_list(lam(V,R), L, lam(New,RDup)) :-
    !, alpha_list(R,[(V,New)|L],RDup).

alpha_list(ex(V,R), L, ex(New,RDup)) :-
    !, alpha_list(R,[(V,New)|L],RDup).

alpha_list(all(V,R), L, all(New,RDup)) :-
    !, alpha_list(R,[(V,New)|L],RDup).
```

⟨*Semantik/lambdaTerme.pl*⟩+≡

```
alpha_list(T*S, L, TDup*SDup) :-
    !, alpha_list(T,L,TDup),
    alpha_list(S,L,SDup).

alpha_list(C, _, C) :-
    atomic(C), !.

alpha_list(T, L, TDup) :-  % für:  Atom(Term,...,Term)
    compound(T),
    !, T =.. [F|Ts],
    alphas(Ts,L,TsDup),
    TDup =.. [F|TsDup].

alphas([T|Ts], L, [TDup|TsDup]) :-
    alpha_list(T,L,TDup), alphas(Ts,L,TsDup).
alphas([],_L,[]).
```

Umbenennung von Variablen

⟨*Semantik/lambdaTerme.pl*⟩+≡

```
% rename(+Var, +Variablenpaare, -VarUmbenannt)

rename(V, [], V).
rename(V, [(Var,Dup)|_], Dup) :-
    V == Var, !.
rename(V, [_|C], Dup) :-
    !, rename(V,C,Dup).
```

Beispiele zur α- und β-Reduktion

Nur die λ-gebundenen Variablenvorkommen werden umbenannt:

$\langle$*Beispiel zur α-Reduktion*$\rangle\equiv$

```
?- alpha(X*lam(X,(Y*X)), Dup).
X = _G148
Y = _G147
Dup = _G148*lam(_G251, _G147*_G251)
```

Vereinfachung durch Einsetzungen für gebundene Variable:

$\langle$*Beispiel zur β-Reduktion*$\rangle\equiv$

```
?- normalize(lam(X,(X*Y)*(X*Y)) * lam(V,c*V), Nf).
X = _G147
Y = _G148
V = _G160
Nf = c*_G148 * (c*_G148)
```

λ-Terme in der CL-Literatur (Wenger u.a.)

Manchmal definiert die Literatur zur Computerlinguistik (z.B. Wenger, s.97f) die Reduktion von Lambda-Termen durch

⟨*Beispiele/pseudolambda.pl*⟩≡

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

wobei `X^P` unserem Term `lam(X,P)` und `beta(X^T,S,Q)` unserem `normalize(lam(X,T)*S,Q)` entsprechen soll.

Man will so die Einsetzung `t[x/s]` und Normalisierung auf die Unifikation von Prolog zurückführen. Beim Aufruf muß man ggf. dieselbe Variable `X` mehrfach binden und von strukturierten Argumenten abstrahieren:

⟨*Beispiel*⟩+≡

```
?- beta((X^N)^((X^VP)^all(X,N => VP)),Y^n(Y),Q).
X = _G147
N = n(_G147)
VP = _G151
Y = _G147
Q = (_G147^_G151)^all(_G147, (n(_G147) => _G151))
```

Wir schreiben stattdessen

⟨*Beispiel in Lambda-Notation*⟩≡

```
?- normalize(lam(M,lam(P,all(X,M*X => P*X)))
              * lam(Y,n(Y)),Q).
```

Aber: `beta/3` liefert i.a. ein Ergebnis mit falschen Bindungen oder Gleichheiten von Variablen, z.B. (SWI-Version 5.2.13 bzw. 5.6.47)

⟨*Fehler* ⟩≡

```
?- beta(X^t(X),s(X),Q).
X = s(s(s(s(s(s(s(s(s(s(...))))))))))  bzw. s(**)
Q = t(s(s(s(s(s(s(s(s(s(...))))))))))  bzw. t(s(**))
```

Dagegen liefert unsere Normalisierung den richtigen Wert:

⟨*Korrekte Auswertung*⟩≡

```
?- normalize(lam(X,t(X))*s(X),Q).
Q = t(s(X))
```

Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing)
- Semantik:
  1. Datenbank (Objekte und Grundrelationen)
  2. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  3. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

Beispiel: Semantik zu einem NP-Baum

Möglichkeit A: den Aufbau der logischen Formel in die Grammatikregeln einbauen (Kategorien mit Semantik-Ausgabe).

Lexikonregeln: `Kat(Sem) -> [Wort].`

Syntaxregeln: `Kat(Sem) --> Kat1(Sem1),..., KatK(SemK).`

Bei der Analyse von *jeder Astronom* wird ein λ-Term durch die Grammatikregeln „mit Semantik“ aufgebaut:

$$
\begin{array}{rcl}
n[\lambda x\, \textit{astronom}(x)] & \rightarrow & \textit{Astronom} \\
det[\lambda N\, \lambda P\, (\forall x(N \cdot x \rightarrow P \cdot x))] & \rightarrow & \textit{jeder} \\
np[D \cdot N] & \rightarrow & det[D]\; n[N]
\end{array}
$$

⟨*Beispiele/semNPA.pl*⟩≡

```
:- op(600,xfx,'=>').

n(lam(X,astronom(X))) --> ['Astronom'].
det(lam(N,lam(P,all(X,N*X => P*X)))) --> [jeder].

np(SemDet*SemN) --> det(SemDet), n(SemN).
```

⟨*Beispiel einer Analyse (lesbare Variable eingesetzt)*⟩≡

```
?- ['Beispiele/semNPA.pl','Semantik/lambdaTerme.pl'].
?- np(Sem,[jeder,'Astronom'],[]),
   normalize(Sem,Nf).

Sem = lam(N,lam(P,all(X,N*X=>P*X)))*lam(Z,astronom(Z))
Nf = lam(Q,all(X,astronom(X)=>Q*X))
```

Beispiel: Semantik zu einem NP-Baum

Möglichkeit B: Der Aufbau der Formel wird *nach* der Syntaxanalyse durch `sem(+Syntaxbaum,-LamTerm)` berechnet:

Vorteil: wir müssen die Grammatikregeln nicht ändern.

Zum Syntaxbaum von *jeder Astronom* muß ein λ-Term aus den λ-Termen der Teilbäume konstruiert werden:

$\langle$*Beispiele/semNPB.pl*$\rangle\equiv$

```
:- op(600,xfx,'=>').
sem([np([quant, 3, _], [sg, _]), Det, N], SemNP) :-
     sem(Det,SemDet), sem(N,SemN),
     SemNP = SemDet * SemN.
sem([det([quant], [_, sg, _]), [jeder]],
     lam(N,lam(P,all(X,N*X => P*X)))).
sem([n([mask], [sg, _]), ['Astronom']],
     lam(X,astronom(X))).
```

⟨*Beispiel einer Analyse (mit lesbaren Variablen)*⟩≡

```
?- ['Beispiele/semNPB.pl','Semantik/lambdaTerme.pl'].
?- Baum = [np([quant,3,mask],[sg,nom]),
             [det([quant],[mask,sg,nom]), [jeder]],
             [n([mask],[sg,nom]), ['Astronom']]],
   sem(Baum,SemTerm),
   normalize(SemTerm,Normalform).

SemTerm = lam(V, lam(W, all(X, V*X=>W*X)))
          * lam(Z, astronom(Z))
Normalform = lam(Y, all(X, astronom(X)=>Y*X))
```

Quantorenskopus

Für eine mehrfach verzweigende Regel wie $S \to$ *NP TV NP* wird die Semantik $SemS$ in

$$S[SemS] \to \textit{NP}[A]\ \textit{TV}[R]\ \textit{NP}[B]$$

aus den λ-Termen A, R, B aufgebaut.

Wird bei *TV* das transitive Verb r benutzt, so soll

$$R = \lambda x \lambda y\, r(x, y)$$

sein, also $r(\tilde{x}, \tilde{y})$ die Normalform von $((R \cdot \tilde{x}) \cdot \tilde{y})$.

Sind die *NP* quantifizierte Nominalphrasen, $(\tilde{Q}\ \tilde{N})^{nom}$ und $(Q\ N)^{akk}$, so soll $SemS$ mit logischen Quantoren beginnen und an die Stelle der *NP*s sollen Individuenvariable zum Verb treten (erste Argumentstelle für das Subjekt), also

$$\tilde{Q}x \in \tilde{N}\, Qy \in N\ r(x, y) \quad \text{und} \quad Qy \in N\, \tilde{Q}x \in \tilde{N}\ r(x, y).$$

entstehen können.

Stehen die *NP* für $(\tilde{Q}\ \tilde{N})[A]$, $(Q\ N)[B]$, so brauchen wir zum Aufbau der Formeln

$$A \cdot Z \to_\beta (\tilde{Q}x \in \tilde{N})(Z \cdot x) \quad \text{bzw.} \quad A = \lambda Z(\tilde{Q}x \in \tilde{N})(Z \cdot x)$$

$$B \cdot P \to_\beta (Qy \in N)(P \cdot y) \quad \text{bzw.} \quad B = \lambda P(Qy \in N)(P \cdot y)$$

und zwei $S[$*SemS*$]$ in den Regeln:

$$\begin{array}{rcl} S[A \cdot \lambda x(B \cdot \lambda y\,((R \cdot x) \cdot y))] & \to & \textit{NP}[A]\ \textit{TV}[R]\ \textit{NP}[B] \\ S[B \cdot \lambda y\,(A \cdot \lambda x\,((R \cdot x) \cdot y))] & \to & \textit{NP}[A]\ \textit{TV}[R]\ \textit{NP}[B] \end{array}$$

Übersetzung NL nach PL:
Syntaxbaum $\mapsto$ logische Formel

Wir brauchen zuerst die Operatordeklarationen für die logischen Junktoren und eine Hilfsfunktion:

⟨*Semantik/sem.pl*⟩≡

```
:- op(500,yfx,&), op(600,yfx,'\/'),
     op(600,xfx,'=>'), op(600,xfx,'<=>').

kleinschreibung(Wort,Klein) :-
     atom(Wort),
     name(Wort,[C|Chars]),
     (member(C,"ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ")
     ->  K is C+32,
         name(Klein,[K|Chars])
     ; Klein = Wort).
```

A. Bedeutung der Wörter

Von der Vollform eines Worts im Syntaxbaum gehen wir zur Stammform und von dort zu seiner Bedeutung:

1. Eigennamen bedeuten entsprechende Konstanten der Datenbank:

   $\langle$*Semantik/sem.pl*$\rangle+\equiv$

   ```
   sem([en(Art,Form), [Vollform]],LamTerm) :-
           wort(Stammform, en(Art,Form), Vollform),
           kleinschreibung(Stammform,EN),
           LamTerm = EN.
   ```

2. Relationsnomen und transitive Verben bedeuten Beziehungen zwischen Objekten. Sie werden zu λ-Termen, die bei Anwendung auf eine bzw. zwei Konstante die passende atomare Formel ergeben:

$\langle$*Semantik/sem.pl*$\rangle+\equiv$

```
sem([rn(Art,Form), [Vollform]],LamTerm) :-
      wort(Stammform, rn(Art,Form), Vollform),
      kleinschreibung(Stammform,Stamm),
      Formel =.. [Stamm,X,Y],
      LamTerm = lam(X,lam(Y,Formel)).

sem([v([nom,akk],Form),[Vollform]],LamTerm) :-
      kleinschreibung(Vollform,VollformKl),
      wort(Stammform,v([nom,akk],Form),VollformKl),
      Formel =.. [Stammform,X,Y],
      LamTerm = lam(X,lam(Y,Formel)).
```

3. Absolute Nomen bedeuten Eigenschaften:

$\langle$*Semantik/sem.pl*$\rangle+\equiv$

```
sem([n(Art,Form),[Vollform]],LamTerm) :-
        wort(Stammform, n(Art,Form), Vollform),
        kleinschreibung(Stammform,Stamm),
        Formel =.. [Stamm,X],
        LamTerm = lam(X,Formel).
```

Werden sie aus Relationsnomen abgeleitet, bedeuten sie die Projektion des Relationsnomens:

$\langle$*Semantik/sem.pl*$\rangle+\equiv$

```
sem([n(Art,Form),[rn(Art,Form),[Vollform]]],LamTerm) :-
        wort(Stammform, rn(Art,Form), Vollform),
        kleinschreibung(Stammform,Stamm),
        Formel =.. [Stamm,X],
        LamTerm = lam(X,Formel).
```

4. Artikel ($\neq$ `qu`) bedeuten Funktionen, die zwei Eigenschaften von Objekten einen Wahrheitswert zuordnen:

$$
\begin{array}{rcl}
\textit{jeder}\ N\ \ \textit{VP} & = & (N \subseteq \textit{VP})\,? \\
\textit{ein}\ N\ \ \textit{VP} & = & (N \cap \textit{VP} \neq \emptyset)\,? \\
k\ N\ \ \textit{VP} & = & (\,|N \cap \textit{VP}| \geq k)\,? \\
\textit{der}\ N\ \ \textit{VP} & = & (|N| = 1 \wedge\ N \cap \textit{VP} \neq \emptyset)\,?
\end{array}
$$

Sie werden zu λ-Termen, die bei Anwendung auf zwei Eigenschaften eine passende Formel liefern:

$\langle$*Semantik/sem.pl*$\rangle+\equiv$

```
sem([det([quant],_Form),_Vollform],
    lam(N,lam(P,all(X,N*X => P*X)))).

sem([det([indef],_Form),_Vollform],
    lam(N,lam(P,ex(X,N*X & P*X)))).
```

$\langle$*Semantik/sem.pl*$\rangle+\equiv$

```
sem([det([def],[_,pl,_]),[Vollform]],
     lam(N,lam(P, anzahl(X,N*X & P*X,Zahl)))) :-
     anzahl(Vollform,Zahl), !.
sem([det([def], [_,sg,_]),_Vollform],
     lam(N,lam(P,
     ex(Y,all(X,eq(X,Y) <=> N*X) & P*Y)))).
```

5. Interrogativartikel bedeuten Funktionen, die zwei Eigenschaften von Objekten die Menge derjenigen Objekte, die korrekte Antworten bedeuten, zuordnen:

$$\textit{welcher}\ N\ \textit{VP} \quad = \quad \{a \in N \mid a \in VP\}$$

$\langle$*Semantik/sem.pl*$\rangle+\equiv$

```
sem([det([qu], _Form),_Vollform],
     lam(N,lam(P,qu(X,N*X & P*X)))).
```

6. Adjektive, die eine Vergleichsrelation bedeuten, werden zu λ-Termen, die bei Anwendung auf zwei Zahlen eine Formel mit dem entsprechenden Prolog-Prädikat bilden:

   ⟨*Semantik/sem.pl*⟩+≡

   ```
   sem([a([als(nom)],[komp]),[kleiner]],
           lam(X,lam(Y,(X < Y)))). % Fehlermeldung?
   sem([a([als(nom)],[komp]),[größer]],
           lam(X,lam(Y,(Y < X)))). % > kein Grundprädikat
   ```

   (Für die attributive Verwendung der Adjektive muß man diese Bedeutung über die Stammform holen.)

7. Relativ- und Interrogativpronomen bedeuten Eigenschaften von Individuen, die ggf. als erfragt markiert werden:

   ⟨*Semantik/sem.pl*⟩+≡

   ```
   sem([pron([rel],_Form),_Rel], lam(P,lam(X,P*X))).
   sem([pron([qu],_Form),_Pron], lam(P,qu(X,P*X))).
   ```

7. Relativierende Possessivpronomen bedeuten auf ein korreliertes Individuum „verschobene“ Eigenschaften:

   ⟨*Semantik/sem.pl*⟩+≡

   ```
   sem([poss([rel],[_Gen,_Num,gen]),_Poss],
       lam(RN,lam(P,lam(Y,ex(X,RN*X*Y & P*X))))).
       % nur bei RNsg
   ```

8. Hilfsverben `v([sein],Form)` haben keine eigenständige (nur „synkategorematische“) Bedeutung; keine Klausel.

Laden der Grammatik und Semantik

Mit `?- [semantik,astronomie]` werden die nötigen Dateien geladen.

⟨*astronomie.pl*⟩+≡

```
:- ['Anwendungen/Astronomie/datenbank.pl'].
```

⟨*semantik.pl*⟩≡

```
:- style_check(-discontiguous).
:- [grammatik],
   ['Semantik/auswertung.pl',
    'Semantik/lambdaTerme.pl',
    'Semantik/sem.pl',
    'Semantik/type.pl'].   % spaeter
```

Die `sem/2`-Klauseln für zusammengesetzte Ausdrücke und die Programme zur Typisierung folgen unten.

Zum Testen dient ein Parseraufruf mit Ausgabe der aus dem Syntaxbaum berechneten Lambda-Terme:

$\langle$*semantik.pl*$\rangle+\equiv$

```
parses :-
     write('Beende die Eingabe mit <Punkt><Return>'),
     nl,read_sentence(user,Sentence,[]),
     tokenize(Sentence,Atomlist),
     startsymbol(S),
     addTree:translate1(S,Term,Baum),
     addDifflists:translate(Term,ExpTerm,Atomlist,[]),
     nl,write('Aufruf: '), portray_clause(ExpTerm),
     call(ExpTerm), write('Baum:   '),
     nl,writeTrLam(Baum,4),nl,
     fail.
parses.
```

Ausgabe des Baums mit den Lambda-Termen

Mit `writeTrLam` schreiben wir den Syntaxbaum formatiert und dabei unter die Knoten die entsprechenden Lambda-Terme:

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLam([Wurzel|Bs],Einrueckung) :-
    tab(Einrueckung),writeq(Wurzel),
    (sem([Wurzel|Bs],LamTerm)
    -> nl,tab(Einrueckung),
        write('+ '),writen(LamTerm)
    ; true % für Wörter ohne Bedeutung
    ),
    (Bs = [[Blatt]]
    -> tab(1), writeq(Blatt)
    ; Einrueckung2 is Einrueckung + 3,
      writeTrsLam(Bs,Einrueckung2)
    ).
```

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLam([],Einrueckung) :-
    tab(Einrueckung), writeq([]).

writeTrsLam([Baum|Baeume],Einrueckung) :-
    nl,writeTrLam(Baum,Einrueckung),
    writeTrsLam(Baeume,Einrueckung).
writeTrsLam([],_).
```

Variable schreiben wir lesbar als `X,Y,Z,A1,B1,...` durch:

⟨*Parser/showTree.pl*⟩+≡

```
writen(Term) :- \+ \+((numbervars(Term,23,_),
           write_term(Term,[numbervars(true)]))).
```

Beispiel: Syntaxbaum mit Semantik

Mit den unten folgenden Klauseln von `sem/2` wird dann die Analyse mit semantischer Information so ausgegeben:

⟨*Beispiel einer Analyse*⟩+≡

```
?- [semantik,astronomie].
?- parses. jeder Astronom.

Aufruf: np([A,B,C],[D,E],F,[jeder,'Astronom'],[]).
Baum:
    np([quant,3,mask],[sg,nom])
    + lam(X, all(Y, astronom(Y)=>X*Y))
       det([quant],[mask,sg,nom])
       + lam(X, lam(Y, all(Z, X*Z=>Y*Z))) jeder
       n([mask],[sg,nom])
       + lam(X, astronom(X)) 'Astronom'
```

Für jede Startkategorie wird eine Analyse versucht, und wenn die gelingt, wird dazu die Bedeutung berechnet.

Nach dem folgenden Kompositionsprinzip wird die Bedeutung von den Blättern zur Wurzel des Syntaxbaums berechnet.

Sie ist also nur von der Form, nicht vom Kontext des Ausdrucks abhängig; die λ abstrahieren u.a. vom Kontext (in einem Satz).

B. Bedeutung zusammengesetzter Ausdrücke

Kompositionsprinzip: Was ein zusammengesetzter Ausdruck bedeutet, hängt vom Syntaxbaum B und den Bedeutungen der Teilausdrücke ab und wird durch einen λ-Term angegeben.

Diesen λ-Term t (ggf. mehrere) berechnet man *rekursiv*:

1. bestimme die Verzweigungsform R an der Wurzel von B,
2. berechne die λ-Terme t_i seiner direkten Teilbäume B_i,
3. konstruiere aus $t_1, \ldots, t_n$ den λ-Term t für B je nach R.

Wir definieren —nur für die wichtigsten Konstruktionen!— das Prädikat `sem(+Syntaxbaum,-LamTerm)` je nach der Form

```
[Kategorie(Art,Form), Teilbaum1, ..., TeilbaumN]
```

des Syntaxbaums des zusammengesetzten Ausdrucks:

1. Nominalphrasen (Kopf `n`, nicht `rn`) ohne Relativsatz:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([_,3,_],[_,_]),
          Det, [n(Art,Form),Vf]], SemNP) :-
      sem(Det,SemDet),
      Vf \= km, sem([n(Art,Form),Vf],SemN),
      normalize(SemDet * SemN,SemNP).
```

⟨*Beispiel*⟩+≡

```
?- parses. jeder Astronom.
lam(A, all(B, astronom(B)=>A*B))
?- parses. die Sonne.
lam(A, ex(B, all(C, eq(C, B)<=>sonne(C))&A*B))
```

2. Eigennamen als Nominalphrase bedeuten „die Anwendung von Prädikaten auf die Namenskonstante“:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([def,3,Gen],[Num,Kas]),
       [en([Gen],[Num,Kas]),EN]], SemNP) :-
    sem([en([Gen],[Num,Kas]),EN],SemEN),
    SemNP = lam(P,P*SemEN).
```

⟨*Beispiel*⟩+≡

```
?- parses. Kepler.
np([def, 3, mask], [sg, nom])
+ lam(X, X*kepler)
   en([mask], [sg, nom])
   + kepler 'Kepler'
```

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([def,3,Gen],[Num,_Kas]),
       [det([def],_),_],
       [en([Gen],[Num,nom]),EN]],
    SemNP) :-
    !, sem([en([Gen],[Num,nom]),EN],SemEN),
    SemNP = lam(P,P*SemEN).
```

⟨*Beispiel*⟩+≡

```
?- parses. der Uranus.
lam(A, A*uranus)
```

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([def,3,Gen],[Num,Kas]),
       [det([def],_),_],
       [n([Gen],[Num,Kas]),N],
       [en([Gen2],[Num,nom]),EN]], SemNP) :-
    !, sem([n([Gen],[Num,Kas]),N],SemN),
    sem([en([Gen2],[Num,nom]),EN],SemEN),
    normalize(lam(P,SemN*SemEN & P*SemEN),SemNP).
```

⟨*Beispiel*⟩+≡

```
?- parses. der Astronom Kepler.
lam(A, astronom(kepler)&A*kepler)
```

3. Relativ- und Interrogativ-Pronomen als Nominalphrase:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([Def,3,Gen],[Num,Kas]),
        [pron([DefP],[Gen,Num,Kas]),Pron]],
      SemNP) :-
      ( DefP = rel, Def = rel(Gen,Num)
      ; DefP = qu, Def = qu ),
      sem([pron([DefP],[Gen,Num,Kas]),Pron],
          SemNP).
```

⟨*Beispiel*⟩+≡

```
?- parses. der.
+ lam(X, lam(Y, X*Y))
?- parses. wer.
+ lam(X, qu(Y, X*Y))
```

4. Relativierende Nominalphrase mit Possessiv:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([rel(Gen2,Num2),3,Gen],[sg,Kas]),
       [poss([rel],[Gen2,Num2,gen]),Poss],
       [rn([Gen],[sg,Kas]),RN]],
    SemNP) :-
    sem([poss([rel],[Gen2,Num2,gen]),Poss],
        SemPoss),
    sem([rn([Gen],[sg,Kas]),RN],SemRN),
    normalize(SemPoss*SemRN,SemNP).
```

⟨*Beispiel*⟩+≡

```
parses. dessen Planet.
+ lam(A, lam(B, ex(C, planet(C, B))&A*C))
```

5. Maßangabe als Nominalphrase (Sonderfall `det n`):

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([_,3,_],[_,_]),
           [det([def],_),[Zahlatom]],[n(_,_),[km]]],
      lam(P,P*Zahl)) :-
      anzahl(Zahlatom,Zahl). % Ausnahme: Maß
```

Hier werden Zahlen nicht als Zahlquantoren behandelt:

⟨*Beispiel*⟩+≡

```
?- parses. 20 km.
lam(A, A*20)
?- parses. 20 Planeten.
lam(A, anzahl(B, planet(B)&A*B, 20))
```

6. Nominalphrase mit NP-Attribut beim Relationsnomen:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([_,3,_],[_,_]),
          Det,[rn(Art,Form),RN],NPgen], Sem) :-
     sem(Det,SemDet),
     sem([rn(Art,Form),RN],SemRN),
     sem(NPgen,SemNPgen),
     SemN = lam(X,SemNPgen*lam(Y,(SemRN*X)*Y)),
     normalize(SemDet*SemN,Sem).
```

⟨*Beispiel*⟩+≡

```
parses. jeder Mond eines Himmelskörpers.
lam(A, all(B, ex(C, himmelskörper(C) &
                    mond(B, C))      => A*B))
```

Aber: bei `ein Mond jedes Planeten` bräuchte man die umgekehrte Anordnung der Quantoren!

7. Sätze mit Vollverb in `vz`, Vorfeld-NP mit weitem Skopus:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([_Def],[_,_,vz]),
      NP1, [v([nom,akk],[3,Num,_,ind]),V], NP2],
    SemS)
:-  sem(NP1,SemNP1), sem(NP2,SemNP2),
    sem([v([nom,akk],[3,Num,_,ind]),V], SemV),
    NP1 = [np([_,3,_],[Num1,Kas1])|_Konst],
    (Kas1 = nom, Num1 = Num,
        Sem = SemNP1 * lam(X,SemNP2 *
                           lam(Y,SemV * X * Y))
    ; Kas1 = akk,
        Sem = SemNP1 * lam(Y,SemNP2 *
                           lam(X,SemV * X * Y))
    ), normalize(Sem,SemS).
```

⟨*Beispiel*⟩+≡

```
parses. Galilei entdeckte einen Stern.
Baum: s([def], [praet, ind, vz])
      + ex(X, stern(X) & entdecken(galilei, X))
         np([def, 3, mask], [sg, nom])
         + lam(X, X*galilei)
            en([mask], [sg, nom])
            + galilei 'Galilei'
         v([nom, akk], [3, sg, praet, ind])
         + lam(X, lam(Y, entdecken(X,Y))) entdeckte
         np([indef, 3, mask], [sg, akk])
         + lam(X, ex(Y, stern(Y) & X*Y))
            det([indef], [mask, sg, akk])
            + lam(X, lam(Y, ex(Z, X*Z & Y*Z))) einen
            n([mask], [sg, akk])
            + lam(X, stern(X)) 'Stern'
```

- Die Bedeutung errechnet sich durch Normalisieren von
  `Sem = SemNP1 * lam(X,SemNP2 * lam(Y,SemV * X * Y))`
  Davon rechnen wir einen Teil nach:

⟨*Normalisierung von* `SemNP2 * lam(Y,SemV * X * Y)`⟩≡

```
  SemNP2 = lam(X, ex(Y, stern(Y) & X*Y))
  SemV = lam(X, lam(Y, entdecken(X, Y)))


  SemNP2 * lam(Y,SemV*X*Y)
   = lam(X,ex(Y,stern(Y) & X*Y)) * lam(Y,SemV*X*Y)
  ->       ex(Y,stern(Y) & X*Y)[X/lam(Y,SemV*X*Y)]
   = ex(Y1,stern(Y1) & lam(Y,SemV*X*Y)*Y1)
  -> ex(Y1,stern(Y1) & (SemV*X*Y)[Y/Y1])
   = ex(Y1,stern(Y1) & lam(X2,lam(Y2,entdecken(X2,Y2)))*X*Y1
  -> ex(Y1,stern(Y1) & lam(Y2,entdecken(X2,Y2))[X2/X]*Y1)
   = ex(Y1,stern(Y1) & lam(Y3,entdecken(X,Y3))*Y1)
  -> ex(Y1,stern(Y1) & entdecken(X,Y3)[Y3/Y1])
   = ex(Y1,stern(Y1) & entdecken(X,Y1))
```

- Das Programm nennt bei $\lambda x\, t \cdot s \rightarrow_\beta t[x/s]$ nicht nur die durch λ, sondern auch die durch $\exists$,$\forall$ gebundenen Variablen in t um.

$\langle$*Beispiel einer fehlenden Umbennenung*$\rangle\equiv$

```
?- SemNP = lam(X, ex(Y, stern(Y) & X*Y)),
   SemV = lam(Z, ex(Y, entdecken(Y, Z))),
   normalize(SemNP * SemV,Nf).

SemNP = lam(_G152, ex(_G150, stern(_G150)&_G152*_G150))
X = _G152
Y = _G150
SemV = lam(_G168, ex(_G150, entdecken(_G150, _G168)))
Z = _G168
Nf = ex(_G381, stern(_G381)&ex(_G484,
                                    entdecken(_G484, _G381)))
```

- Die NP im Vorfeld erhält weiten Wirkungsbereich; die Stellung der NPs wirkt sich also auf die Bedeutung aus:

$\langle$*Beispiel:*$\rangle \equiv$

```
?- parses. jeder Astronom entdeckte einen Stern.
+ all(X,astronom(X)=>ex(Y,stern(Y)
                              & entdecken(X,Y)))
?- parses. einen Stern entdeckte jeder Astronom.
+ ex(X,stern(X) & all(Y,astronom(Y)
                            => entdecken(Y,X)))
```

Das ist wohl die richtige Lesart, wenn das Vorfeld (vor dem Verb) betont ist, also insbesondere bei Fragen:

$\langle$*Beispiel*$\rangle{+}{\equiv}$

```
?- parses.
   welchen Planeten entdeckte jeder Astronom.
+ qu(X,planet(X) & all(Y,astronom(Y)
                       => entdecken(Y,X)))
?- parses.
   welcher Astronom entdeckte einen Planeten.
+ qu(X,astronom(X)&ex(Y,planet(Y)
                      & entdecken(X,Y)))
?- parses. welchen Planeten umkreisen 4 Monde.
+ qu(X,planet(X)&anzahl(Y,mond(Y)
                        & umkreisen(Y,X),4))
```

Bem.: Wie müßte die Regel geändert werden, damit stets die Subjekt-NP den weiten Wirkungsbereich hat?

8. Ja/Nein-Fragen mit Vollverb: hier soll die erste Nominalphrase weiten Wirkungsbereich bekommen:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([qu],[_,ind,ve]),
       [v([nom,akk],[3,Num,_,ind]),V], NP1, NP2],
     SemS)
  :-
     sem([v([nom,akk],[3,Num,_,ind]),V], SemV),
     sem(NP1,SemNP1), sem(NP2,SemNP2),
     NP1 = [np(_,[Num1,Kas1])|_Konstituenten],
     (Kas1 = nom, Num1 = Num,
         Sem = SemNP1 * lam(X,SemNP2 *
                              lam(Y,SemV * X * Y))
     ; Kas1 = akk,
         Sem = SemNP1 * lam(Y,SemNP2 *
                              lam(X,SemV * X * Y))
     ), normalize(Sem,SemS).
```

$\langle$*Beispiel*$\rangle$+$\equiv$

```
?- parses. umkreist jeder Mond einen Planeten.
+ all(X,mond(X)=>ex(Y,planet(Y) & umkreisen(X,Y)))

?- parses. umkreist ein Mond alle Planeten.
+ ex(X, mond(X)&all(Y, planet(Y)=>umkreisen(X, Y)))

?- parses. umkreist alle Planeten ein Mond.
+ all(X, planet(X)=>ex(Y, mond(Y)&umkreisen(Y, X)))
```

Die Stellung der Komplemente beeinflußt also wieder die Bedeutung, da nur eine der beiden Skopusverhältnisse implementiert wurde.

9. Ja/Nein-Fragen mit prädikativer Nominalphrase: diese soll am Ende stehen:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([qu],[_,ind,ve]),
      [v([sein],[3,Num,_,ind]),_V], NP1, NP2],
    SemS)
:-
    NP1 = [np(_,[Num,nom])|_Konst1],
    NP2 = [np([Def2,3,_],[Num,nom])|_Konst2],
    sem(NP1,SemNP1),
    sem(NP2,SemNP2), (Def2 = indef ; Def2 = def),
    Sem = SemNP1*lam(X,SemNP2*lam(Y,eq(X,Y))),
    normalize(Sem,SemS).
```

$\langle$*Beispiel*$\rangle+\equiv$

```
?- parses. ist Uranus ein Planet.
+ ex(X, planet(X) & eq(uranus, X))

?- parses. ist ein Mond ein Planet.
    + ex(X, mond(X)&ex(Y, planet(Y)&eq(X, Y)))
```

Im zweiten Fall läge die generische Lesart des unbestimmten Artikels,

```
all(X, mond(X) => planet(X)),
```

näher.

10. Ja/Nein-Fragen mit Vergleich:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([qu],[_,ind,ve]),
        [v([sein],[3,Num,_,ind]),_], NP1, AP],
      SemS)
  :-
      NP1 = [np(_,[Num,nom])|_Konst1],
      AP  = [ap([def],[komp])|_Konst2],
      sem(NP1,SemNP1), sem(AP,SemAP),
      normalize(SemNP1*SemAP, SemS).
```

⟨*Beispiele* ⟩+≡

```
?- parses. sind 250 km kleiner als 300 km.
250<300
?- parses. ist der Durchmesser des Uranus
           kleiner als 3000 km.
+ ex(X, all(Y, eq(Y, X) <=> durchmesser(Y,uranus))
        & (X<3000))
```

11. Relativsätze: (mit Vollverb)

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([rel(GenR,NumR)],[Tmp,Mod,vl]),
       NP1,
       NP2,
       [v([nom,akk],[3,Num,Tmp,Mod]),V]],
    SemS)
:-
    NP1 = [np([rel(GenR,NumR),3,GenR],[_,Kas1])|_],
    sem(NP1,SemNP1),
    sem([v([nom,akk],[3,Num,Tmp,Mod]),V], SemV),
    sem(NP2,SemNP2),
    ( Kas1 = nom, Sem =
      SemNP1 * lam(X,SemNP2 * lam(Y,SemV * X * Y))
    ; Kas1 = akk, Sem =
      SemNP1 * lam(Y,SemNP2 * lam(X,SemV * X * Y))
    ),
    normalize(Sem,SemS).
```

$\langle$*Beispiel*$\rangle+\equiv$

```
?- parses. der den Uranus umkreist.
+ lam(X, umkreisen(X, uranus))
?- parses. den der Uranus umkreist.
+ lam(X, umkreisen(uranus, X))
?- parses. der jeden Stern umkreist.
+ lam(X, all(Y, stern(Y)=>umkreisen(X, Y)))
```

12. Relativsätze mit AP-Prädikativ:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([rel(GenR,NumR)],[Temp,ind,vl]),
      NP1, AP, [v([sein],[3,Num,Temp,ind]),_]],
    SemS)
:-
    NP1 = [np([rel(GenR,NumR),3,_],[Num,nom])|_],
    AP  = [ap([def],[komp])|_Konst2],
    sem(NP1,SemNP1),
    sem(AP,SemAP),
    normalize(SemNP1*SemAP, SemS).
```

⟨*Beispiele*⟩+≡

```
?- parses. der kleiner als 20 km ist.
+ lam(X, X<20)

?- parses. deren Durchmesser kleiner als 200 km ist.
+ lam(X, ex(Y, durchmesser(Y, X)& (Y<200)))

?- parses. ein Stern, dessen Durchmesser kleiner
           als 50000 km ist.
+ lam(X, ex(Y, stern(Y)&ex(Z, durchmesser(Z, Y)
                                    & (Z<50000))
                 & X*Y))
```

13. Relativsätze mit NP-Prädikativ:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([rel(GenR,NumR)],[Temp,ind,vl]),
        NP1, NP2, [v([sein],[3,Num,Temp,ind]),_V]],
     SemS)
:-
     NP1 = [np([rel(GenR,NumR),3,_],[Num,nom])|_],
     NP2 = [np([Def2,3,_],[Num,nom])|_Konst2],
     sem(NP1,SemNP1),
     sem(NP2,SemNP2), (Def2 = indef ; Def2 =def),
     Sem = SemNP1*lam(X,SemNP2*lam(Y,eq(X,Y))),
     normalize(Sem,SemS).
```

⟨*Beispiel*⟩+≡

```
?- parses. der ein Planet ist.
+ lam(X, ex(Y, planet(Y) & eq(X,Y)))

?- parses. deren Planet die Venus ist.
+ lam(X, ex(Y, planet(Y,X) & eq(Y,venus)))

?- parses. die Planeten sind.   % scheitert!
```

Man braucht hier die Bedeutung des (Relations-)Nomens im Plural als Nominalphrase.

14. Nominalphrasen mit Relativsatz: (bisher: `n`, kein `rn`)

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([_Def,3,Gen],[Num,Kas]),
        DET,
        [n([Gen],[Num,Kas]), N],
        [s([rel(Gen,Num)],[_,_,vl])|Konst]],
    LamTerm)
  :-
    sem(DET,SemDET),
    sem([n([Gen],[Num,Kas]),N],SemN),
    sem([s([rel(Gen,Num)],[_,_,vl])|Konst],SemSREL),
    SemNREL = lam(X,(SemN * X) & (SemSREL * X)),
    normalize(SemDET * SemNREL, LamTerm).
```

⟨*Beispiel*⟩+≡

```
?- parses. ein Astronom, der den Uranus entdeckte.
+ lam(X, ex(Y, astronom(Y)
               & entdecken(Y,uranus)
               & X*Y))
?- parses. jeder Stern, den ein Astronom entdeckte.
+ lam(X, all(Y, stern(Y)
                & ex(Z, astronom(Z)
                        & entdecken(Z,Y))
                => X*Y))
```

15. Für Eigennamen mit Relativsatz fehlen `sem`-Klauseln.

15. Adjektivphrasen: die Vergleichs-NP im 2.Argument der Relation

⟨*Semantik/sem.pl*⟩+≡

```
sem([ap([def],[komp]),
         [a([als(nom)],[komp]),Adj], [als], NP],
     SemAP)
:-   sem([a([als(nom)],[komp]),Adj], SemAdj),
     NP = [np([_,3,_],[_,nom])|_],
     sem(NP,SemNP),
     normalize(lam(X,SemNP*lam(Y,SemAdj*X*Y)), SemAP).
```

⟨*Beispiel*⟩+≡

```
?- parses. größer als 3000 km.
+ lam(X, X>3000)

?- parses. kleiner als der Durchmesser des Uranus.
+ lam(X, ex(Y, all(Z, eq(Z, Y) <=>
                      durchmesser(Z, uranus))
                 & (X<Y)))
```

Fehlende Fälle in der Semantik

Relativsätze an (mit Artikel, Nomen erweiterten) Eigennamen:

$\langle$*fehlt.semantik*$\rangle\equiv$

```
der Planet Venus, der ein Planet ist.
die Venus, die ein Planet ist.
der Venus, die ein Planet ist.
```

NP-Prädikativsätze in Verbzweitstellung:

$\langle$*fehlt.semantik*$\rangle{+}\equiv$

```
der Uranus ist ein Planet. % unbehandelt
```

NP-Prädikative mit indefiniter NPpl (ohne Artikel):

$\langle$*fehlt.semantik*$\rangle{+}\equiv$

```
die Astronomen sind
dessen Planeten Sterne sind.
```

Verbzweitsätze mit AP-Prädikativen:

$\langle$*fehlt.semantik*$\rangle$ $+\equiv$

```
der Durchmesser der Venus ist kleiner als der
 Durchmesser des Uranus.
welcher Mond ist kleiner als der Durchmesser
 des Uranus.
```

Prädikativsätze mit Plural:

$\langle$*fehlt.semantik*$\rangle$ $+\equiv$

```
sind die Planeten die Monde der Venus.
[Sterne, ] die Planeten sind.
```

Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing)
- Semantik:
  1. Datenbank (Objekte und Grundrelationen)
  2. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  3. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache

Auswertung von Fragen

Bei einer Ja/Nein-Frage (Verberststellung) muß ein Wahrheitswert berechnet werden, bei einer W-Frage die Liste der Objekte mit der jeweiligen Bedingung:

$\langle$*semantik.pl*$\rangle+\equiv$

```
frage :-
    write('Beende die Frage mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    Startsymbol = s([qu],[_Temp,ind,Vst]),
    addTree:translate1(Startsymbol,Term,Baum),
    addDifflists:translate(Term,ExpTerm,Atomlist,[]),
    nl,write('Aufruf: '), portray_clause(ExpTerm),
```

⟨*semantik.pl*⟩+≡

```
      (call(ExpTerm) ->
           (sem(Baum,LamTerm) ->
                (Vst = ve -> beantworte(LamTerm,Wert)
                ; antworten(LamTerm,Wert)),
                nl,write('Antwort: '),write(Wert)
           ; write('\nDie Frage kann leider noch nicht \
                   in eine Formel umgewandelt werden.'),
                nl,nl,write('Baum:    '), writeq(Baum)
           )
      ;
           nl, write('\nDie Eingabe wurde \
                     syntaktisch nicht erkannt.\n')
      ).
```

Aufgabe: Man erzeuge aus `Baum` und `Wert` eine lesbare Antwort!

Beispiele zur Auswertung von Fragen

Fragen werden nicht an Fragezeichen, sondern an der Verbstellung oder den Fragepronomen und -Artikeln erkannt.

⟨*Beispiel einer Ja/Nein-Frage*⟩≡

```
?- frage.
Beende die Frage mit <Punkt><Return>
|: entdeckte Galilei einen Mond des Uranus.

Antwort: nein

?- frage. umkreist Uranus die Sonne.
Antwort: ja
```

W-Fragen nach Subjekt oder Objekt (als erster Konstituente) können mit Frageartikel oder -Pronomen formuliert werden:

⟨*Beispiel für Subjektfragen*⟩≡

```
?- frage. welcher Astronom entdeckte den Uranus.
Antwort: [herschel]

?- frage. wer entdeckte einen Mond des Uranus.
Antwort: [lassell]
```

⟨*Beispiel einer Objektfrage*⟩≡

```
?- frage. welchen Mond entdeckte Galilei.
Antwort: [europa, ganymed, io, kallisto]
```

Beispiele mit Anzahlquantoren

Fragen nach einer Anzahl sind nicht implementiert, aber Anzahlquantoren werden behandelt:

⟨*Beispiel einer Frage mit Zahlquantor*⟩≡

```
?- frage. welche Astronomen entdeckten 2 Monde.
Antwort: [cassini, galilei, herschel, lassell, nicholson]
```

⟨*Beispiel einer Frage mit Relativsatz*⟩≡

```
?- frage. welcher Astronom, der 2 Monde entdeckte,
   entdeckte einen Planeten.
Antwort: [herschel]


?- frage. welcher Mond, den ein Astronom entdeckte,
          umkreist Uranus.
Antwort: [ariel]
```

Prädikativsätze

Hierzu gibt es nur einen kleinen Anfang, z.B.

⟨*Beispiel eines Frage mit Hilfsverb*⟩≡

```
?- frage. ist Uranus ein Planet, den Galilei entdeckte.
Antwort: nein

?- frage. ist Uranus ein Planet, den ein Mond umkreist.
Antwort: ja

?- frage. ist Galilei ein Astronom,
          der 3 Monde eines Planeten entdeckte.
Antwort: ja

?- frage. umkreist jeder Planet,
          dessen Mond den Uranus umkreist, eine Sonne.
Antwort: ja
```

Zahlangaben

In Maßangaben wie $n\ km$ werden Zahlen nicht als Zahlquantoren behandelt (wie in Zahlangaben bei $n\ Monde$):

⟨*Beispiele*⟩+≡

```
?- frage.
|: sind 10 km kleiner als der Durchmesser des Uranus.
Antwort: ja

?- frage.
|: ist der Durchmesser des Uranus kleiner als 15000 km.
Antwort: nein

?- frage.
|: ist der Durchmesser des Uranus kleiner als 55000 km.
Antwort: ja
```

Unbehandelt: Kontextabhängige Bedeutung

Die Bedeutung eines Ausdrucks kann vom Kontext abhängen:

1. Aus einem transitiven Verb `TV` mit (Individuen-) Objekt `c` läßt sich ein *einstelliges* Prädikat `VP1` mit der Semantik

   ```
   SemVP1 = lam(X,SemTV*X*c)
   ```

   bilden, z.B. `VP1 = die Sonne umkreisen`.

   Man kann daraus aber auch ein (symmetrisches) *zweistelliges* Prädikat `VP2` mit der Semantik

   ```
   SemVP2 = lam(X,lam(Y,SemTV*X*Y & SemTV*Y*X))
   ```

   machen, wie `VP2 = einander umkreisen`.

   Die Bedeutung einer `NPpl` hängt davon ab, ob sie in einem Satz `NPpl VP1` oder in `NPpl VP2` vorkommt:

   ```
   SemNPpl = lam(P1,...,P1*X, ,...)
           | lam(P2,...,P2*X*Y,...)
   ```

Die Wahl zwischen diesen Bedeutungen hängt vom *Kontext* `..VP1` oder `..VP2` ab! Z.B.

```
einige N = lam(P,ex(X,N*X & P*X))
         | lam(P,ex(X,N*X & ex(Y,N*Y & neq(X,Y)
                                    & P*X*Y)))
```

in `einige Sterne VP1` versus `einige Sterne VP2`.

Die `sem`-Regel für `S -> NPpl VP` muß also `VP` zur Berechnung von `SemNPpl` ausnutzen.

2. Die Bedeutung eines possessiven Relativpronomens hängt vom Numerus des folgenden Relationsnomens ab:

```
deren Mond = für (den) einen ihrer Monde
deren Monde = für alle ihre Monde
```

Für den zweiten Fall brauchen wir eine neue Regel:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([rel(Gen2,Num2),3,Gen],[pl,Kas]),
          [poss([rel],[Gen2,Num2,gen]),_Poss],
          [rn([Gen],[pl,Kas]),RN]],
      SemNP) :-
      SemPoss = lam(R,lam(P,lam(Y,
                      all(X,R*X*Y => P*X)))),
      sem([rn([Gen],[pl,Kas]),RN],SemRN),
      normalize(SemPoss*SemRN,SemNP).
```

Hier hängt `SemPoss` vom Numerus `pl` des Relationsnomen ab; die Bedeutung im Lexikon gilt für `sg`.

⟨*Beispiel*⟩+≡

```
?- parses. dessen Monde.
+ lam(X, lam(Y, all(Z, mond(Z,Y) => X*Z)))
?- parses. dessen Mond.
+ lam(X, lam(Y, ex(Z, mond(Z,Y) & X*Z)))
```

Unbehandelt: Zusammengesetzte Sätze

Sätze, die mit Junktoren aus Teilsätzen zusammengesetzt werden, haben wir ignoriert, da sie als Fragen selten vorkommen.

`Wenn Ariel ein Mond ist, dann umkreist er einen Planeten.`

In zusammengesetzten Sätzen kommen oft Personalpronomen vor, und die bringen weitere Schwierigkeiten mit sich:

1. die syntaktische Analyse muß mögliche Bezugsnominalphrasen kennzeichnen,
2. die Auswertung erfordert einen Kontext bekannter Objekte zur Auswahl der Pronomenbedeutungen

3. die Behandlung der quantifizierten Nominalphrasen muß berücksichtigen, welche Pronomina sich auf sie beziehen:

   `Wenn ein Planet die Sonne umkreist, dann beleuchtet sie ihn.`

   Man kann die Quantoren nicht wie bei einfachen Sätzen behandeln, da sie sich auch auf die Pronomen in anderen Teilsätzen beziehen. Man braucht neue Regeln wie:

   `Wenn (...(Q N)...), dann (...Pronomen ...).`

$$\mapsto \quad Q'x \in N((...\ x\ ...) \to (...\ x\ ...))$$

   Das betrachtet H.Kamp in seiner “Diskursrepräsentationstheorie” (DRT).

Semantische Berücksichtigung des Numerus

Man muß i.a. den Numerus in der Semantik berücksichtigen:

Angemessene Behandlung des bestimmten Artikels

Der bestimmte Artikel im Plural wird semantisch gar nicht behandelt, der im Singular als Eindeutigkeitsbehauptung.

Man sollte einen Kontext genannter Objekte verwalten und die Bedeutung der bestimmten Artikel durch die Suche nach den zuletzt in diesen Kontext eingeführten Objekten interpretieren.

Überblick

- Lexikographie: Vollformen- vs. Stammformenlexikon
- Tokenisierung und lexikalische Analyse eines Texts
- Syntax: Grammatikregeln, Syntaxanalyse (Parsing)
- Semantik:
  1. Datenbank (Objekte und Grundrelationen)
  2. λ-Terme/Formeln als Darstellung der Bedeutung von Ausdrücken/Sätzen
  3. Übersetzung von Syntaxbäumen in λ-Terme/Formeln
- Auswertung: Datenbankabfrage mit logischen Formeln
- Generierung: Fragebeantwortung in natürlicher Sprache
- Typisierung: Typkorrektheit der λ-Terme, schnellere Suche

Generierung natürlicher Sprache

Als Antwort auf eine Frage wird durch eine Datenbankabfrage ein *Wert* ermittelt, ein Wahrheitswert oder eine Namensliste.

Wie erzeugen wir eine Antwort in natürlicher Sprache? So:

1. Errechne den Syntaxbaum der Frage und den Wert,
2. konstruiere daraus einen Syntaxbaum der Antwort,
3. gib dessen Blattfolge als Antwortsatz aus.

Wie entsteht der Syntaxbaum der Antwort aus dem der Frage?

- bei Ja/Nein-Fragen durch Umstellung des Verbs,
- bei W-Frage durch Ersetzen der interrogativen Konstituente durch Syntaxbäume der Antwortwerte.

Zusätzlich sind leichte Transformationen der Bäume nötig, z.B. Einfügung von Floskeln, Konjunktionen, Numerusänderungen.

Bem.: Für Doppelfragen reichen diese Umformungen nicht aus.

Frage mit Antwort in natürlicher Sprache

Um eine Antwort in natürlicher Sprache zu bekommen, muß man die Frage „nett“ stellen, d.h. mit `fragen/0` statt mit `frage/0`:

⟨*Generierung/antworten.pl*⟩≡

```
fragen :-
    write('Beende die Frage mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    Startsymbol = s([qu],[_Temp,ind,Vst]),
    addTree:translate1(Startsymbol,Term,Baum),
    addDifflists:translate(Term,ExpTerm,Atomlist,[]),
    nl,write('Aufruf: '), portray_clause(ExpTerm),
```

⟨*Generierung/antworten.pl*⟩+≡

```
    (call(ExpTerm) ->
        (sem(Baum,LamTerm) ->
            (Vst = ve ->
                beantworte(LamTerm,Wert)
            ; antworten(LamTerm,Wert)),
            (antwortbaum(Baum,Wert,Antwort)
            -> blaetter(Antwort,Blaetter),
                concat_atom(Blaetter,' ',Atom),
                nl,writef('Antwort: %w.',[Atom])
            ; nl,writef('Antwort: %w',[Wert]),
                write('\nDie Antwort kann noch nicht \
                     in natürlicher Sprache \
                     formuliert werden.\n')
            )
```

⟨*Generierung/antworten.pl*⟩+≡

```
        ; write('\nDie Frage kann leider noch nicht \
                    in eine Formel umgewandelt werden.'),
              nl,nl,write('Baum:   '), writeq(Baum)
        )
    ;
        nl, write('\nDie Eingabe wurde \
                      syntaktisch nicht erkannt.\n')
    ).
```

Blattfolge eines Baums

Die Folge der Blätter eines Syntaxbaums wird mit einer Hilfsliste `Korb` gesammelt, während man den Baum durchläuft:

$\langle$*Generierung/antworten.pl*$\rangle{+}\equiv$

```
% blaetter(+Baum,-Liste der Blaetter).

blaetter([],[]).
blaetter([Wurzel|Teilbaeume],Blaetter) :-
    blaetter([Wurzel|Teilbaeume],[],
             GespiegelteBlaetter),
    reverse(GespiegelteBlaetter,Blaetter).

blaetter([Wurzel],Korb,Blaetter) :-
    % [Wurzel] besteht aus einem Blatt (Atom)
     Blaetter = [Wurzel|Korb].
```

⟨*Generierung/antworten.pl*⟩+≡

```
blaetter([_Wurzel,Baum],Korb,Blaetter) :-
    blaetter(Baum,Korb,Blaetter).
blaetter([Wurzel,Baum,Baum2|Baueme],Korb,Blaetter) :-
    blaetter(Baum,Korb,GrosserKorb),
    blaetter([Wurzel,Baum2|Baueme],
             GrosserKorb,Blaetter).
```

Beantwortung von Ja/Nein-Fragen

Die Antwort auf eine Ja/Nein-Frage formulieren wir als

⟨*Antworten bei Wert* `ja`⟩≡

```
Ja, <Verbzweit-Satz>.
```

⟨*Antwort bei Wert* `nein`⟩≡

```
Nein, es ist nicht der Fall, daß <Verbletzt-Satz>.
```

Der Syntaxbaum der Antwort ist ein Aussagesatz, der aus dem Adverbial der einleitenden Floskel und dem in die passende Verbstellung transformierten Syntaxbaum der Frage besteht:

⟨*Generierung/antworten.pl*⟩+≡

```
% ----- Beantwortung von Ja/Nein-Fragen: ------
antwortbaum(Frage,ja,Antwort) :-
    !, Frage = [s([qu],[Temp,Mod,ve])|_],
    ve>>vz(Frage,Antwort1),
    Antwort = [s([def],[Temp,Mod,vz]),
                [adv,['Ja,']],Antwort1].

antwortbaum(Frage,nein,Antwort) :-
    !, Frage = [s([qu],[Temp,Mod,ve]) |_],
    ve>>vl(Frage,Antwort1),
    Antwort = [s([def],[Temp,Mod,vz]),
                [adv,['Nein, es ist nicht \
                        der Fall, daß']],
               Antwort1].
```

Die Antwortbäume sind keine Analysebäume im Sinne der Grammatik; sie erleichtern nur die Formulierung der Antwort.

Änderung der Verbstellung

Für einfache Sätze ist die Umstellung gleich, egal, ob das Prädikat aus einem transitiven Vollverb oder einem Hilfsverb besteht:

⟨*Generierung/antworten.pl*⟩+≡

```
ve>>vz([s([qu],[Temp,Mod,ve]),TV,Obj1,Obj2],
       [s([def],[Temp,Mod,vz]),Obj1,TV,Obj2]) :-
    ( TV = [v([nom,akk],_),_V]
    ; TV = [v([sein],[3,sg,Temp,Mod]),_V]).

ve>>vl([s([qu],[Temp,Mod,ve]),TV,Obj1,Obj2],
       [s([def],[Temp,Mod,vl]),Obj1,Obj2,TV]) :-
    ( TV = [v([nom,akk],_),_V]
    ; TV = [v([sein], [3,sg,Temp,Mod]),_V]).
```

⟨*Beispiel*⟩+≡

```
?- [semantik,astronomie,'Generierung/antworten'].
?- fragen. entdeckte Herschel einen Planeten.
Antwort: Ja, Herschel entdeckte einen Planeten.

?- fragen. umkreisen den Jupiter 4 Monde.
Antwort: Ja, den Jupiter umkreisen 4 Monde.

?- fragen. ist der Uranus ein Planet.
Antwort: Ja, der Uranus ist ein Planet.

?- fragen. ist der Uranus ein Mond.
Antwort: Nein, es ist nicht der Fall, daß
               der Uranus ein Mond ist.
```

Beantwortung von Konstituentenfragen

Die Formulierung einer Antwort auf W-Fragen ist schwieriger.

Es wurden nur W-Fragen mit einer interrogativen Nominalphrase als Konstituente am Satzanfang erlaubt. Der Antwortwert ist eine Namensliste.

$\langle$*Antwort bei Wert* `[Name1,Name2,..]`$\rangle\equiv$

```
  <Frage>[ [np([qu,3,_],Form)|_] /
                  [np([def,3,Num],_) ...Name1..] ]
```

Der Syntaxbaum der Antwort entsteht aus dem der Frage, indem man die interrogative Nominalphrase durch eine definite ersetzt, die aus den Namen der Liste gebildet wird.

⟨*Generierung/antworten.pl*⟩+≡

```
% --- Beantwortung von W-Fragen (W im Vorfeld) ---

antwortbaum(Frage,Namen,Antwort) :-
     Frage = [s([qu],[Temp,Mod,vz]),
                [np([qu,3,Gen],[Num,Kas])|_] | Rest],
     antwortbaum([np([qu,3,Gen],[Num,Kas])|_],
                 Namen,NP),
     Antwort = [s([def],[Temp,Mod,vz]),NP | Rest].
```

Ergänze: Ist die `NP` im Nominativ, so sollte je nach |`Namen`| der Numerus des Verbs Plural sein! Ist sie ein Eigenname im Akkusativ, so sollte das durch einen Artikel klargemacht oder das Subjekt ins Vorfeld gestellt werden.

Konstruktion der Antwort-Nominalphrase

Die Antwort-Nominalphrase sollte von der Form der interrogativen Nominalphrase abhängen; pro Eigenname der Wertliste:

$\langle$*Beispiel*$\rangle{+}{\equiv}$

```
wer          |--> Eigenname,
welcher N    |--> der N Eigenname
wessen RN    |--> ein/der RN des NP
wieviele km  |--> n km
```

Fall 1: die interrogative NP ist ein Pronomen, wie `wer`. Aus der Wertliste wird i.a. eine Konjunktion von Eigennamen gebildet:

Zwischen die Namen werden Kommata eingefügt, sofern es noch mindestens drei sind:

⟨*Generierung/antworten.pl*⟩+≡

```
antwortbaum([np([qu,3,Gen],[sg,Kas]),
             [pron([qu],_)|_]],[N1,N2,N3|Ns],Baum) :-
    !, antwortbaum([np([qu,3,_Gen1],[sg,Kas]),
                   [pron([qu],_)|_]],[N1],Baum1),
    antwortbaum([np([qu,3,Gen],[sg,Kas]),
                [pron([qu],_)|_]],[N2,N3|Ns],Baum2),
    Baum = [np([def,3,Gen],[pl,Kas]), % Numerus!
            Baum1,[conj,[',']],Baum2].% Komma
```

Zwischen die letzten beiden Namen wird `und` eingefügt:

⟨*Generierung/antworten.pl*⟩+≡

```
antwortbaum([np([qu,3,Gen],[sg,Kas]),
            [pron([qu],_)|_]],[N1,N2],NpBaum) :-
    !, antwortbaum([np([qu,3,_Gen1],[sg,Kas]),
                   [pron([qu],_)|_]],[N1],NP1),
    antwortbaum([np([qu,3,_Gen2],[sg,Kas]),
                [pron([qu],_)|_]],[N2],NP2),
    NpBaum = [np([def,3,Gen],[pl,Kas]),
             NP1,[conj,[und]],NP2].    % Konjunktion
```

Ist der Wert ein einzelner Name (in Kleinschreibung), so erhalten wir den Syntaxbaum der Antwort, indem wir den Namen (in Großschreibung) syntaktisch analysieren:

⟨*Generierung/antworten.pl*⟩+≡

```
antwortbaum([np([qu,3,_Gen],[sg,Kas]),
             [pron([qu],_)|_]],[Name],Baum) :-
    großschreibung(Name,GrossName),
    np([def,3,_Gen1],[sg,Kas],
       Baum,[GrossName],[]). % Parser aufrufen!

großschreibung(Wort,Gross) :-
    atom(Wort),
    name(Wort,[C|Chars]),
    (member(C,"abcdefghijklmnopqrstuvwxyzäöü")
    ->  K is C-32,
        name(Gross,[K|Chars])
    ; Gross = Wort).
```

Der Numerus beim Verb ist noch anzupassen, und bei Eigennamen im Akkusativ sollte man einen Artikel mit ausgeben:

⟨*Beispiele* ⟩+≡

```
?- fragen. wer entdeckte 3 Monde.
Antwort: Galilei und Herschel entdeckte 3 Monde.

?- fragen. wen umkreist Triton.
Antwort: Neptun umkreist Triton.     % zweideutig

?- fragen. wen entdeckte Galilei.
Antwort: Europa , Ganymed , Io und Kallisto entdeckte Galilei
```

Ist die Antwortliste leer, so bilden wir die Antwort-Nominalphrase aus dem Determinator `kein`. Etwa wie folgt:

⟨*Generierung/antworten.pl*⟩+≡

```
antwortbaum([np([qu,3,Gen],[sg,Kas]),
             [pron([qu],_)|_]],[],Baum) :-
    (Kas = nom,
        (Gen = mask, Kein = 'Keiner'
        ; Gen = fem, Kein = 'Keine')
    ; Kas = akk,
        (Gen = mask, Kein = 'Keinen'
        ; Gen = fem, Kein = 'Keine')
    ),
    Baum = [np([def,3,Gen],[sg,Kas]),
           [det([nindef],[Gen,sg,Kas]),[Kein]]].
```

Bem. Negierte indefinite Determinatoren kommen hier ad-hoc vor, nicht im Lexikon oder den Grammatikregeln.

⟨*Beispiel* ⟩+≡

```
?- fragen. wer entdeckte die Sonne.
Antwort: Keiner entdeckte die Sonne.

?- fragen. wen umkreist die Sonne.
Antwort: Keinen umkreist die Sonne.
```

Fall 2: die interrogative NP ist ein interrogativer Artikel beim Nomen, wie `welcher N`:

Hier sollte das Nomen `N` in die Antwort aufgenommen werden, aber eine Kurzform geht auch jetzt schon:

⟨*Beispiele* ⟩+≡

```
?- fragen. welcher Mond umkreist den Neptun.
Antwort: Triton umkreist den Neptun.

?- fragen. welcher Astronom entdeckte 4 Monde.
Antwort: Galilei entdeckte 4 Monde.

?- fragen. welche Monde entdeckte Galilei.
Antwort: Europa , Ganymed , Io und Kallisto
         entdeckte Galilei.
```

Antworten lassen sich oft nur deshalb nicht formulieren, weil manche Eigennamen der Sterne im Lexikon fehlen.

⟨*Beispiel*⟩+≡

```
?- fragen. welcher Mond umkreist den Uranus.
Antwort: [ariel]
Die Antwort kann noch nicht in natürlicher
Sprache formuliert werden.
```

Außerdem spielt der Numerus der Frage eine Rolle:

⟨*Beispiel*⟩+≡

```
?- fragen. welcher Astronom entdeckte einen Planeten.
Antwort: Herschel und Tombaugh entdeckte einen Planeten.
```

Aufgabe: Man korrigiere das und führe die übrigen Fälle aus.

Es reicht nicht, wenn man die Frage mit Plural-NP auf die mit dem Singular zurückführt:

⟨*Generierung/antworten.pl*⟩+≡

```
  antwortbaum([np([qu,3,Gen],[pl,Kas]),
               [pron([qu],_)|_]],Namen,Baum) :-
      antwortbaum([np([qu,3,Gen],[sg,Kas]),
                   [pron([qu],_)|_]],Namen,Baum).
```

Es sollte aber nicht schwer sein, das zu korrigieren.

Klausurtermin

Tag Mittwoch, 28. Januar 2009

Ort Oettingenstr. 67

Raum 1.14 , **Zeit** 16-18 Uhr,

Selbstverständlich

- *ohne* die Kursmaterialien, aber
- *mit* Schreibstift (und Ausweis).