Computerlinguistik II

CIS, WS 2004/2005

Hans Leiß

7. Februar 2005

Prolog: Terme

Terme in Prolog sind

⟨*Term* ⟩≡
```
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:

⟨*Variable von Prolog*⟩≡
```
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:

⟨*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.Ziffernfolgen:

⟨*Konstante von Prolog*⟩≡
```
Const := [0-9][0-9]*
```

Terme der Form `Atom(Term,...,Term)` sind *zusammengesetzte Terme*.

Prolog: Programme, Regeln, Ziele

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

⟨*Programm* ⟩≡
```
Regel1
Regel2
...
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
```

Ziele können einfach oder zusammengesetzt sein:

⟨*Ziele* ⟩≡
```
Ziel := Term                % einfaches Ziel
    | true                  % erfuellbares 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:

⟨*Beweissuche* ⟩≡

```
?- Ziel.
```

Falls das Ziel ein Term ≠`true` ist, wird es 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`.

Das Ziel `true` gilt als bearbeitet (gelöst, bewiesen).

Falls das Ziel zusammengesetzt ist, wird es 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 1

⟨*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`.

Äquivalente Schreibweisen

Äquivalente Programme sind:

⟨*Alternative im Rumpf*⟩≡
```
Kopf :- Ziel1; Ziel2.
```

⟨*Alternative Regeln mit gleichem Kopf*⟩≡
```
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

⟨*Dateien laden*⟩≡

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

Eine andere Schreibweise ist:

⟨*Dateien laden*⟩+≡

```
?- ['erste_Datei.pl', 'zweite_Datei.pl'].
```

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:

⟨*In der Hauptdatei*⟩≡

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

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

⟨*In der Hauptdatei*⟩+≡

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

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

Nach dem Laden ist das exportierte 2-stellige Prädikat mit *kurzem Namen* `p/2` bekannt; die Hilfsprädikate `q/3` und `p/1` sind nur mit *langem Namen* `paket:q/3,paket:p/1` bekannt:

⟨*Kurze und lange Prädikatnamen*⟩≡
```
?- p(X,Y), paket:q(a,Y,Z).
```

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

Die Importliste ist ein Teil der Exportliste. Der Moduldateiname ist i.a. *nicht* der Modulname. (Im Unterschied zu `ensure_loaded` wird die Moduldatei in jedem Fall geladen, aber mehrfaches Laden verdoppelt die geladenen Regeln nicht.)

Hinweis: Ein großes Programm sollte man in Module zerlegen.

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

% Fehlt: Einschraenkung bei es-Subjekt, Imperativ im sg und 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
```

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

⟨*Vollformenlexikon, Schema*⟩≡

```
wort(?Stammform,?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:

⟨*Morphologie/vollformenlexikon.pl*⟩≡

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

% Schwach konjugiertes Verb:

wort(entdecken,[1,sg,praes,ind],entdecke ).
wort(entdecken,[2,sg,praes,ind],entdeckst).
wort(entdecken,[3,sg,praes,ind],entdeckt ).
wort(entdecken,[1,pl,praes,ind],entdecken).
wort(entdecken,[2,pl,praes,ind],entdeckt ).
wort(entdecken,[3,pl,praes,ind],entdecken).

wort(entdecken,[1,sg,praet,ind],entdeckte  ).
wort(entdecken,[2,sg,praet,ind],entdecktest).
wort(entdecken,[3,sg,praet,ind],entdeckte  ).
wort(entdecken,[1,pl,praet,ind],entdeckten ).
wort(entdecken,[2,pl,praet,ind],entdecktet ).
wort(entdecken,[3,pl,praet,ind],entdeckten ).
```

```
% Stark konjugiertes Verb:

wort(gehen,[1,sg,praes,ind],gehe ).
wort(gehen,[2,sg,praes,ind],gehst).
wort(gehen,[3,sg,praes,ind],geht ).
wort(gehen,[1,pl,praes,ind],gehen).
wort(gehen,[2,pl,praes,ind],geht ).
wort(gehen,[3,pl,praes,ind],gehen).

wort(gehen,[1,sg,praet,ind],ging  ).
wort(gehen,[2,sg,praet,ind],gingst).
wort(gehen,[3,sg,praet,ind],ging  ).
wort(gehen,[1,pl,praet,ind],gingen).
wort(gehen,[2,pl,praet,ind],gingt ).
wort(gehen,[3,pl,praet,ind],gingen).

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

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$_v$lex.pl*⟩≡

```
:- use_module(['wortformen.pl','vollformenlexikon.pl']).

% Verbflexion bei einem Vollformenlexikon:
% verbflexion_vlex(+Stammform,+Formmerkmale,-Vollform)

verbflexion_vlex(Stammform,Formmerkmale,Vollform) :-
    wort(Stammform,Formmerkmale,Vollform).
```

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

⟨*Morphologie/flexion$_v$lex.pl*⟩+≡

```
% Flexionsprogramm: verbflexion_vlex(Stammform+)

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

Unberücksichtigt bleiben hier die bei manchen Verbstellungen im deutschen Satz nötigen *unzusammenhängenden* Wortformen: `hörst du auf?`, `du hörst auf?`, `wenn du aufhörst`.

⟨*Abfragebeispiel* ⟩≡

```
 ?- verbflexion_vlex(gehen,[1,pl,Tempus,ind],Vollform).
```

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,?Flexionsklasse)

lex(ablegen, v,[nom,akk],     rg(2)          ).
lex(aufgeben,v,[nom,akk],     urg(3,(e,a,e),5)).
lex(geben,   v,[nom,dat,akk], urg(0,(e,a,e),2)).
lex(heben,   v,[nom,akk],     urg(0,(e,o,o),2)).
lex(entdecken,v,[nom,akk],    rg(0)          ).
lex(singen,  v,[nom,akk],     urg(0,(i,a,u),2)).
lex(umfahren,v,[nom,akk],     urg(2,(a,u,a),4)).
lex(umfahren,v,[nom,akk],     urg(0,(a,u,a),4)).
```

Die Angabe der Flexionsklasse wird unten erläutert.

⟨*Unregelmäßige Verbflexion:*⟩≡

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

Flexionsprogramm mit Stammformenlexikon

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

- die Wortart und die Flexionsklasse abliest,
- eine für die Wortart erlaubte abstrakte Form sucht,
- aus dem Stamm und der abstrakten Form mit einem Flexionsprogramm die Vollform konstruiert.

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

$\langle$*Morphologie/flexion$_s$lex.pl*$\rangle\equiv$

```
:- module(flexion_slex,[erzeuge_vollformenlexikon/1]).
:- use_module([wortformen]). % (stammformen geladen?!)

flektiere(Stammform,Vollform) :-
    lex(Stammform,Wortart,_,Flexionsklasse),
    form(Wortart,Formmerkmale),
    ( (Wortart = v,
          konjugiere(Flexionsklasse,Stammform,
                     Formmerkmale,Vollform))
    ; (member(Wortart,[n,rn,en]),
          dekliniere(Flexionsklasse,Stammform,
                     Formmerkmale,Vollform))
    ).
```

Für andere Wortarten sind weitere Fälle einzufügen.

Bisher wurden nur finite Verbformen erklärt; mindestens diese sollen erlaubte Verbformen sein:

$\langle$*Morphologie/wortformen.pl*$\rangle+\equiv$

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

Natürlich kann man alle Vollformen am Bildschirm anzeigen lassen. Am besten testet man zuerst, ob die Stammform im Stammlexikon vorkommt:

⟨*Morphologie/flexion$_s$lex.pl*⟩+≡

```
flexion_slex(Stammform) :-
    (lex(Stammform,_,_,_) % Stammform bekannt?
    -> lex(Stammform,Wortart,_,Flexionsklasse),
        ( (Wortart = v,
              verbflexion_slex(Flexionsklasse,
                               Stammform))
        ; (member(Wortart,[n,rn,en]),
              nomenflexion_slex(Flexionsklasse,
                                Stammform))
        )
    ; nl, % schreibe neue Zeile
        write('Das Stammlexikon enthält keinen Eintrag zu: '),
        write(Stammform)
    ).

% verbflexion_slex(+Konjugationsklasse,+Stammform)

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

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 das Tripel der Stammvokale bestimmter Formen, etwa `(i,a,u)` bei *s<u>i</u>ngen,s<u>a</u>ng,ges<u>u</u>ngen*.

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|)` bzw. `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: *br<u>e</u>nnen,br<u>a</u>nnte,gebr<u>a</u>nnt*.

Regelm. Verbflexion (Forts.)

⟨*Morphologie/flexion$_s$lex.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)
    -> konjugation(rge,[Pers,Num,Temp,Mod],Endung)
    ;  konjugation(rg, [Pers,Num,Temp,Mod],Endung)
    ),
    concat(Stamm,Endung,Vollf).

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

Nicht nur bei Dentalstämmen, sondern auch in einigen weiteren Fällen ist die `e`-Einfügung nötig.

Unreglm. Verbflexion

⟨*Morphologie/flexion$_s$lex.pl*⟩+≡

```
konjugiere(urg(_,(V1,V2,V3),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),
             [Pers,Num,Temp,Mod],Vollform).
```

Unregelm.Verbflexion (Forts.)

⟨*Morphologie/flexion$_s$lex.pl*⟩+≡

```
infix(Vor,Mitte,Nach,Kette) :-
    concat(Vor,Mitte,Praefix),
    concat(Praefix,Nach,Kette).

verbform((Vor,Nach),(V1,_V2,_V3),
         [Pers,Num,praes,ind],Vfinit) :-
    not(member([Pers,Num],[[2,sg],[3,sg]])),
    infix(Vor,V1,Nach,Stamm),
    konjugation(urg,[Pers,Num,praes,ind],Endung),
    concat(Stamm,Endung,Vfinit).
verbform((Vor,Nach),(V1,_V2,_V3),
         [Pers,Num,praes,ind],Vfinit) :-
    member([Pers,Num],[[2,sg],[3,sg]]),
    umlaut(V1,U),
    infix(Vor,U,Nach,Stamm),
    konjugation(urg,[Pers,Num,praes,ind],Endung),
    concat(Stamm,Endung,Vfinit).

verbform((Vor,Nach),(_V1,V2,_V3),
         [Pers,Num,praet,ind],Vfinit) :-
    infix(Vor,V2,Nach,Stamm),
    konjugation(urg,[Pers,Num,praet,ind],Endung),
    concat(Stamm,Endung,Vfinit).
verbform((Vor,Nach),(_V1,V2,_V3),
         [Pers,Num,praet,konj],Vfinit) :-
    umlaut(V2,U),
    infix(Vor,U,Nach,Stamm),
    konjugation(urg,[Pers,Num,praet,konj],Endung),
    concat(Stamm,Endung,Vfinit).
```

```
umlaut(a,ä).   umlaut(i,i).
umlaut(o,ö).   umlaut(ie,ie).
umlaut(u,ü).   umlaut(ei,ei).
umlaut(e,i).
```

Ein Umlaut muß bei der 2. und 3.Person (im Konjunktiv) vorgenommenen werden. (Man korrigiere die Vollformbildung, wo es nötig ist!)

Konjugationstabellen

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

⟨*Morphologie/flexion$_s$lex.pl*⟩+≡

```
% -- Konjugationstabelle fuer regelm.Konjugation --

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

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

konjugation(rg,[2,sg,praes,ind], 'st').
konjugation(rg,[2,sg,praes,konj],'est').
konjugation(rg,[2,sg,praet,ind], 'test').
konjugation(rg,[2,sg,praet,konj],'test').

konjugation(rg,[2,pl,praes,ind], 't').
```

```
konjugation(rg,[2,pl,praes,konj],'et').
konjugation(rg,[2,pl,praet,ind], 'tet').
konjugation(rg,[2,pl,praet,konj],'tet').

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

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

% ---- mit e-Erweiterung:

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

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

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

konjugation(rge,[2,pl,praes,ind], 'et').
```

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

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

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

% -- Unregelmaessige Konjugation (Finite Verbformen) --

konjugation(urg,[1,sg,praes,ind],   'e').
konjugation(urg,[1,sg,praes,konj],  'e').
konjugation(urg,[1,sg,praet,ind],   '').
konjugation(urg,[1,sg,praet,konj],  'e').

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

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

```
konjugation(urg,[3,sg,praet,ind],   '').
konjugation(urg,[3,sg,praet,konj],  'e').

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

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

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

Prolog: Schreiben von Termen und Regeln

Einen Prolog-Term schreibt man (mit belegten Variablen) durch

⟨*Schreiben eines Terms*⟩≡
```
?- A=a, write(p(A,'B')).
p(a, B)
?- A=a, writeq(p(A,'B')).
p(a, 'B')
```

Um ein Atom aus konstantem und variablem, vom Programmablauf abhängigen Teil zusammenzusetzen, benutzt man

⟨*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` ist. Mit

⟨*Schreiben auf eine Datei*⟩≡
```
    tell(<Dateiname>),
    <Schreibbefehle>,
    told.
```
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$_s$lex.pl*⟩+≡

```
% --- Erzeuge Einträge in ein Vollformenlexikon: ---

erzeuge_vollformenlexikon(Ausgabedatei) :-
    tell(Ausgabedatei),
    lex(Stammform,Wortart,Art,Flexionsklasse),
    form(Wortart,Form),
    ( (Wortart = v,
          konjugiere(Flexionsklasse,Stammform,
                    Form,Vollform),
          portray_clause(wort(Stammform,v(Art,Form),
                             Vollform))
      )
    ; (member(Wortart,[n,rn,en]),
          dekliniere(Flexionsklasse,Stammform,
                    Form,Vollform),
          Kategorie =.. [Wortart,Art,Form],
          portray_clause(wort(Stammform,Kategorie,
                             Vollform))
      )
    ),
    fail.
erzeuge_vollformenlexikon(Ausgabedatei) :-
    told,
    format(user,'Vollformen erzeugt und
                nach ''~w'' geschrieben', Ausgabedatei),
    nl.
```

Anwendung

⟨*Erzeugung des Vollformenlexikons*⟩≡

```
?- ['Morphologie/flexion_slex'].
%  stammformenlexikon compiled into stammformenlexikon
Warning: Clauses of wortformen:form/2 are not together
         in the source-file
%  wortformen compiled into wortformen
Warning: Singleton variables: [V2, V3]
Warning: ...
% flexion_slex compiled into flexion_slex
Yes

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

Die entstandene Datei enthält die Einträge (inzw. anders!)

⟨*Auzug aus vollformen.testlexikon.pl*⟩≡

```
wort(ablegen, [1, sg, praes, ind], ablege).
...
wort(aufgeben, [1, sg, praes, ind], aufgebe).
wort(aufgeben, [1, sg, praet, ind], aufgab).
wort(aufgeben, [1, sg, praet, konj], aufgäbe).
...
wort(heben, [2, pl, praet, konj], höbet).
wort(heben, [3, sg, praes, ind], hibt).   % Fehler
wort(heben, [3, sg, praes, ind], hibet).  % Fehler
...
wort(umfahren, [3, pl, praet, ind], umfuhren).
wort(umfahren, [3, pl, praet, konj], umführen).
```

Kontextfreie Grammatik

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

$\langle$*kontextfreie Regel*$\rangle\equiv$
```
<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 (Konstituenten-) Ausdrücken der Kategorien `<Kat1>` bis `<KatN>` bestehen kann.

$\langle$*Beispiel einer CFG*$\rangle\equiv$
```
s -> np vp
np -> det n
vp -> v | v np

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

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

$\langle$*Gleichwertige Schreibweisen*$\rangle\equiv$
```
vp -> v | v np

vp -> v
vp -> v np
```

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.
np --> det, n.
vp --> v ; v, np.

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

⟨*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 Eingabe `Atomliste1` von Wörtern mit einem Ausdruck der Kategorie `<Kat>` beginnt und als Rest die Liste `Atomliste2` übrigbleibt.

Ein *Text* ist eine Folge von Wörtern; in Prolog stellt man ihn als eine Liste von Prolog-Atomen dar:

⟨*Beispieltext in Prolog*⟩≡

```
['Der','Programmierer',schrieb,das,'Programm'].
```

Da dieser Text mit einer Nominalphrase beginnt und der Rest des Texts `[schrieb,das,'Programm']` lautet, ist

⟨*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

```
[<Element1>,<Element2>,...]
```

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

```
[<Element1>,<Element2> | 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 aus einer Liste kann man die Anfangselemente holen:

⟨*Entnehmen von Listenelementen*⟩≡

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

⟨*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*⟩≡

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

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

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)
        ).

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

Bei der bei der Übersetzung einer Grammatikregel müssen die gleichen Variablen `I,J` für die Eingabe- und Ausgabeliste zur Übersetzung der Regelseiten benutzt werden:

⟨*Parser/addDifflists.pl*⟩≡

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

Die Seiten werden je nach ihrer Form übersetzt:

⟨*Parser/addDifflists.pl*⟩+≡

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

Durch `!` verhindert man, daß `(A,B)`, ...,`{Prolog}` auch vom letzten Fall (d.h. als atomare Kategorie) behandelt werden.

Prolog: Verändern von Ausdrücken beim Einlesen

Mit dem Prädikat `term_expansion/2` kann man erzwingen, daß beim Einlesen von Grammatikregeln statt der üblichen Übersetzung unser `translate/2` benutzt wird:

⟨*Beispiele/term$_e$xpansion0.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, bei `'Beispiele/programmierer.pl'` sind es:

⟨*Übersetzung von* `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.

⟨*Beispiele/programmierer.baum.pl*⟩≡

```
%% s --> np, vp.
%% vp --> v ; v, np.
%% v --> [steht].
%% v --> [schrieb].

%% 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) }.

%% 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]
```

Automatische Ergänzung einer Baumausgabe

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

Aber da z.B. bei `A --> (B;(C,D))` die Anzahl der Teilbäume von der Anwendung beim Parsen abhängt, kann die Übersetzung nur *Variable* zur Baumausgabe und `{Code}`-Instruktionen zum Aufbau der Bäume in die Ziel-DCG einfügen.

1. Wir setzen voraus, daß die Kategorien der Grammatik keine Merkmale haben (CFG), also Prolog-Atome sind.

   ⟨*Parser/addTreeCFG.pl*⟩≡

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

   translate((L --> R),(Ltr --> (Rtr,{Code}))) :-
      (atom(L) -> translate1(L,Ltr,Baum)
      ; format(user,'addTreeCFG:translate/2 kann nur Atome
              um ein Baumargument erweitern, nicht ~w.\n',L),
          fail),     % wegen =.. [L|_]
      translate2(R,Rtr,Baumliste),
      Code = (Baum =.. [L|Baumliste]).
   ```

2. Eine Kategorie (auch mit Merkmalen) erhält ein Argument zur Baumausgabe:

   ⟨*Parser/addTreeCFG.pl*⟩+≡

   ```
   % translate1(+Kategorie,-Term,?Baum)

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

3. Die Seiten werden je nach ihrer Form übersetzt.

   ⟨*Parser/addTreeCFG.pl*⟩+≡

```
% translate2(+Cat,-CatTr,-Bäume)

translate2((A,B),((Atr,Btr),{Code}),Baeume) :-
    !, translate2(A,Atr,BaeumeA),
       translate2(B,Btr,BaeumeB),
    Code = append(BaeumeA,BaeumeB,Baeume).

translate2((A;B),(Atr;Btr),Baeume) :-
    !, translate2(A,Atr0,BaeumeA),
       translate2(B,Btr0,BaeumeB),
    Atr = (Atr0,{ Baeume=BaeumeA }),
    Btr = (Btr0,{ Baeume=BaeumeB }).

translate2([Atom],[Atom],[Atom]) :- !.

translate2({Prolog},{Prolog},[]) :- !.

translate2(Cat,CatTr,[Baum]) :-
    !, translate1(Cat,CatTr,Baum).
```

   Durch `!` verhindert man, daß `(A,B)`, ...,`{Prolog}` auch vom letzten Fall (als Kategorie) behandelt werden.

Bem.: Es gibt eine einfachere Übersetzung, wenn keine `(A;B)` in den Regeln der Quell-DCG vorkommen.

Übersetzung DCG $\mapsto$ DCG mit Baumausgabe

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

$\langle$*Beispiele/term$_e$xpansion1a.pl*$\rangle\equiv$

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

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].
```

Beispiel: Übersetzung in DCG+Baumausgabe

⟨*Ergebnis der Übersetzung DCG ↦ DCG+Baumausgabe*⟩≡

```
?- ['Beispiele/term_expansion1a',
    'Beispiele/programmierer'].
% ... Yes
?- listing.
s(A)  --> ((np(B), vp(C)), {append([B], [C], D)}),
          {A=..[s|D]}.
np(A) --> ((det(B), n(C)), {append([B], [C], D)}),
          {A=..[np|D]}.
vp(A) --> (v(B), {C=[B]}
         ; ((v(D), np(E)), {append([D], [E], F)}),
           {C=F}
         ),
         {A=..[vp|C]}.
det(A) --> ([der], {B=[der]};[das], {B=[das]}),
         {A=..[det|B]}.
n(A) --> (['Programmierer'], {B=['Programmierer']}
         ;['Programm'], {B=['Programm']}
         ),
        {A=..[n|B]}.
v(A) -->[steht], {A=..[v, steht]}.
v(A) -->[schrieb], {A=..[v, schrieb]}.
```

Von Hand würde man etwas kompakter übersetzen, etwa:

⟨*Beispiele/programmierer.baum.pl spaeter*⟩≡

```
%% vp --> v ; v, np.
vp(Baum) --> ( (v(BaumV), { Bäume = [BaumV] })
             ; (v(BaumV), np(BaumNP),
                          { Bäume = [BaumV,BaumNP] })
             ), { Baum =.. [vp|Bäume] }.
```

Übersetzung: CFG $\mapsto$ 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:

$\langle Parser/term_e xpansion1.pl \rangle \equiv$

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

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

Lädt man zuerst diese Datei und dann eine DCG-Datei, so wird die Grammatik in Prolog-Regeln übersetzt, die an die Kategorien (ohne Merkmale) eine Baumausgabe und die Argumente für die Ein- und Ausgabelisten von Wörtern anfügen.

Aus einer Kategorie `np` wird das Prolog-Prädikat

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

Beispiel: Übersetzung in Prolog+Baumausgabe

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

```
?- ['Parser/term_expansion1','Beispiele/programmierer'].
% ... Beispiele/programmierer compiled ...
?- listing.
n(A, B, C) :-
        (   B=['Programmierer'|C],
            D=['Programmierer']
        ;   B=['Programm'|C],
            D=['Programm']
        ),
        A=..[n|D].

det(A, B, C) :-
        (   B=[der|C],
            D=[der]
        ;   B=[das|C],
            D=[das]
        ),
        A=..[det|D].

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

Fortsetzung

⟨*Ergebnis* ⟩≡
```
np(A, B, C) :-
        det(D, B, E),
        n(F, E, C),
        append([D], [F], G),
        A=..[np|G].

v(A, B, C) :-
        B=[steht|C],
        A=..[v, steht].
v(A, B, C) :-
        B=[schrieb|C],
        A=..[v, schrieb].

s(A, B, C) :-
        np(D, B, E),
        vp(F, E, C),
        append([D], [F], G),
        A=..[s|G].
```

⟨*Aufruf zur Analyse einer Eingabe:*⟩≡
```
?- s(Baum,[der,'Programmierer',schrieb,das,'Programm'],L).

Baum = s(np(det(der), n('Programmierer')), vp(v(schrieb)))
L = [das, 'Programm'] ;

Baum = s(np(det(der), n('Programmierer')),
         vp(v(schrieb), np(det(das), n('Programm'))))
L = [] ;
No
```

Baumausgabe bei DCGs mit Merkmalen

Damit wir unsere Übersetzung nicht nur für DCGs mit Kategorien ohne Merkmale, wie `np`, bekommen, sondern auch für solche mit Merkmalen, wie `np([Genus],[Numerus,Kasus])`, brauchen wir eine etwas andere Übersetzung:

⟨*Parser/term$_e$xpansion2.pl*⟩≡

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

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

Im Modul `addTree` muß gegenüber dem Modul `addTreeCFG` nur der Hauptaufruf geändert werden:

⟨*Parser/addTree.pl*⟩≡

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

translate((L --> R),(Ltr --> (Rtr,{Code}))) :-
   translate1(L,Ltr,Baum),
   translate2(R,Rtr,Baumliste),
   Code = (L =.. [Kat|Merkmale],
           append(Merkmale,Baumliste,Args),
           Baum =.. [Kat|Args]).
```

Die übrigen Regeln sind genau wie in `addTreeCFG.pl`:

⟨*Parser/addTree.pl*⟩+≡

```
% translate1(+Kategorie,-Term,?Baum)

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

% translate2(+Cat,-CatTr,-Bäume)

translate2((A,B),((Atr,Btr),{Code}),Baeume) :-
    !, translate2(A,Atr,BaeumeA),
       translate2(B,Btr,BaeumeB),
    Code = append(BaeumeA,BaeumeB,Baeume).

translate2((A;B),(Atr;Btr),Baeume) :-
    !, translate2(A,Atr0,BaeumeA),
       translate2(B,Btr0,BaeumeB),
    Atr = (Atr0,{ Baeume=BaeumeA }),
    Btr = (Btr0,{ Baeume=BaeumeB }).

translate2([Atom],[Atom],[Atom]) :- !.
translate2([],[],[]) :- !.
translate2({Prolog},{Prolog},[]) :- !.

translate2(Cat,CatTr,[Baum]) :-
    !, translate1(Cat,CatTr,Baum).
```

Beispiel: Grammatik mit Merkmalen

⟨*Beispiele/dcg.merkmale.pl*⟩≡

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

startsymbol(s([_Temp])).

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,_Kas])
    ).

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].
```

⟨*Ergebnis der Übersetzung der* `s`*-Regel*⟩≡

```
s([A], B, C, D) :-
        np([E], [F, nom], G, C, H),
        vp([A, F], I, H, D),
        append([G], [I], J),
        s([A])=..[K|L],
        append(L, J, M),
        B=..[K|M].
Yes
?- 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'] ;

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, nom],
               det([neut, sg, nom], das),
               n([neut], [sg, nom], 'Programm'))))
Rest = []
```

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 mit `name/2` in Atome um:

⟨*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]
```
(Mit `char_code(Atom,Code)` kann man einzelne Code-Nummern umwandeln.)

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, wenn das Satzende erreicht ist, den 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) :-
    ((Seen = [C|Cs], satzende([C,Char]))
    -> reverse(Cs,Sentence)
    ; read_sentence(Stream,Sentence,[Char|Seen])).
```

Als Satz- bzw. Eingabeende gelte `".<Return>"`:

⟨*Parser/tokenizer.mini.pl*⟩+≡

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

Das Satzende wurde nicht als Teil des Satzes ausgegeben.

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 spalte dieses an den (einzelnen!) Leerzeichen:

⟨*Parser/tokenizer.mini.pl*⟩+≡
```
% tokenize(+String,-Atomlist).
tokenize(String,Atomlist) :-
    name(Atom,String),
    concat_atom(AtomlistK,' ',Atom),
    entferne_komma(AtomlistK,Atomlist).
```

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 zeige `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.

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),
    Term =.. [F|Args],
    append(Args,[Atomlist,[]],NewArgs),
    NewTerm =.. [F|NewArgs],
    nl,write('Aufruf: '), portray_clause(NewTerm),
    call(NewTerm), write('Baum:   '), writeq(Baum),nl,
    fail.
parse.
```

⟨*Anwendung auf die Beispielgrammatik*⟩≡

```
?- use_module('Parser/tokenizer.mini.pl').
?- [term_expansion2,'Beispiele/dcg.merkmale'].
                    % mit: startsymbol(s([_Temp]))).
?- parse.
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)))
```

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?

Entwicklung der Grammatik

Welche syntaktische 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>,...])
```

Nominalphrasen

⟨*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]).
```

Relativsätze in Nominalphrasen müssen noch erlaubt werden.

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].

n([mask],[sg,nom]) --> ['Astronom'].
```

Sätze

Logisch betrachtet erlaubt die Datenbank elementare Aussagen der Formen

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

Die hier angedeuteten semantischen Beschränkungen der Argumente werden in der Grammatik noch nicht berücksichtigt.

Definition der syntaktischen Kategorien

⟨*Grammatik/kategorien.pl*⟩≡

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

% importiere person/1,numerus/1,tempus/1,modus/1:
:- use_module(wortformen).

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 = rel(Gen,Num)
    ; Def = qu ; Def = quant).
kasus(X) :-
    (X = nom ; X = gen ; X = dat ; X = akk).
genus(Gen) :-
    (Gen = fem ; Gen = mask ; Gen = neut).

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

definitheit(s,Def) :-
    (Def = def ; Def = rel(Gen,Num) ; Def = qu).
verbstellung(Vst) :-
    (Vst = ve ; Vst = vz ; Vst = vl).
```

Def. der Kategorien (Fors.)

Weitere Kategorien sind die lexikalischen Kategorien:

⟨*Grammatik/kategorien.pl*⟩+≡

```
n([Gen],[Num,Kas]) :-
    genus(Gen),
    numerus(Num),
    kasus(Kas).

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

v([nom,akk],[Pers,Num,Temp,Mod]) :-
    Pers = 3,
    numerus(Num),
    tempus(Temp),
    modus(Mod).

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

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

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

s([rel(Gen,Num)],[Temp,Mod,vl]) -->
    np([rel(Gen,Num),3,_Gen1],[_Num1,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,nom]) --> ['Kepler'].
en([mask],[sg,gen]) --> ['Keplers'].

% n([mask],[sg,nom]) --> ['Planet']. % jetzt rn
% n([mask],[sg,nom]) --> ['Mond'].   % jetzt rn
```

Startsymbole

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*⟩≡
```
:- module(startsymbole,[startsymbol/1]).

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

Was muß man laden?

Zum Testen der Grammatik müssen bestimmte Dateien geladen werden. Das vereinfachen wir durch eine Datei, die die anderen nachlädt:

⟨*grammatiktest.pl*⟩≡

```
:- ['Parser/tokenizer.mini.pl'],
   ['Parser/term_expansion2'],
   ['Grammatik/np.pl',
    'Grammatik/saetze.pl',
    'Beispiele/testlexikon.pl',
    'Grammatik/startsymbole.pl'].
```

Die Datei `kategorien.pl` ist nur eine Dokumentation dessen, was als korrekte Kategorie erlaubt ist. Man könnte mit ihr eine Korrektheitsprüfung der DCG-Regeln zu schreiben.

Beispiel für Prädikativsätze

⟨*Beispiele für Prädikativsätze mit Prädikatsnomen*⟩≡

```
?- [grammatiktest].
% ... grammatiktest compiled ...
?- parse.
Beende die Eingabe mit <Punkt><Return>
|:ist Triton ein Planet.

Aufruf: s([A], [B, C, D], E,
          [ist, 'Triton', ein, 'Planet'], []).
Baum:   s([qu], [praes, ind, ve],
          v([sein], [3, sg, praes, ind], ist),
          np([def, 3, mask], [sg, nom],
             en([mask], [sg, nom], 'Triton')),
          np([indef, 3, mask], [sg, nom],
             det([indef], [mask, sg, nom], ein),
             n([mask], [sg, nom], 'Planet')))

?- parse.
Beende die Eingabe mit <Punkt><Return>
|: der ein Planet ist.

Aufruf: s([A], [B, C, D], E,
          [der, ein, 'Planet', ist], []).
Baum:   s([rel(mask, sg)], [praes, ind, vl],
          np([rel(mask,sg), 3, mask], [sg, nom],
             pron([rel], [mask, sg, nom], der)),
          np([indef, 3, mask], [sg, nom],
             det([indef], [mask, sg, nom], ein),
             n([mask], [sg, nom], 'Planet')),
          v([sein], [3, sg, praes, ind], ist))
```

Weitere Nominalphrasen

Wir brauchen noch Regeln für Nominalphrasen wie *der Astronom Kepler* und für Nominalphrasen mit einem Relativsatz:

⟨*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]).

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

Vorsicht: Kommata sollten beim Relativsatz nicht auftreten: sie werden vom Tokenizer in das vorangehende Wort eingebaut.

Im Plural sind auch Nominalphrasen ohne Artikel üblich:

⟨*Grammatik/np.pl*⟩+≡

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

Nominalphrasen mit Genitivattribut

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 Genitiobjekt) erlaubt; diese dürfen aber auch „absolut“ verwendet werden.

⟨*Beispiele/testlexikon.pl*⟩+≡

```
% im Lexikon, wegen weiterer n's(!):
n([Gen],[Num,Kas]) --> rn([Gen],[Num,Kas]).

rn([mask],[sg,nom]) --> ['Durchmesser'].
rn([mask],[sg,nom]) --> ['Planet'].
rn([mask],[sg,nom]) --> ['Mond'].
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].
```

Anpassung der Kategoriendefinition

Bem. Statt `rn` sollte ein Artmerkmal bei `n` erlaubt sein.

... (Hier sind einige Kategoriendefinitionen nachzutragen) ...

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([],[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([],[komp]).

s([Def1],[Temp,Mod,vl]) -->
    np([Def1,3,_Gen1],[Num,nom]),
    { member(Def1,[def,indef,quant,qu, rel(_Gen2,_Num2)]) },
    ap([],[komp]),
    v([sein],[3,Num,Temp,Mod]).

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

Maßangaben

Es fehlen noch die Größenangaben wie `50000 km`. Wir klassifizieren hier Zahlen n als Artikel (wegen $n > 12$ im Plural):

⟨*Beispiele/testlexikon.pl*⟩+≡
```
det([def],[_Gen,pl,_Kas]) --> [Zahlatom],
    { anzahl(Zahlatom,Zahl), Zahl >= 2 }.

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

anzahl(Atom) :-
    atom_codes(Atom,Codes),
    number_codes(1234567890,NumCodes),
    subset(Codes,NumCodes).
anzahl(ZahlAtom,N) :-
    anzahl(ZahlAtom),
    !, atom_codes(ZahlAtom,Codes),
    number_codes(N,Codes).
```

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

⟨*Grammatik/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([],[komp])).
```

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]) },
      { ( [Num,nom,akk] = [Num1,Kas1,Kas2]
        ; [Num,nom,akk] = [Num2,Kas2,Kas1]
        ), ((qu = Def1 ; qu = Def2)
           -> 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(Gen,Num)],[Temp,Mod,vl]) -->
    np([rel(Gen,Num),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] )
    }.
```

⟨*Beispiele/testlexikon.pl*⟩+≡

```
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'].
```

Lexikalische Regeln der DCG

⟨*Grammatik/lexikon$_d$etpron.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,akk]) --> [einen].
det([def],  [mask,sg,akk]) --> [den].
det([qu],   [mask,sg,akk]) --> [welchen].
det([quant],[mask,sg,akk]) --> [jeden].

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

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

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

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

anzahl(Atom) :-
    atom_codes(Atom,Codes),
    number_codes(1234567890,NumCodes),
    subset(Codes,NumCodes).
anzahl(ZahlAtom,N) :-
    anzahl(ZahlAtom),
    !, atom_codes(ZahlAtom,Codes),
    number_codes(N,Codes).

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].

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

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

```
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].

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) }.

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

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

% Nicht erzeugte Formen:
% en([mask],[sg,gen]) --> ['Uranus\''].
% en([mask],[sg,gen]) --> ['Triton'].
```

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

Ein solches Vollformenlexikon, `lexikon_verbnomen.pl`, wird nun aus einem Stammlexikon erzeugt.

(Fehlende Lexikoneinträge sind ggf. nachzutragen.)

Nomen-, Eigennamen- und Verblexikon

Die Vollformen für Nomen, Eigennamen und Verben erzeugen wir mit Flexionsprogrammen[1] aus Stammformen.

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

⟨*Grammatik/stammformenlexikon.pl*⟩≡

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

% lex(?Stammform,?Wortart,?Artmerkmale,?Flexionsklasse)

lex(entdecken,      v,[nom,akk],rg(0)).
lex(umkreisen,      v,[nom,akk],rg(0)).

lex('Astronom',     n,[mask], (s2e,p3e)).
lex('Durchmesser', rn,[mask], (s1, p2 )).
lex('Himmelskörper',n,[mask], (s1, p2 )).
lex('Mond',        rn,[mask], (s1e,p1 )).
lex('Planet',      rn,[mask], (s2e,p3e)).
lex('Sonne',        n,[fem],  (s3, p3 )).

lex('Galilei',     en,[mask],(s1,-)).
lex('Kepler',      en,[mask],(s1,-)).
lex('Triton',      en,[mask],(s1,-)).
lex('Uranus',      en,[mask],(s1,-)).
lex('Venus',       en,[fem], (s3,-)).
```

[1] Nomenflexion: siehe Anhang

Erzeugung der Nomen- und Verbvollformen

Daraus muß mit `erzeuge_vollformenlexikon` das Vollformenlexikon zu Nomen, Eigennamen und Verben gebildet werden:

⟨*Erstellung der Nomen- und Verbvollformen*⟩≡

```
?- ['Morphologie/flexion_slex',
    'Grammatik/stammformenlexikon'].
...
% stammformenlexikon compiled into ...

?- erzeuge_vollformenlexikon('Grammatik/lexikon_verbnomen.pl').
```

Das muß man nach Änderungen in `stammformenlexikon.pl` machen, um `lexikon_verbnomen.pl` zu erneuern. (Fehler: Doppelte Verbeinträge)

Laden (vorläufig)

⟨*laden.pl*⟩≡

```
:- ['Parser/tokenizer.mini.pl'],
   ['Parser/term_expansion2'],
   ['Grammatik/np.pl',
    'Grammatik/saetze.pl',
    'Grammatik/lexikon_detpron.pl',   % Artikel,Pronomen
    'Grammatik/lexikon_verbnomen.pl', % Verben,Nomen,Namen
    'Grammatik/startsymbole.pl',
    'Grammatik/dot.syntaxbaum.pl'],
   ['Semantik/datenbank.pl'].
```

Dann kann man `parse/0` aufrufen.

Testsätze (unfertig)

Zum Testen der Grammatik ist es nützlich, wenn man Beispielausdrücke nicht nur mit `parse/0` an der Konsole eingeben kann, sondern auch aus einer Datei lesen kann. Dazu dient:

⟨*Erweiterung von read$_s$entence*⟩≡

...

Folgende Beispielsätze wurden erkannt und müssen 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 Planet Venus, der 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.

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

Nachtrag: Nomenflexion

Zuerst muß die Datei `wortformen.pl` um die Definition der Formen der Nomen und Relationsnomen 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]).
```

Das Programm `erzeuge_vollformenlexikon/1` braucht noch eine Definition der Nomenflexion.

⟨*Morphologie/flexion*$_s$*lex.pl*⟩+≡
```
% ------------- Nomenflexion ----------------
% dekliniere(+Flex.klasse,+NomSg,?[Num,Kas],-Vollform)
dekliniere(Deklinationsklasse,Stammform,
           Formmerkmale,Vollform) :-
    deklination(Deklinationsklasse,
                Formmerkmale,Endung),
    concat(Stammform,Endung,Vollform).

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

Deklinationstabellen

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$_s$lex.pl*⟩+≡

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

deklination((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

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],'').
```

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

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

Graphische Baumdarstellung

Da die Termdarstellung des Syntaxbaums bei größeren Ausdrücken unleserlich ist, fügen wir noch eine graphische Darstellung hinzu, die das Programm `graphviz/dot` benutzt:

⟨*Parser/tokenizer.mini.pl*⟩+≡

```
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),
    Term =.. [F|Args],
    append(Args,[Atomlist,[]],NewArgs),
    NewTerm =.. [F|NewArgs],
    call(NewTerm).
```

Das Prädikat `displayTrees/1` wird von der Moduldatei

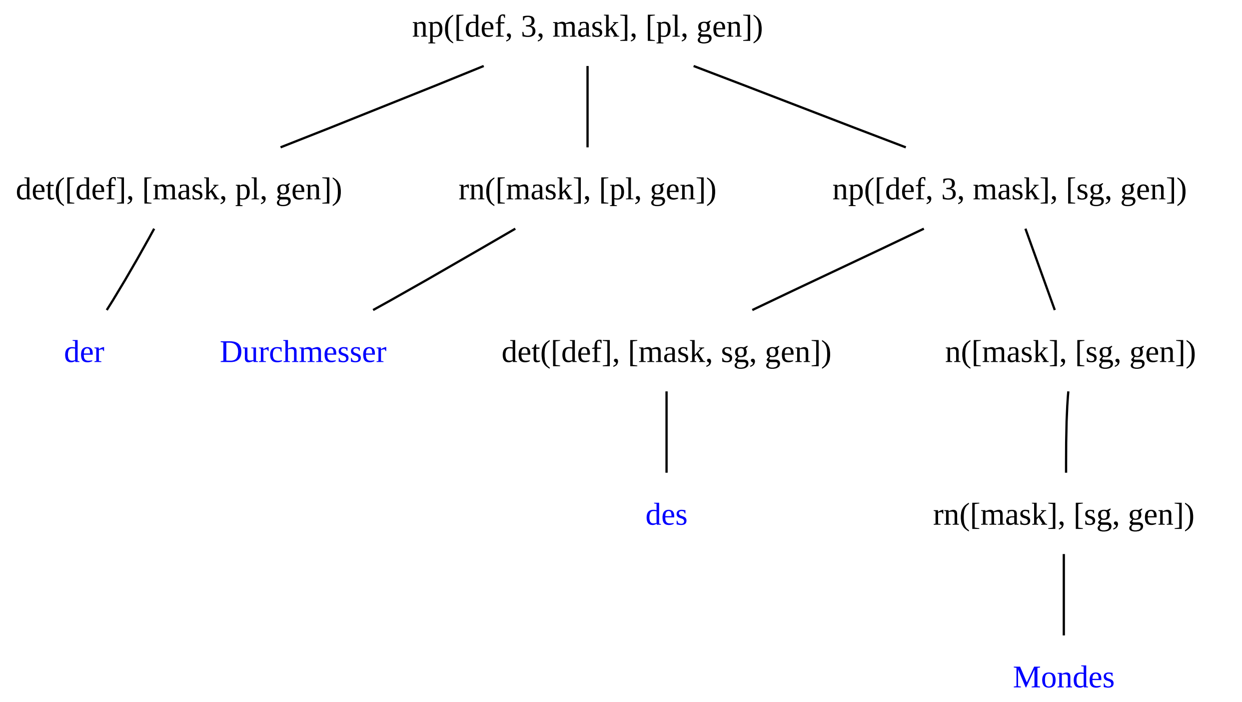
`Grammatik/dot.syntaxbaum.pl`

exportiert. Es steht nicht auf diesen Folien; siehe dazu das Unterverzeichnis `Grammatik`.

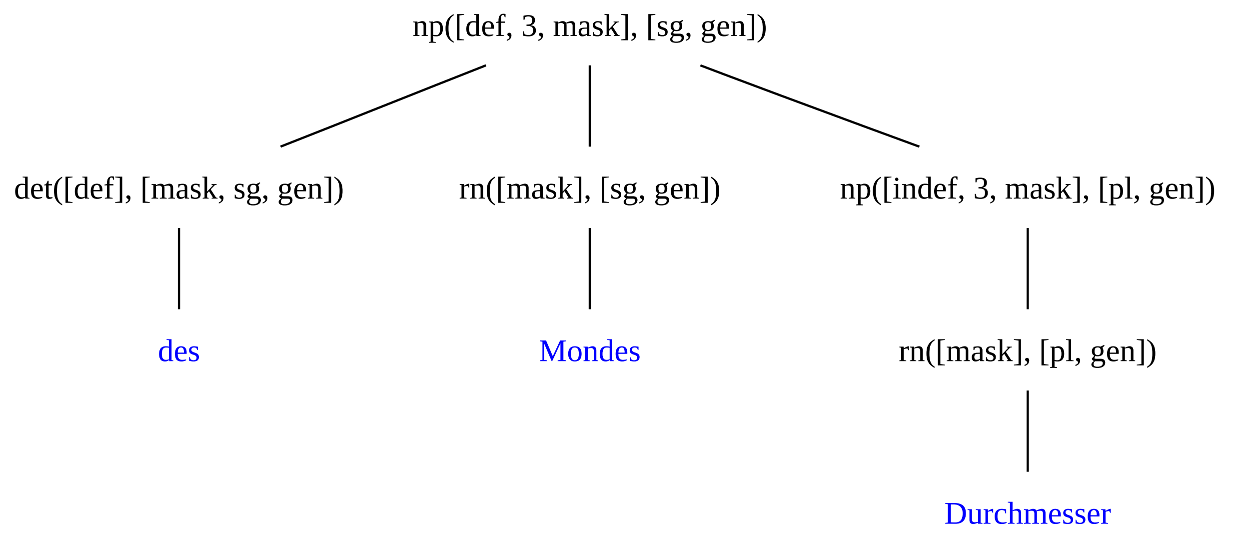
Durch `displayTrees(Trees)` werden Syntaxbäume im `.dot`-Format in die Datei `testbaum.tmp.dot` geschrieben, aus der `/bin/dot` eine Postscriptdatei `testbaum.tmp.ps` erzeugt, die mit Anzeiger `/usr/X11R6/bin/gv` am Bildschirm gezeigt wird. Diese Dateien werden bei der nächsten Analyse überschrieben.

Anzeige von `testbaum.tmp.ps`

Da wir Genitivattribute wie in



zugelassen haben und gleichzeitig Nominalphrasen im Plural ohne Artikel, erhalten wir unerwünschte Lesarten wie z.B.



Lexikalische Regeln der DCG (Forts.)

Bisher hatten wir keine Artikel und Pronomen im Plural.

⟨*Grammatik/lexikon$_d$etpron.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,akk]) --> [einige].
det([def],  [mask,pl,akk]) --> [die].
det([qu],   [mask,pl,akk]) --> [welche].
det([quant],[mask,pl,akk]) --> [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],[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,akk]) --> [einige].
det([def],  [fem, pl,akk]) --> [die].
det([qu],   [fem, pl,akk]) --> [welche].
det([quant],[fem, pl,akk]) --> [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].

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].
```

Weitere Testsätze

⟨*Grammatik/testsaetze.txt*⟩+≡

```
ist der Durchmesser der Venus kleiner als der Durchmesser der Sonne.
welcher Mond eines Planeten ist kleiner als der Durchmesser des Uranus.
sind die Planeten die Monde der Venus.
sind die Planeten Monde der Sonne.

welcher Mond des Uranus.
welcher Mond, der Uranus ist.  % nur als NP; , wird geloescht

ist der Planet Venus die Venus.
ist die Venus ein Mond des Uranus.
der Uranus ist ein Himmelkörper, dessen Durchmesser kleiner als 50000 km ist.
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.

ein Himmelskörper, dessen Monde Planeten sind.
```

⟨*Korrekterweise werden abgelehnt:*⟩≡

```
% Ellipse:
ist der Durchmesser der Venus kleiner als der der Sonne.
% Eigennamen mit nicht-definitem Artikel:
eine Venus. jede Venus. welche Venus.
% Possesiv und Genitiv-Attribut bei absoluten Nomina:
dessen Astronom
der Astronom der Venus
```

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:

⟨*Semantik/datenbank.pl*⟩≡

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

% Datensaetze:
% 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).

db( adrastea, mond,      ??,         _, jupiter).
db( amalthea, mond,     270,         _, jupiter).
db(   ananke, mond,      30,nicholson, jupiter).
db(    ariel, mond,     900,  lassell,  uranus).
db(    carme, mond,      30,  melotte, jupiter).
db(   charon, mond,    1500,         _, jupiter).
db(   deimos, mond,      15,     hall,    mars).
db(    diana, mond,    1100,  cassini,  saturn).
db(    dione, mond,      ??, herschel,  saturn).
db(    elara, mond,      30,nicholson, jupiter).
db(enkeladus, mond,     500, herschel,  saturn).
db(   europa, mond,    3126,  galilei, jupiter).
db(  ganymed, mond,    5276,  galilei, jupiter).
db(  himalia, mond,      30,  perrine, jupiter).
db( hyperion, mond,     400,     bond,  saturn).
db(  iapetus, mond,    1500,  cassini,  saturn).
db(       io, mond,    3640,  galilei, jupiter).
db( kallisto, mond,    4848,  galilei, jupiter).
db(    mimas, mond,     390, herschel,  saturn).
db(     mond, mond,    3473,         _,    erde).

db(    titan, mond,    5150, huyghens,  saturn).
db(   triton, mond,    5000,  lassell,  neptun).
```

. . . und viele weitere Einträge. (Die sind nachzutragen).

Datenbank (Forts.)

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

⟨*Semantik/datenbank.pl*⟩+≡

```
% Zugriffspraedikate:

stern(S) :-    db(S,_,_,_,_).
astronom(E) :- db(_,_,_,A,_), nonvar(A), E = A.
sonne(S) :-    db(S,sonne,_,_,_).

planet(S,Z) :- db(S,planet,_,_,Z).
mond(S,Z) :-   db(S,mond,  _,_,Z).

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

Prädikatenlogik (PL) als Datenbanksprache

Wir wollen als (maschinen-) interne 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     % der Datenbank: mars,...
Formel :=
     datenbank:Grundpraedikat(Terme)
   | Zahl < Zahl           | Zahl =< Zahl
   | eq(Term,Term)         | neg(Formel)
   | (Formel & Formel)     | (Formel v Formel)
   | (Formel => Formel)    | (Formel <=> Formel)
   | ex(Variable,Formel)   | all(Variable,Formel).
```

Voraussetzung: Alle Variablen `V` bei den Teilformeln `ex(V,F)` und `all(V,F)` 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 `&`, `v`, `=>` als Infix-Operatoren für *und*, *oder*, und *wenn-dann*, wobei `&` enger binden soll als `v` und `=>`:

⟨*Semantik/auswertung.pl*⟩≡

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

Trotz der Bindungsstärken: Klammern besser hinschreiben!

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

   ⟨*Semantik/auswertung.pl*⟩+≡
   ```
   wahr(astronom(X)) :-
       datenbank:astronom(X),!.
   wahr(stern(X)) :-
       datenbank:stern(X),!.
   wahr(sonne(X)) :-
       datenbank:sonne(X),!.

   wahr(planet(X,Y)) :-
       datenbank:planet(X,Y),!.
   wahr(mond(X,Y)) :-
       datenbank:mond(X,Y),!.
   wahr(durchmesser(X,Y)) :-
       datenbank:durchmesser(X,Y),!.

   wahr(entdecken(X,Y)) :-
       datenbank:entdecken(X,Y),!.
   wahr(umkreisen(X,Y)) :-
       datenbank:umkreisen(X,Y),!.

   wahr(mond(X)) :- wahr(ex(Y,mond(X,Y))).
   wahr(planet(X)) :- wahr(ex(Y,planet(X,Y))).
   ```

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

   ⟨*Semantik/auswertung.pl*⟩+≡
   ```
   wahr((F & G)) :- wahr(F) , wahr(G).
   wahr((F v G)) :- wahr(neg(neg(F) & neg(G))).
   wahr((F => G)) :-  wahr(neg(F) v G).
   wahr((F <=> G)) :- wahr((F => G) & (G => F)).
   wahr(neg(F)) :- falsch(F).

   falsch(F) :- (wahr(F) -> fail ; true).
   ```

3. 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)))).
   ```

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

   ⟨*Semantik/auswertung.pl*⟩+≡
   ```
   wahr(ex(Var,Formel)) :-
       objekt(Var), wahr(Formel), !.
   objekt(X) :-
       datenbank:stern(X)
       ; datenbank:astronom(X)
       ; datenbank:durchmesser(X,_).
   ```

   Die (teure) Suche in der endlichen Datenbank terminiert.

5. 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.
   ```

6. 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.
   ```

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

Beispiele: Ja/Nein-Fragen als Aussagen in PL

Frage: *Entdeckte Herschel einen Planeten der Sonne?*

⟨*Anfrage in Prolog*⟩≡
```
:- [laden], ['Semantik/auswertung.pl'].
?- wahr(ex(X,planet(X,sonne)
        & entdecken(herschel,X))).
```

Frage: *Entdeckte Herschel jeden Planeten der Sonne?*

⟨*Anfrage in Prolog*⟩+≡
```
?- wahr(all(X,(planet(X,sonne)
              => entdecken(herschel,X)))).
```

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

⟨*Anfrage in Prolog*⟩+≡
```
?- wahr(durchmesser(DE,erde)
       & durchmesser(DU,uranus) & (DE < DU)).
```

Bei dieser Formulierung setzen wir voraus, daß jeder Stern nur einen Durchmesser hat. (`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,planet(X,sonne) & entdecken(herschel,X))).
X = uranus
Yes
```

Da wir prädikatenlogische Variable durch Prolog-Variable dargestellt haben, wird deren Belegung ausgegeben – auch für *gebundene* prädikatenlogische Variablen!

Wenn man mit den Wahrheitswerten weiterrechnen will, sollte man sie als Antwort ausgeben:

⟨*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: `setof(+Muster,+Ziel,-Liste)`

sucht alle Beweise von `Ziel`, wobei freie Prolog-Variablen belegt werden, sammelt pro Belegung der Variablen von Ziel, die nicht im Muster vorkommen, die durch diese Belegungen entstehenden Instanzen von `Muster` ohne Duplikate in `Liste`:

⟨*Lösungen mit* `setof`⟩≡

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

E = _G147
Y = carme
Es = [melotte] ;

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.

Prolog: `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
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-∃-Quantifizierung kann nicht in normalen Zielen verwendet werden, nur innerhalb `setof/3,bagof/3`.

Prolog: `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 einfach 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 berechen 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`:

⟨*Wert einer Formel: Antwortmenge*⟩≡

```
?- 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]
```

Um die durch ein Fragepronomen, `wer` oder `was`, gebildeten Fragen zu beantworten, werden i.a. alle Objekte durchlaufen:

⟨*Semantik/auswertung.pl*⟩+≡

```
antworten(qu(Var,Formel), As) :-
    findall(Var,(objekt(Var), wahr(Formel)),Ws),
    sort(Ws,As).
```

Wenn der zu durchsuchende Teilbereich aus der interrogativen Konstituente hervorgeht, wie bei `welcher Astronom`, ließe sich die Suche einschränken.

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 α $\mapsto$ Syntaxbaum t,
2. Syntaxbaum t $\mapsto$ logische Formel φ,
3. PL-Formel $\varphi(x)$ $\mapsto$ Wert $\{a \in D \mid \mathit{wahr}(\varphi[a/x])\}$

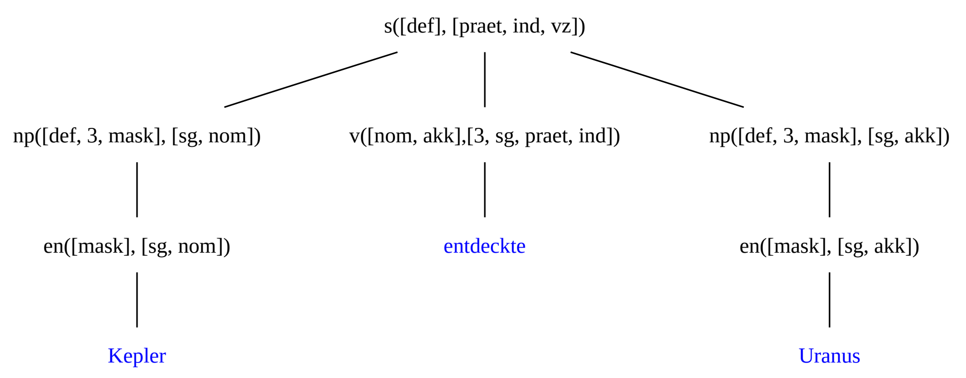
Wie berechnet man aus dem Syntaxbaum die passende Formel?

Was ist „die“ passende Formel?

Atomare Aussage:

Aussage: `Kepler entdeckte Uranus.`

Baum:



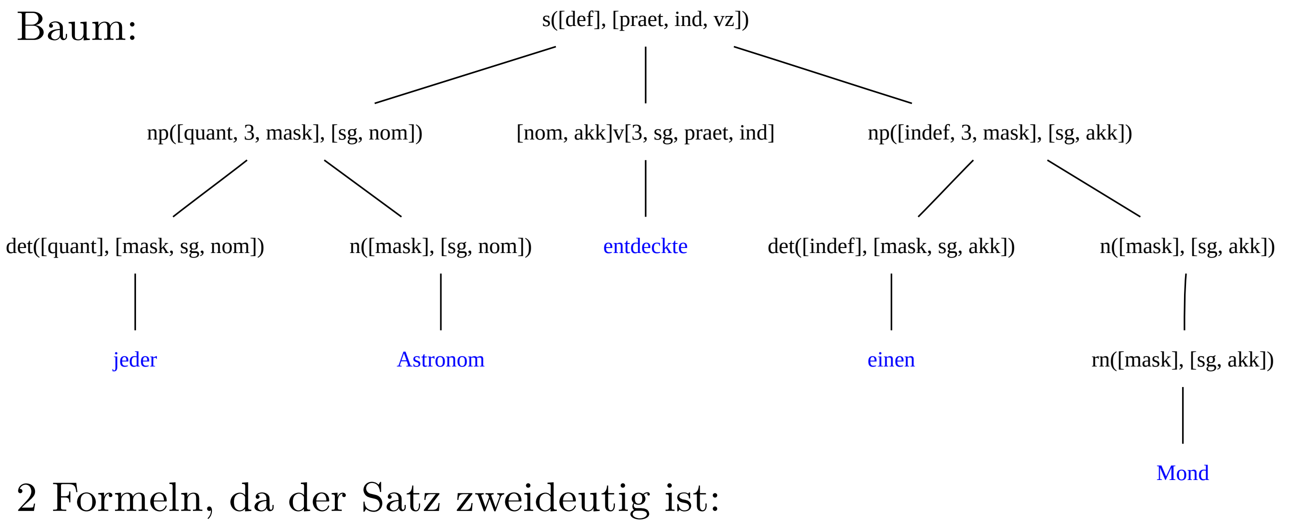
Formel: `entdecken(kepler,uranus).`

Dasselbe sollte bei anderen Wortstellungen herauskommen.

Quantifizierte Nominalphrasen:

Aussage: `Jeder Astronom entdeckte einen Mond.`

Baum:



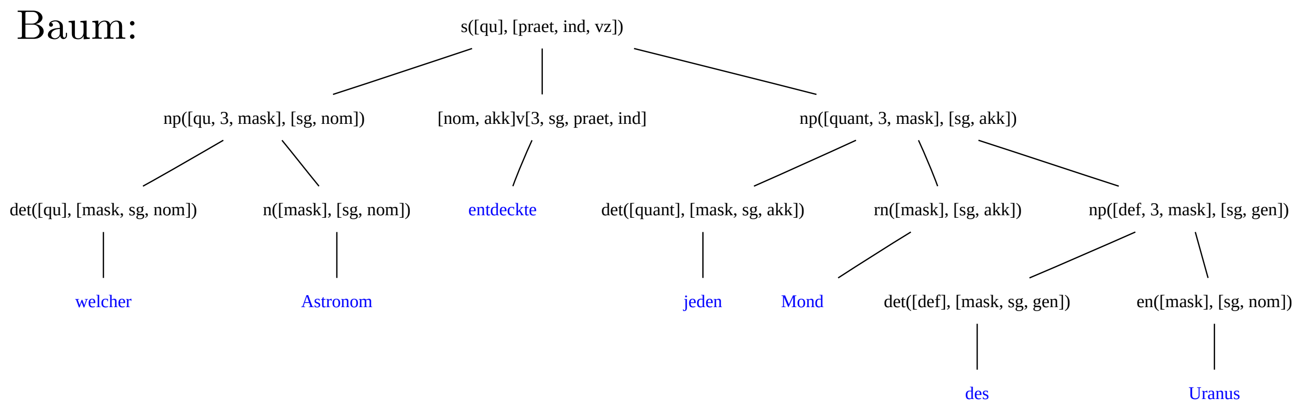
2 Formeln, da der Satz zweideutig ist:

```
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:
```
{ X | astronom(X) /\
      all(Y,mond(Y,uranus) => entdecken(X,Y)) }
```

Was ist der schwierige Teil?

Natürliche Sprache:

- Verben bedeuten eine Relation zwischen Individuen,
- Verbargumente sind u.a quantifizierte Nominalphrasen,
- quantifizierte Nominalphrasen bedeuten *keine* Individuen,
- der Wirkungsbereich der Quantoren ist nicht eindeutig.

Prädikatenlogik (erster Stufe):

- Prädikate bedeuten Relationen zwischen Individuen,
- alle Prädikatargumente, d.h.Terme, bedeuten Individuen,
- Quantoren (in Formeln!) wirken auf die folgende Formel

Bedeutung einer quantifizierten NP?

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

$$\begin{aligned}
(\textit{jeder } M \textit{ ist ein } L) &\mapsto M \subseteq L,\\
(\textit{eín } M \textit{ ist ein } L) &\mapsto M \cap L \neq \emptyset,\\
(\textit{eín } M \textit{ ist kein } L) &\mapsto M \not\subseteq L,\\
(\textit{kein } M \textit{ ist ein } L) &\mapsto M \cap L = \emptyset,
\end{aligned}$$

Unvollständig: Quantoren in Objektposition, andere Verben?

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

$$\begin{aligned}
(\textit{jeder } M \textit{ singt ein } L) &\mapsto \forall x \in M\, \exists y \in L\ (x \textit{ singt } y),\\
(\textit{eín } M \textit{ singt kein } L) &\mapsto \exists x \in M\ Ky \in L\ (x \textit{ singt } y),\\
&= \exists x \in M\ \neg\exists y \in L\ (x \textit{ singt } y),
\end{aligned}$$

Allgemein für einfache Aussagen (Verb mit Komplementen):

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

Mathematisch ausgedrückt:

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

Daher wollen wir übersetzen:

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

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

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 $\mathcal{D}$. Dazu:

$$\frac{r \to t, \quad s \to u}{(r\cdot s) \to (t\cdot u)}\,(\mathrm{cong}_1) \qquad \frac{r \to s}{\lambda x\, r \to \lambda x\, s}\,(\mathrm{cong}_2)$$

$$\frac{}{(\lambda x\, t\cdot s) \to t[x/s]}\,(\beta) \qquad \frac{x \notin \mathit{frei}(t)}{\lambda x\,(t\cdot x) \to t}\,(\eta)$$

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 $y \in \mathit{frei}(s)$ in den Wirkungsbereich eines λy von t gerät.

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

Vereinfachung (Reduktion) von λ-Termen

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

Durch Anwenden der Reduktionsregeln können neue Redexe entstehen, wie in

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

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

Die „Vereinfachung“ kann sogar divergieren! Aber wenn man Variablen mit *Typen*

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

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:

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

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

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

Semantik: 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-Konversion)

⟨*Semantik/lambdaTerme.pl*⟩≡

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

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

normalize(Term,Nf) :-
    beta(Term,Red,Tag),
    normalizeLO(Red,Nf,Tag), !.

normalizeLO(T,T,not) :- !.
normalizeLO(T,Nf,chd) :-
    normalize(T,Nf), !.

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

β-Reduktion

⟨*Semantik/lambdaTerme.pl*⟩+≡

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

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

beta(M*N, M*ConN, Tag) :-
    var(M),
    beta(N,ConN,Tag), !.

beta(lam(X,M)*N, MDup, chd) :-
    alpha(lam(X,M),lam(XDup,MDup)),
    XDup = N, !.

beta(M*N, T, Tag) :-
    beta(M,ConM,TagL),
    wrLO(TagL,M,ConM,N,T,Tag), !.  % s.u.

beta(lam(V,T), lam(V,ConT), Tag) :-
    beta(T,ConT,Tag), !.

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

beta(V, V, not) :- !.
```

Bem. Die `!` verhindern andere Vereinfachungsmöglichkeiten

Reduktion von t im obersten $t \cdot s$

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

```
% Schwache leftmost-outermost-Reduktion

wrLO(not, M, _, N, M*ConN, Tag) :-
    beta(N,ConN, Tag), !.

wrLO(chd, _,ConM, N, ConM*N, chd) :- !.
```

α-Reduktion: Umbenennung λ-geb. Variable

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

```
% Alpha-Reduktion:

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

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

alpha_list(lam(Var,Body), L, lam(New,BodyDup)) :-
    !, alpha_list(Body,[(Var,New)|L],BodyDup).

alpha_list(M*N, L, MDup*NDup) :-
    !, alpha_list(M,L,MDup),
    alpha_list(N,L,NDup).

alpha_list(F,_, F) :-
    atomic(F), !.
```

α-Reduktion (Forts.)

⟨*Semantik/lambdaTerme.pl*⟩+≡

```
alpha_list(T, L, TDup) :-  % für:  f(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,[]).

rename(V, [], V).
rename(V, [(Var,Dup)|_], Dup) :-
    V == Var, !.
rename(V, [_|C], Dup) :-
    !, rename(V,C,Dup).
```

2 Beispiele

⟨*Beispiel zur α-Reduktion*⟩≡

```
?- alpha(X*lam(X,(Y*X)), Dup).
X = _G148
Y = _G147
Dup = _G148*lam(_G251, _G147*_G251)
```

⟨*Beispiel zur β-Reduktion*⟩≡

```
?- normalize(lam(X,(X*Y)*(X*Y)) * lam(V,c*V), Nf).
X = _G147
Y = _G148
V = _G160
Nf = c*_G148 * (c*_G148)
```

Übersetzung NL nach PL

Möglichkeit A: den Aufbau der logischen Formel in die Grammatikregeln einbauen. (Aber wie bei Regeln mit Alternativen?)

Aus der Satzregel $S \to NP\ \ TV\ \ NP$ wird eine Syntaxregel „mit Semantik“:

$$S[Sem(A, B, R)] \to NP[A]\ \ TV[R]\ \ NP[B]$$

wobei die Semantik ein λ-Term ist, der aus den λ-Termen A, B, C der Teilausdrücke aufgebaut wird.

Wird bei TV das transitive Verb r benutzt, so soll R der Term $\lambda x \lambda y\, r(x, y)$ sein, worin $r(x, y)$ eine logische Formel ist. Sie ist die Normalform des λ-Terms $((R \cdot x) \cdot y)$.

Sind die NP quantifizierte Nominalphrasen, $(\tilde{Q}\ \tilde{N})^{nom}$ und $(Q\ N)^{akk}$, so soll die Formel mit logischen Quantoren beginnen und an die Stelle der NPs 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 $S[Sem(A, B, R)]$ in den Regeln:

$$\begin{array}{lcl} S[A \cdot \lambda x(B \cdot \lambda y\,((R \cdot x) \cdot y))] & \to & NP[A]\ \ TV[R]\ \ NP[B] \\ S[B \cdot \lambda y\,(A \cdot \lambda x\,((R \cdot x) \cdot y))] & \to & NP[A]\ \ TV[R]\ \ NP[B] \end{array}$$

Möglichkeit B: die logische Formel aus dem Syntaxbaum berechnen. (Dabei werden dieselben λ-Terme benutzt, s.u.)

Beispiel: Semantik zu einem NP-Baum

Zur Analyse von *jeder Astronom* muß ein λ-Term nach folgenden Grammatikregeln „mit Semantik“ angefügt werden:

$$\begin{array}{rcl} n[\lambda x\, astronom(x)] & \to & Astronom \\ det[\lambda N\, \lambda P\, (\forall x(N \cdot x \to P \cdot x))] & \to & jeder \\ np[D \cdot N] & \to & det[D]\; n[N] \end{array}$$

Berechnet man den λ-Term nicht (wie hiermit) *während* der Syntaxanalyse, sondern *nachher* (ohne Grammatikänderung), so braucht man ein Prädikat `sem(+Syntaxbaum,-LamTerm)`:

⟨*Beispiele/semNP.pl*⟩≡

```
:- 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)*⟩≡

```
?- 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))
```

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,v),
     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 Objekte der Datenbank, werden aber nicht konstante λ-Terme, sondern deren Eigenschaften:

   ⟨*Semantik/sem.pl*⟩+≡
   ```
   sem(en(Art,Form, Vollform),LamTerm) :-
       wort(Stammform, en(Art,Form), Vollform),
       kleinschreibung(Stammform,EN),
       LamTerm = lam(P,P*EN).
   ```

2. Absolute Nomen bedeuten Eigenschaften, Relationsnomen und Verben Beziehungen zwischen Objekten. Sie werden zu λ-Termen, die bei Anwendung auf eine bzw. zwei Konstante die passende atomare Formel ergeben:

   ⟨*Semantik/sem.pl*⟩+≡
   ```
   sem(n(Art,Form,Vollform),LamTerm) :-
       wort(Stammform, n(Art,Form), Vollform),
       kleinschreibung(Stammform,Stamm),
       Formel =.. [Stamm,X],
       LamTerm = lam(X,Formel).

   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,VollformKlein),
       wort(Stammform, v([nom,akk],Form),VollformKlein),
       Formel =.. [Stammform,X,Y],
       LamTerm = lam(X,lam(Y,Formel)).
   ```

   Absolute Nomina, die aus Relationsnomen gebildet werden, bedeuten die Projektion des Relationsnomens:

   ⟨*Semantik/sem.pl*⟩+≡
   ```
   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).
   ```

3. Artikel ($\neq$ `qu`) bedeuten Funktionen, die zwei Eigenschaften von Objekten einen Wahrheitswert zuordnen:

$$\begin{array}{rcl} \textit{jeder } N\ \ VP & = & (N \subseteq VP)\,? \\ \textit{ein } N\ \ VP & = & (N \cap VP \neq \emptyset)\,? \\ k\ N\ \ VP & = & (\,|N \cap VP| \geq k)\,? \\ \textit{der } N\ \ VP & = & (N \subseteq VP \wedge |N| = 1)\,? \end{array}$$

Sie werden zu λ-Termen, die bei Anwendung auf zwei Eigenschaften eine passende Formel liefern:

⟨*Semantik/sem.pl*⟩+≡

```
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)))).

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*X)))))).
```

4. Interrogativartikel bedeuten Funktionen, die zwei Eigenschaften von Objekten die Menge derjenigen Objekte, die korrekte Antworten bedeuten, zuordnen:

$$\textit{welcher } N\ \ VP \quad = \quad \{a \in N \mid a \in VP\}$$

⟨*Semantik/sem.pl*⟩+≡
```
sem(det([qu], _Form, _Vollform),
    lam(N,lam(P,qu(X,N*X & P*X)))).
```

5. 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,(X > Y)))).
   ```

   (Für die attributive Verwendung der Adjektive muß man diese Bedeutung über die Stammform holen.)

6. Relativ- und Interrogativpronomen bedeuten Eigenschaften von Individuen, die ggf. als erfragt markiert werden:

   ⟨*Semantik/sem.pl*⟩+≡
   ```
   sem(pron([rel],_Form,_Rel), lam(P,P)).
   sem(pron([qu],_Form,_Pron), lam(P,qu(X,P*X))).
   ```

   Hier bewirkt `lam(P,P)` dasselbe wie `lam(P,lam(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)))).
   ```

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 Verzweigungform 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[2] `sem(+Syntaxbaum,-LamTerm)` je nach der Form

```
<Cat>(Formmerkmale,Artmerkmale, Teilbaum1, ...)
```

des Syntaxbaums eines 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*C)))
   ```

[2] Unvollständig: nur für die wichtigsten Konstruktionen, wenig getestet!

2. Eigennamen:

   ⟨*Semantik/sem.pl*⟩+≡
   ```
   sem(np([def,3,Gen],[Num,Kas],
          en([Gen],[Num,Kas],EN)), SemNP) :-
       sem(en([Gen],[Num,Kas],EN),SemNP).
   ```
   ⟨*Beispiel* ⟩+≡
   ```
   ?- parses. Kepler.
   lam(A, A*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),SemNP).
   ```
   ⟨*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,SemEN*SemN & SemEN*P),SemNP).
   ```

   ⟨*Beispiel* ⟩+≡
   ```
   ?- parses. der Astronom Kepler.
   lam(A, astronom(kepler)&A*kepler)
   ```

3. 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(A, A)
?- parses. wer.
lam(A, qu(B, A*B))
```

4. Relativierende Nominalphrase mit Possessiv:

   ⟨*Semantik/sem.pl*⟩+≡
```
sem(np([rel(Gen2,Num2),3,Gen],[Num,Kas],
       poss([rel],[Gen2,Num2,gen],Poss),
       rn([Gen],[Num,Kas],RN)),
    SemNP) :-
    sem(poss([rel],[Gen2,Num2,gen],Poss),SemPoss),
    sem(rn([Gen],[Num,Kas],RN),SemRN),
    normalize(SemPoss*SemRN,SemNP).
```

   ⟨*Beispiel* ⟩+≡
```
parses. dessen Planet.
Lambda: 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))
```

Man braucht aber eine andere Lesart dieser Konstruktion für `die Durchmesser aller Monde` oder für `ein Mond aller Planeten`.

7. Sätze mit Vollverb (in vz) (ergeben logische Formeln):

⟨*Semantik/sem.pl*⟩+≡
```
sem(s([_Def],[_,_,vz],
      NP1, v([nom,akk],[3,Num,_,ind],Verb), NP2),
    SemS)
:-
    sem(NP1,SemNP1),
    sem(v([nom,akk],[3,Num,_,ind],Verb), SemV),
    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).
```

Hier wird für `NP1` der weitere Wirkungsbereich erzeugt.

⟨*Beispiel* ⟩+≡
```
?- parses. Kepler entdeckte eine Sonne.
ex(A, sonne(A)&entdecken(kepler, A))

?- parses. Kepler entdeckte jeden Mond der Sonne.
all(A, ex(B,all(C,eq(C,B)<=>sonne(C)&mond(A,C)))
       => entdecken(kepler,A))

?- parses. jeder Astronom entdeckte
           einen Himmelskörper.
all(A, astronom(A) =>
```

```
        ex(B,himmelskörper(B)&entdecken(A,B)))

?- parses. einen Himmelskörper
           entdeckte jeder Astronom.
ex(A, himmelskörper(A)
      & all(B, astronom(B)=>entdecken(B, A)))
```

8. Ja/Nein-Fragen (interrogative Verberstsätze):

   Ist das Verb ein Vollverb, so soll die erste Nominalphrase weiten Wirkungsbereich bekommen:

   ⟨*Semantik/sem.pl*⟩+≡

```
sem(s([qu],[_,ind,ve],
      v([nom,akk],[3,Num,_,ind],Verb), NP1, NP2),
    SemS)
:-
    sem(NP1,SemNP1),
    sem(v([nom,akk],[3,Num,_,ind],Verb), SemV),
    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).
```

   ⟨*Beispiel* ⟩+≡

```
?- parses. umkreist jeder Mond einen Planeten.
Lambda: all(A, mond(A)=>
           ex(B, planet(B)&umkreisen(A,B)))
```

Bei Ja/Nein-Fragen mit Hilfsverb nehmen wir an, daß das prädikative (und indefinite) Nomen am Ende steht:[3]

⟨*Semantik/sem.pl*⟩+≡
```
sem(s([qu],[_,ind,ve],
      v([sein],[3,Num,_,ind],_Verb), NP1, NP2),
    SemS)
:-
    NP1 =.. [np,_,[Num,nom]|_Konstituenten1],
    NP2 =.. [np,[indef,3,_],[Num,nom]|_Konst2],
    sem(NP1,SemNP1),
    sem(NP2,SemNP2),
    Sem = SemNP1 * lam(X,SemNP2 * lam(Y,eq(X,Y))),
    normalize(Sem,SemS).
```

⟨*Beispiel* ⟩+≡
```
parses. ist Uranus ein Planet.
ex(A, planet(A)&eq(uranus, A))
```

Für Ja/Nein-Fragen mit Vergleich braucht man:

⟨*Semantik/sem.pl*⟩+≡
```
sem(s([qu],[_,ind,ve],
      v([sein],[3,Num,_,ind],_), NP1, AP),
    SemS)
:-
    NP1 =.. [np,_,[Num,nom]|_Konst1],
    AP  =.. [ap,[],[komp]|_Konst2],
    sem(NP1,SemNP1),
    sem(AP,SemAP),
    normalize(SemNP1*SemAP, SemS).
```

[3] *Fehler*: Ist `NP1` quantifiziert, erhält man keine oder eine falsche Lesart.

⟨*Beispiele* ⟩+≡

```
?- parses. sind 250 km kleiner als 300 km.
250<300
?- parses. ist der Durchmesser des Uranus
           kleiner als 3000 km.
ex(A, all(B, eq(B,A)<=>
             durchmesser(B,uranus)& (B<3000)))
```

9. Relativsätze: (mit Vollverb)

⟨*Semantik/sem.pl*⟩+≡

```
sem(s([rel(GenR,NumR)],[Tmp,ind,vl],
      NP1,
      NP2,
      v([nom,akk],[3,Num,Tmp,ind],Verb)),
    SemS)
:-
    NP1 =.. [np,[rel(GenR,NumR),3,GenR],[NumR,KasR]|_],
    sem(NP1,SemNP1),
    sem(v([nom,akk],[3,Num,Tmp,ind],Verb), SemV),
    sem(NP2,SemNP2),
    (KasR = nom,
        Sem = SemNP1 * lam(X,SemNP2 *
                           lam(Y,SemV * X * Y))
    ; KasR = akk,
        Sem = SemNP1 * lam(Y,SemNP2 *
                           lam(X,SemV * X * Y))
    ),
    normalize(Sem,SemS).
```

⟨*Beispiel* ⟩+≡

```
?- parses. der den Uranus umkreist.
```

```
lam(A, umkreisen(A, uranus))
?- parses. den der Uranus umkreist.
lam(A, umkreisen(uranus, A))
```

Es fehlt noch die Angabe der Bedeutung für Relativsätze mit NP- oder AP-Prädikativ.

10. 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],REL,NP,TV)),
    LamTerm)
:-
    sem(DET,SemDET),
    sem(n([Gen],[Num,Kas],N),SemN),
    sem(s([rel(Gen,Num)],[_,_,vl],REL,NP,TV),
        SemSREL),
    SemNREL = lam(X,(SemN * X) & (SemSREL * X)),
    normalize(SemDET * SemNREL, LamTerm).
```

⟨*Beispiel* ⟩+≡

```
?- parses. ein Astronom, der den Uranus entdeckte.
lam(A, ex(B, astronom(B)
            & entdecken(B,uranus)
            & A*B))
?- parses. jeder Himmelskörper,
           den ein Astronom entdeckte.
lam(A, all(B, himmelskörper(B)
             & ex(C,astronom(C)&entdecken(C,B))
             =>A*B))
```

11. Adjektivphrasen:

   ⟨*Semantik/sem.pl*⟩+≡
```
sem(ap([],[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(A, A>3000)

?- parses. größer als der Durchmesser des Uranus.
lam(A, ex(B, all(C, eq(C,B)<=>
               durchmesser(C,uranus)& (A>C))))
```

12. Weitere Fälle fehlen noch, insbesondere (a) die mit Adjektivphrasen (b) andere Lesarten des Genitivattributs

Laden der Grammatik und Semantik

⟨*semantik.pl*⟩≡
```
:- style_check(-discontiguous).
:- [laden], % Grammatik
   ['Semantik/datenbank.pl',
    'Semantik/auswertung.pl',
    'Semantik/lambdaTerme.pl',
    'Semantik/sem.pl'].
```

Zum Testen braucht man einen Parseraufruf mit Ausgabe der aus dem Syntaxbaum berechnteten Lambda-Terme:

⟨*semantik.pl*⟩+≡
```
parses :-
    write('Beende die Eingabe mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    startsymbol(S),
    addTree:translate1(S,Term,Baum),
    Term =.. [F|Args],
    append(Args,[Atomlist,[]],NewArgs),
    NewTerm =.. [F|NewArgs],
    nl,write('Aufruf: '), portray_clause(NewTerm),
    call(NewTerm), write('Baum:   '), writeq(Baum),nl,
    sem(Baum,LamTerm),
    write('Lambda: '),
    numbervars(LamTerm,0,_N), % für lesbare Variable
    write_term(LamTerm,[numbervars(true)]),nl,
    fail.
parses.
```

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:

⟨*semantik.pl*⟩+≡

```
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),
    Term =.. [F|Args],
    append(Args,[Atomlist,[]],NewArgs),
    NewTerm =.. [F|NewArgs],
    nl,write('Aufruf: '), portray_clause(NewTerm),
    (call(NewTerm) ->
        (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

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
```

Was bleibt noch zu tun?

1. Lexikon:
   (a) mehr Stammeinträge für Namen aus der Datenbank
   (b) entsprechende Vollformen erzeugen
2. Semantik (ggf. mit Änderungen der Syntax):
   (a) Übersetzung von Prädikativsätzen in Formeln
   (b) Behandlung der Genitivattribute (Quantorenskopus)
   (c) Behandlung der Maßausdrücke (Durchmesser, n km)
   (d) Behandlung von Anzahlfragen mit `(wieviele N)`
   (e) Bestimmte Artikel: weiter Quantorenskopus, `eq/2`
   (f) Alternative Wirkungsbereiche der Quantoren
3. Antwort: Die berechnete Lösung auf eine Frage sollte als Antwort in natürlicher Sprache dargestellt werden.
   (a) Ja/Nein-Fragen: die Frage durch Umstellen des Verbs in eine Aussage umformen. (Ggf.: Negation einfügen!)
   (b) W-Fragen: aus den Namen der Lösung eine Nominalphrase bilden und für die interrogative Konstituente in den Syntaxbaum der Frage einsetzen.

Ferienarbeit

Man versuche, fehlende Fälle in `sem/2` zu ergänzen, z.B.

1. Relativierung bei Relationsnomen
2. Adjektive und Vergleichsaussagen von Maßangaben
3. mehr zu Prädikativsätzen, Plural, Relativsätzen

Fehler in der Grammatik?

1. `der Durchmesser ist 50000 km` wird nicht erkannt, da `5 km` eine `NP(pl)` ist.

   Soll man die Numeruskongruenz aufheben? Nur bei einem neuen Verb `beträgt`?

2. `Die Sonne ist größer als alle Monde`: Die Vergleichsrelation war nur für Zahlen geplant, daher nur für definite NP (wie `23 km`) erlaubt.

   Soll/muß man Vergleiche beliebiger NPs erlauben?

3. `der Durchmesser aller Monde` wird nicht erkannt, denn `der Durchmesser det([Def],_) Mondes` wird eine `NP(def)`, aber nur für (in)definites `Def`.

   Sollte es eine `NP(Def)` werden? Das kollidiert mit `jeder Planet der Sonne`, was sicher eine `NP(quant)` sein sollte.

   Problem: eigentlich brauchen wir 2 Definitheiten, die des Kopfnomens und die des NP-Attributs (etc. rekursiv!). Sollte man *Funktions*- von *Relations*nomen trennen und die Definitheit unterschiedlich festlegen? (Linguisten??)