PCom 3 Erweiterungsmodul Computerlinguistik

Berechenbare Semantik
Vorlesung mit Übung SS 2014

Hans Leiß
Universität München
Centrum für Informations- und Sprachverarbeitung

30. Juni 2014

Organisatorisches

Vorlesung

- Zeit: Mo, 14-16 Uhr
- Ort: Raum U127, Oettingenstr. 67
- Voraussetzung:
  Computerlinguistische Anwendungen 2013
  Modelltheoretische Semantik SoSe 2013

Tafelübung:

- Zeit: Do 14-16 Uhr
- Ort: Raum 061, Oettingenstr. 67
- Programmierung

Modulprüfung

- Modulprüfung: Programmieraufgaben

Inhalt

In der Vorlesung wird eine modelltheoretische Semantik verschiedener natürlichsprachlicher Ausdrucksformen erklärt, u.a.

- Nominalphrasen im Singular und Plural, mit
  - verallgemeinerte Quantoren (*viele CN*; *mehr CN als CN*)
  - Identitäts- und Diversitätsquantoren (*derselbe CN*; *verschiedene CN*)
  - Reflexiv- und Reziprokpronomen (*TV.sg sich*; *TV.pl einander*)
  - absolute und relative Adjektive (*blaue CN*; *große CN*)
- einfache Sätze, mit
  - Vollverb und Nominalobjekten, im Aktiv und Passiv,
  - prädikatsmodifizierenden Adverbien (*schnell*; *absichtlich*)
  - Vergleichskonstruktionen (*TV* NP_1 *mehr als* NP_2 u.ä.)
- Satzkombinationen und -folgen, mit
  - Satzjunktoren (*wenn* S_1 *dann* S_2; S_1 *und* S_2)
  - Personal- und Possessivpronomen (*er/sie/es*; *sein/ihr*)
  - satzinterne und satzübergreifende Pronomenauflösung

Implementierung

In der Übung soll die besprochene Semantik implementiert werden.

- Grammatik: Erweiterung der Beispielgrammatik aus Computerlinguistische Anwendungen 2013
- Montague-Semantik: Erweiterung der Übersetzung von Syntaxbäumen in λ-Terme und logischen Formeln
- Diskurssemantik: Zur Pronomenauflösung wird eine Version der Diskursrepräsentationstheorie (λ-DRT) behandelt.
- Wenn möglich, weitere Themen wie
  - „continuation passing style“-Übersetzung (Kontextübergabe)
  - Semantik von Wissensoperatoren (für Objektsätze)
  - Spielsemantik (Ambiguitätsauflösung durch Kommunikation)

Literatur

- J.van Eijck, C.Unger: Computational Semantics with Functional Programming. Cambridge University Press, 2010
- H.Leiß: Computerlinguistische Anwendungen (Vorlesungsfolien und Programme), Universität München, CIS, 2013
- B.Bringert: Delimited Continuation, Applicative Functors and Natural Language Semantics. 2008

Ggf. weitere Titel

- R. Montague: The Proper Treatment of Quantification in Ordinary English. R.Thomason (Ed.): Formal Philosophy, Reidel 1974
- R.Dowty, R.E.Wall,S. Peters: Introduction to Montague Grammar. D.Reidel, 1978
- H.Leiß: Seminar: Interpretation und Auswertung. 2004/05

Übungsstunde 1

Welche Software? Alternativen sind:

1. SWI-Prolog mit DC-Grammatiken/Parser.
   + Lambda-Terme und Beispielgrammatik aus dem CLA-Kurs.
   + Grammatik leicht erweiterbar
     Neu: semantische Typsisierung, Typrekonstruktion, Diskursrepräsentationssemantik, Plural, Passiv u.a.
     Erweiterungen von Syntax und Semantik
2. SML und ML-Lex/HL-Yacc:
   + Modulsystem für Beispielmodell und Gliederung der Semantik
   + ML-Lex/HL-Yacc: Lexergenerator mit ambigen Token; Parsergenerator für kontextfreie Grammatiken (intern CYK)
   - Kategoriensymbole atomar (Merkmale); Lexikonerstellung umständlich

3. Haskell mit Grammatical Framework
   - \+ Mehrsprachige Grammatiken vorhanden, neue leicht definierbar
   - \+ Exportiert Datentypen nach Haskell
   - \+ Semantikausgabe in den Kategorien im Prinzip möglich
   - \- Keine Erfahrung mit Modellbauen in Haskell
4. Mercury mit DC-Grammatiken
   - \+ Prolog-artig mit Datentypen und DC-Grammatiken/Parser
   - \- nicht interaktiv
   - \- nicht getestet, ob man z.B. alles aus CLA machen kann (Grammatikübersetzung zur Baumausgabe)

Alternative 1: SWI-Prolog und DC-Grammatik

1. Eine Grammatik mit Baumausgabe als DCG schreiben:

⟨*Beispiel grammatik.pl*⟩≡

```
np(NP,[Gen,Num],[Kas])
--> det(DET,[],[Gen,Num,Kas]), n(N,[Gen],[Num,Kas]),
    { NP = [np([Gen,Num]),DET,N] }.
det(DET,[indef],[mask,sg,nom])
--> [ein], { DET = [det([indef]),[ein]] }.
n(N,[mask],[sg,nom])
--> ['Fehler'], { N = [n([mask]),['Fehler']] }.

numerus(sg). numerus(pl).
```

Oder: Baumausgabe durch Grammatikübersetzung erzeugen.

3. Semantikregeln zur Übersetzung der Syntaxbäume in Formeln:

⟨*Beispiel semantik.pl*⟩≡

```
sem([det([indef]),[ein]],
    lam(N,lam(P,ex(X, N*X & P*X)))).
sem([n(_),['Fehler']],
    lam(X, fehler(X))).
sem([np([Gen,Num]),DET,N],SemNP) :-
    sem(DET,SemDET),
    sem(N,SemN),
    SemNP = normalize(SemDet * SemN, SemNP).
normalize(LamTerm,Normalform) :-
    Normalform = LamTerm.        % der Kürze halber
```

4. Datenbank mit Fakten schreiben.

5. Auswertung von Formeln in der Datenbank schreiben.

Alternative 2: SML und ML-Lex/HL-Yacc-Parsergenerator

1. Konfigurationsdatei für den „Compilation Manager“ von SML-NJ

⟨*SML/makefile.cm*⟩≡

```
Library
  structure NL     (* Modulname *)
  structure NLVals (* Hilfsmodul *)
is
  $smlnj/compiler.cm
  $/basis.cm
  $/ml-glr-lib.cm

  nl.sml           (* finite model P and quantifiers Q *)
  nl.lex           (* lexer spec.  *)
  nl.grm:MLGLR     (* grammar spec.:Parsergenerator *)

  nl.interface.sml
  nl.parse.sml
```

2. Grammatik definieren im ML-Yacc/ML-GLR-Stil:

⟨*SML/nl.grm*⟩≡

```
(* User declarations *)
structure D = P (* finite model P of persons *)
datatype value = object of D.PN | truth of bool
fun printVal (truth(b)) = Bool.toString b
  | printVal (object(p)) = D.toString p

%% (* ML-GLR declarations *)
%name NL
%verbose
%lexheader (functor NLLexFun(structure Tokens: NL_TOKENS
                             structure Interface: INTERFACE)
                      : LEXER)
%term
     EOF | PUNKT
   | JEDER | EIN
   | n of Q.N | iv of Q.IV | pn of Q.PN (* cat of value *
```

⟨*SML/nl.grm*⟩+≡

```
%nonterm
   start of (value * string) option
   | s of D.S | np of D.NP | qnp of (D.PN -> D.S) -> D.S
%start start
%eop EOF PUNKT
%pos int

%% (* ML-GLR grammar rules *)

start : s              (SOME (truth(s),Bool.toString s))
      | np             (SOME (object(np), D.toString np))
      |                (NONE)
s     : np iv          (iv(np))
      | qnp iv         (qnp(fn x => iv(x)))
np : pn (pn)
qnp: JEDER n (Q.jeder n) | EIN n (Q.ein n)
```

3. Lexer spezifizieren im ML-Lex-Stil:

⟨*SML/nl.lex*⟩≡

```
structure T = Tokens
structure I = Interface
open I
type pos = I.pos
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) T.token
type lexresult = (svalue,pos) token list * string
fun eof() = (Tokens.EOF(!line,!line)::nil,"Stop")
structure D=Q
%%
%header (functor NLLexFun(structure Tokens: NL_TOKENS
                structure Interface: INTERFACE) : LEXER);
%full

comment=%(.*)\n;
whitespace=[\t\ ]*;
```

⟨SML/nl.lex⟩+≡

```
%%
<INITIAL>{whitespace} => (lex());
<INITIAL>{comment}    => (next_line(); lex());
<INITIAL>\n           => (next_line(); lex());
"."         => (T.PUNKT(!line,!line)::nil,yytext);
"stop"      => (Tokens.EOF(!line,!line)::nil,yytext);
"Anna"      => (T.pn(D.Anna,!line,!line)::nil,yytext);
"Emil"      => (T.pn(D.Emil,!line,!line)::nil,yytext);
"Mann"      => (T.n(D.Mann,!line,!line)::nil,yytext);
"Frau"      => (T.n(D.Frau,!line,!line)::nil,yytext);
"arbeitet" => (T.iv(D.arbeitet,!line,!line)::nil,yytext);
"jede"      => (T.JEDER(!line,!line)::nil,yytext);
"jeder"     => (T.JEDER(!line,!line)::nil,yytext);
"eine"      => (T.EIN(!line,!line)::nil,yytext);
"ein"       => (T.EIN(!line,!line)::nil,yytext);
.           => (error ("ignoring illegal character "
                       ^ yytext, !line,!line); lex());
```

4. Endliches Modell als SML-Modul definieren:

⟨*SML/nl.sml*⟩≡

```
signature INTERPRETATION =
  sig
    datatype person =
      a | b | c | d | e | f | g | h | i | j | k | l | m
    type PN
    type NP
    type S = bool
    type N
    type IV

    val Anna : PN
    val Emil : PN
    val Frau : N
    val Mann : N
    val arbeitet : IV
    val toString : person -> string
  end
```

⟨*SML/nl.sml*⟩+≡

```
structure P : INTERPRETATION =
  struct (* Wertebereiche der syntaktischen Kategorien: *
    datatype person =
       a | b | c | d | e | f | g | h | i | j | k | l | m
    type PN = person
    type NP = person
    type S = bool
    type N = person -> bool
    type IV = (person -> bool)
    val Anna = a  (* Werte der einfachen Ausdruecke: *)
    val Emil = e
    fun Frau( a | b | c | g | h | i ) = true | Frau(_) =
    fun Mann( d | e | f | j | k | l ) = true | Mann(_) =
    fun arbeitet( a | b | c | d | e ) = true
      | arbeitet(_) = false
    val toString = (fn a => "a" | b => "b" | c => "c"
                     | d => "d" | e => "e" | m => "m"
                       | _ => "some person")
  end
```

Erweiterung des Modells um Quantoren:

⟨*SML/nl.sml*⟩+≡

```
signature INTERPRETATION_Q =
  sig
      include INTERPRETATION
      type Quant
      val jeder : Quant
      val ein : Quant
  end

structure Q =
  struct
    open P  (* die Interpretation von NL benutzen *)

    type Quant = (person->bool)->(person->bool)->bool
    local val pers = [a,b,c,d,e,f,g,h,i,j,k,l,m]
    in fun jeder F G =  List.all G (List.filter F pers)
       fun ein F G  = List.exists G (List.filter F pers)
    end
  end
```

⟨*Hilfsdatei SML/nl.parse.sml*⟩≡

```
functor Parse (structure Interface : INTERFACE
               structure Parser : PARSER where type resul
                  sharing type Parser.arg = Interface.arg
                  sharing type Parser.pos = Interface.pos
               structure Tokens : NL_TOKENS
                  sharing type Tokens.token = Parser.Toke
                  sharing type Tokens.svalue = Parser.sva
               ) :
          sig
              val parse : unit -> unit
              val debug : bool ref
              val trees : bool ref
          end = ...
structure NL =
   Parse (structure Interface = Interface
          structure Parser = NLParser
          structure Tokens = NLVals.Tokens );
```

⟨*Hilfsdatei SML/nl.interface.sml*⟩≡

```
signature INTERFACE =
  sig
      type pos
      val line : pos ref
      val init_line : unit -> unit
      val next_line : unit -> unit
      val error : string * pos * pos -> unit

      type arg
      val nothing : arg
  end

structure Interface : INTERFACE =
  struct
    type pos = int
    val line = ref 0
    ...
  end
```

6. Alles zusammenbauen:

2. Vorlesung: Beispielgrammatik

Die Grammatik aus Computerlinguistische Anwendungen (2013).

Regelformat: Definite Clause Grammar

⟨*Regelformat* ⟩≡

```
Cat --> Cat1, ..., Catn, { Nebenbedingung }.
Cat --> Cat1, (Cat2 ; Cat3 ; []), .., Catn. % optionale
```

⟨*Kategorienformat* ⟩≡

```
CatName(Artmerkmale,Formmerkmale)
```

⟨*Merkmalbereiche* ⟩≡

```
Bereichsname(Wert1). ... Bereichsname(Wertn).
```

⟨*Nebenbedingung* ⟩≡

```
Prologprädikat(PrologTerm1,...,PrologTermn).
```

Nominalphrasen- und Satzregeln

Daher sehen wir uns die CLA-Grammatik an und ändern, was besser werden sollte.

⟨*Beispiel einer Regel*⟩≡

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

Artmerkmale: Definitheit, Person, Genus

Formmerkmale: Numerus, Kasus

Verbessern:

1. Numerus ist bei Nomen ein Formmerkmal, bei Nominalphrasen und Determinatoren aber nicht: diese können i.a. nicht im Numerus variiert werden. (Np1 und NP1), (viele N): nur Plural,
2. Definitheit bei Det ist zu grob, wenn die Auswertungsregel für (NP-> Det N) nur die Kategorie Det bekommt, nicht den ganzen Det-Teilbaum.

⟨*Grammatik/kategorien.pl*⟩≡

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

% --- Wertebereiche der Art- und Formmerkmale -------

person(X) :- (X = 1 ; X = 2 ; X = 3).
tempus(X) :- (X = praes ; X = praet).
modus(X) :- (X = ind ; X = konj).

genus(Gen) :-
    (Gen = fem ; Gen = mask ; Gen = neut).
numerus(X) :-
    (X = sg ; X = pl).
kasus(X) :-
    (X = nom ; X = gen ; X = dat ; X = akk).
```

Wir verwenden in der Nomialphrasengrammatik die Kategorien

- *Eigennamen*, *Nomen*, *Relationsnomen*,
- *Pronomen*, *Possessiv* (definit,relativierend,interrogativ)
- *Determinator* (Artikel und Quantor)
- *Nominalphrase* (in/definite, relativierende, interrogative)

⟨*Grammatik/kategorien.pl*⟩+≡

```
% --- Kategorie(Artmerkmale,Formmerkmale) ----------

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

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

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

⟨*Grammatik/kategorien.pl*⟩+≡

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

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

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

Alle Nominalphrasen variieren im Kasus, aber manche variieren nicht im Numerus, weshalb der zu den Artmerkmalen gezählt wird:

Das Genus hat nur im Singular Sinn; vgl. *die Leute*, Person und Numerus nur bei einfachen Nominalphrasen, vgl. *du und die Frauen*

⟨*Grammatik/kategorien.pl*⟩+≡

```
kategorie(np([Def,Pers,Gen,Num],[Kas])) :-
    definitheit(np,Def), % Num inhaerent
    Pers = 3, % TODO für Dialoge: 1.+2.Person
    genus(Gen), numerus(Num), kasus(Kas).

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

Wir verwenden in der Satzgrammatik die Kategorien

- *Verb* (intransitiv, transitiv, Hilfsverb)
- *Satz* (Aussage, Relativsatz, Fragesatz)

⟨*Grammatik/kategorien.pl*⟩+≡

```
kategorie(v([nom],[Pers,Num,Temp,Mod])) :-
    Pers = 3,       % TODO: 1.2 Person fuer Dialoge
    numerus(Num), tempus(Temp), modus(Mod). % Mod := ind

kategorie(v([nom,akk],[Pers,Num,Temp,Mod])) :-
    Pers = 3,       % TODO: 1.2 Person fuer Dialoge
    numerus(Num), tempus(Temp), modus(Mod). % Mod := ind

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

Die Merkmale haben i.a. nur bei einfachen Sätzen Sinn.

⟨*Grammatik/kategorien.pl*⟩+≡

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

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

% TODO: Kategorie komplexer Saetze: welche Merkmale?
% kategorie(s([Def],[])) :-  % koordinierte Saetze
```

Nominalphrasengrammatik

- Numerus ist bei NP und Det ein Artmerkmal, da Det i.a. festen Numerus hat
- TODO: Definitheit bei Det: def|qu + ex,all/every/each,many,most,different wieviele = wh+viele, wer = wh+indef
- Definitheit bei NP: nur noch [qu,def,rel(Gen,Num)] ?

⟨*Grammatik/np.pl*⟩≡

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

np([Def,3,Gen,Num],[Kas]) -->  % Pers 1,2: nicht im Lexik
    pron([DefP],[Gen,Num,Kas]),
    { DefP = rel, Def = rel(Gen,Num)
    ; DefP = qu, Def = qu
    ; DefP = per, Def = def }. % Personalpronomen
```

- Artikel + Nomen/Eigennamen + Relativsatz (im Indikativ)

⟨*Grammatik/np.pl*⟩+≡

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

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

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

- Nominalphrasen mit possessivem Pronomen oder Genitivobjekt bei einem Relationsnomen

⟨*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]) },
    ( [] ; s([rel(Gen,Num)],[_Temp,ind,vl]) ).

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

- Nominalphrasen im Plural, ohne Artikel

⟨*Grammatik/np.pl*⟩+≡

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

- Nominalphrasen aus Reflexiv- oder Reziprokpronomen:

⟨*Grammatik/np.pl*⟩+≡

```
np([Def,3,Gen,Num],[Kas]) -->
    pron([Def],[Gen,Num,Kas]),
    { (Def = refl ; Def = rezi) }.
```

(2. Übungsstunde) Satzgrammatik

Wir definieren hier Regeln für einfache Sätze, deren Prädikat

- ein transitives Vollverb ist, oder
- aus dem Hilfsverb *sein* mit einem Adjektiv gebildet wird, oder
- aus dem Hilfsverb *sein* mit einem Nomen gebildet wird.

Es werden jeweils drei Arten erlaubt:

- Aussagen = definite Sätze: `s([def],Form)`
- Fragen = interrogative Sätze: `s([qu],Form)`
- Relativsätze = relativierende Sätze `s([rel(Gen,Num)],Form)`

Im Tempus sind nur Präsens und Präteritum erlaubt.

- Prädikativsätze mit nominalem Prädikat: *x ist der/ein N*

⟨*Grammatik/saetze.pl*⟩≡

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

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

⟨*Grammatik/saetze.pl*⟩+≡

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

- Prädikativsätze mit adjektivischem Prädikat: *x ist AP*

⟨*Grammatik/saetze.pl*⟩+≡

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

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

⟨*Grammatik/saetze.pl*⟩+≡

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

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

Die einzigen APs sind Vergleichskonstruktionen, um Adjektive im Lexikon auf wenige Formen beschränken zu können.

- Sätze mit transitivem Vollverb und variabler Stellung des Subjekts: NP_1 *TV* NP_2

⟨*Grammatik/saetze.pl*⟩+≡

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

Wenn das Objekt ein Reflexiv- oder Reziprokpronomen ist, muß der Numerus eingeschränkt werden (hier bei `refl` zu stark).

⟨*Grammatik/saetze.pl*⟩+≡

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

⟨*Grammatik/saetze.pl*⟩+≡

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

Lexikon

Es gibt ein kleines Lexikon für Determinatoren und Pronomen:

⟨*Auszug aus Grammatik/lexikon_detpron.pl*⟩≡

```
det([indef],[mask,sg,nom]) --> [ein].
det([def],  [mask,sg,nom]) --> [der].
det([qu],   [mask,sg,nom]) --> [welcher].
det([quant],[mask,sg,nom]) --> [jeder].
...
poss([rel],[mask,sg,gen]) --> [dessen].
...
pron([qu], [mask,sg,nom]) --> [wer].
pron([qu], [mask,sg,akk]) --> [wen].
...
pron([rel],[mask,sg,nom]) --> [der].
...
pron([refl],[mask,pl,akk]) --> [sich].
pron([rezi],[mask,pl,akk]) --> [einander].
```

Und es gibt ein Lexikon für Inhaltswörter, das aber nur einen Zugriff auf ein Lexikon `wort/3` einer Anwendung liefert

⟨*Auszug aus Grammatik/lexikon_nomenverb.pl*⟩≡

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

sowie ein paar Einträge für das Hilfsverb *sein*.

Eine Anwendung muß ein Vollformenlexikon der Inhaltswörter und deren Bedeutung durch eine Datenbank bereitstellen. Wir benutzen

⟨*astronomie.pl*⟩≡

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

⟨*Auszug aus Anwendungen/Astronomie/vollformen.pl*⟩≡

```
% wort(Stammform,Kategorie(Art,Form),Vollform).

wort(entdecken,
     v([nom, akk], [1, sg, praes, ind]), entdecke).
...
wort('Stern', n([mask], [sg, nom]), 'Stern').
...
wort('Galilei', en([mask], [sg, nom]), 'Galilei').
...
```

Laden von Grammatik und Parser

Die Grammatik (*ohne* Anwendungslexikon) und der Tokenizer und Parser bestehen aus mehreren Dateien:

⟨*grammatik.pl*⟩≡

```
:- style_check(-discontiguous).

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

Eine kleine Bedienhilfe wird ebenfalls in `grammatik.pl` formuliert:

⟨*grammatik.pl*⟩+≡

```
bedienung :-
    w(0,'--------------- Bedienfunktionen ---------------
    w(0,'parse.  <Frage | Satz | Nominalphrase>. <Return>
    w(9,'Syntaxbaum als formatierter Text'),
    w(0,'parsed. <Frage | ..>. <Return>'),
    w(9,'Syntaxbaum graphisch (dot+gv; syntaxbaum.tmp.dot

w(N,Atom) :-
    write(user,'\n'),tab(user,N), write(user,Atom).
```

Die graphische Anzeige (vgl. `showTree.pl, dot.syntaxbaum.pl`) erfordern, daß `graphviz/dot` und `gv` (`ghostview`) installiert sind.

Laden von Grammatik und Anwendung

Um die Grammatik benutzen zu können, muß man `grammatik.pl` und eine Anwendung laden:

⟨*Laden der Grammatik und der Astronomie-Anwendung*⟩≡

```
?- [grammatik,astronomie].

?- bedienung.

---------------- Bedienfunktionen ---------------------
parse.  <Frage | Satz | Nominalphrase>. <Return>
         Syntaxbaum als formatierter Text
parsed. <Frage | ..>. <Return>
         Syntaxbaum graphisch (dot+gv; syntaxbaum.tmp.dot
true.
```

Parseaufrufe

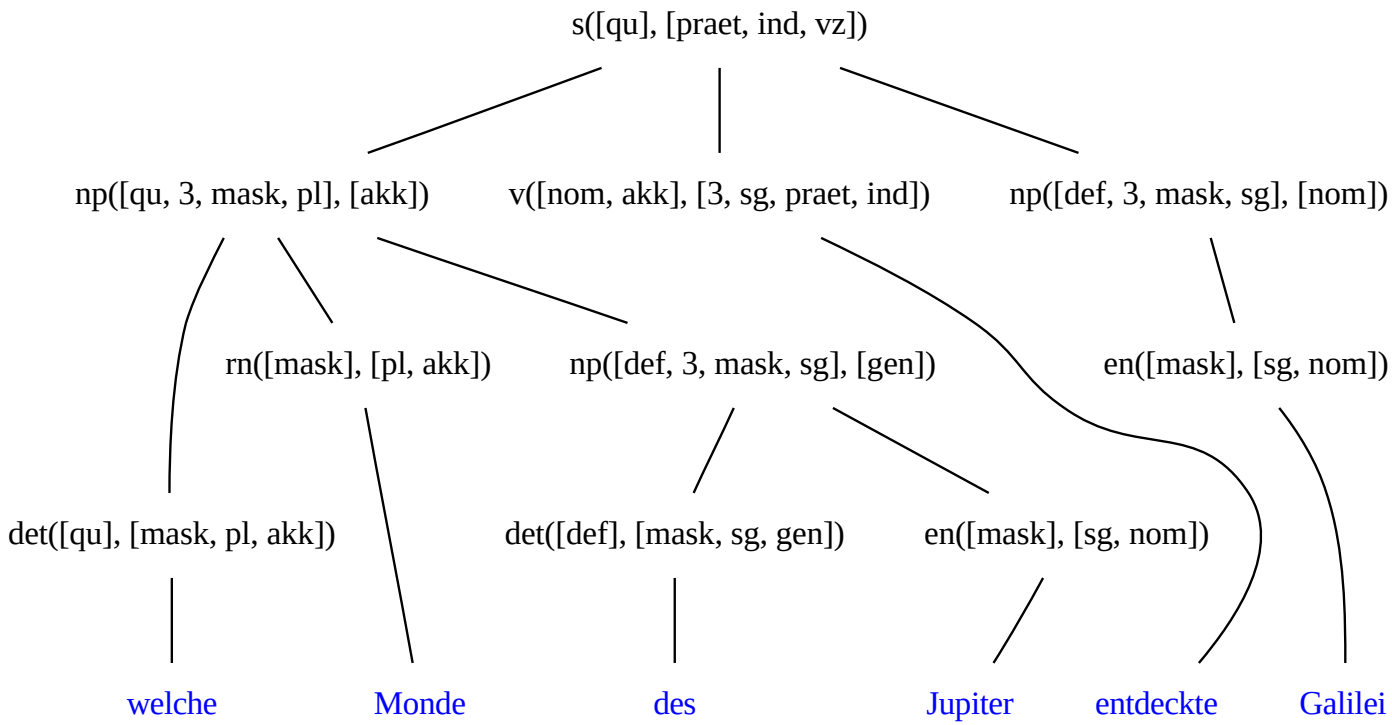
⟨*Parseaufruf mit textueller Baumdarstellung*⟩≡

```
?- parse.
Beende die Eingabe mit <Punkt><Return>
|: Galilei entdeckte einige Monde des Jupiter.

Aufruf: s([_], [_, ind, _], _, ['Galilei', entdeckte,
                einige, 'Monde', des, 'Jupiter'], []).
Baum:
        s([def], [praet, ind, vz])
           np([def, 3, mask, sg], [nom])
              en([mask], [sg, nom]) 'Galilei'
           v([nom, akk], [3, sg, praet, ind]) entdeckte
           np([indef, 3, mask, pl], [akk])
              det([indef], [mask, pl, akk]) einige
              rn([mask], [pl, akk]) 'Monde'
              np([def, 3, mask, sg], [gen])
                 det([def], [mask, sg, gen]) des
                 en([mask], [sg, nom]) 'Jupiter'
```

⟨*Parseaufruf mit graphischer Baumdarstellung*⟩≡

```
?- parsed.
Beende die Eingabe mit <Punkt><Return>
|: welche Monde des Jupiter entdeckte Galilei.
true
```



Die Syntaxbäume werden mit folgenden Regeln formatiert:

⟨*Parser/showTree.pl*⟩≡

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

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

Die Umwandlungen für die graphische Ausgabe wird hier nicht gezeigt. Siehe dazu: `Parser/dot.syntaxbaum.pl`

Regressionstest

Es ist nützlich, wenn man erkannte (oder noch zu erkennende) Beispiele in einer Datei sammelt.

⟨*Auszug aus Grammatik/testsaetze.txt*⟩≡

```
der Uranus ist ein Planet.
... % ein Ausdruck pro Zeile
```

Nach Änderungen der Grammatik kann man die Beispiele dann mit dem Prädikat `parse(+Dateiname)` aus dem Modul `tokenizer` der Datei `Parser/tokenizer.mini.pl` wieder analysieren:

⟨*Analyse von Beispielen aus einer Datei*⟩≡

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

[der, 'Uranus', ist, ein, 'Planet']

   s([def], [praes, ind, vz])
      np([def, 3, mask, sg], [nom])
      ...
...
```

3.Vorlesung: Formeln und λ-Terme

Wir wollen Bedeutungen durch logische Formeln ausdrücken.

⟨*Terme und Formeln*⟩≡

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

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

Satz $\alpha \mapsto$ Syntaxbaum $t_\alpha \mapsto$ Formel $\varphi_{t_\alpha}(x, \ldots)$

Die Berechnung der Bedeutung von Aussagen und Fragen soll in drei Schritten erfolgen:

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

Die Bedeutung $[\![\alpha]\!]$ des Satzes α ist der Wert $[\![\varphi_{t_\alpha}]\!]^{\mathcal{D}}$ der logischen Formel φ_{t_α}, die man dem Syntaxbaum t_α von α zuordnet, in einem endlichen Modell $\mathcal{D}$.

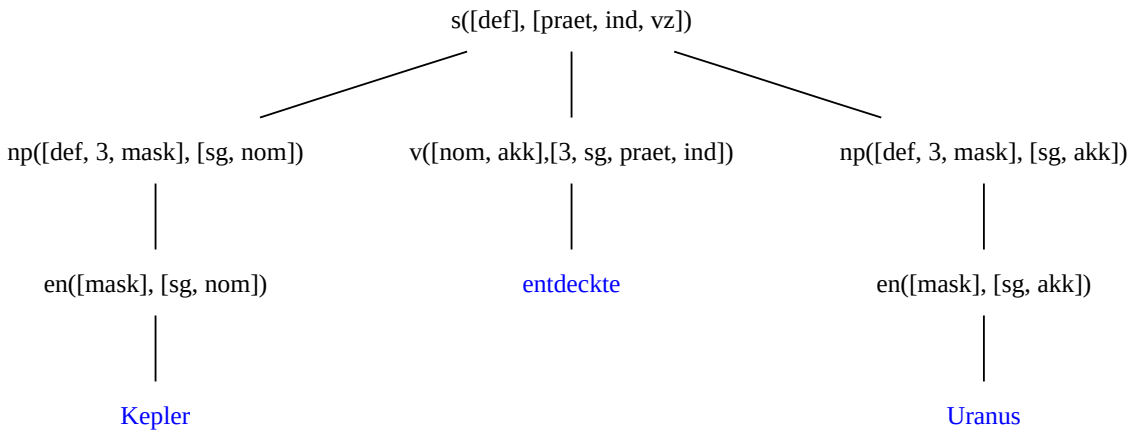
Ein endliches Modell $\mathcal{D} = (D, P^{\mathcal{D}}, \ldots, c^{\mathcal{D}}, \ldots)$ besteht aus einem endlichen Individuenbereich D und Interpretationen $P^{\mathcal{D}} \subseteq D^n$ der Prädikate und $c^{\mathcal{D}} \in D$ der Eigennamen, z.B. eine Datenbank.

Wie berechnet man aus dem Syntaxbaum „die“ passende Formel?

Atomare Aussage:

Aussage: `Kepler entdeckte Uranus.`

Baum:



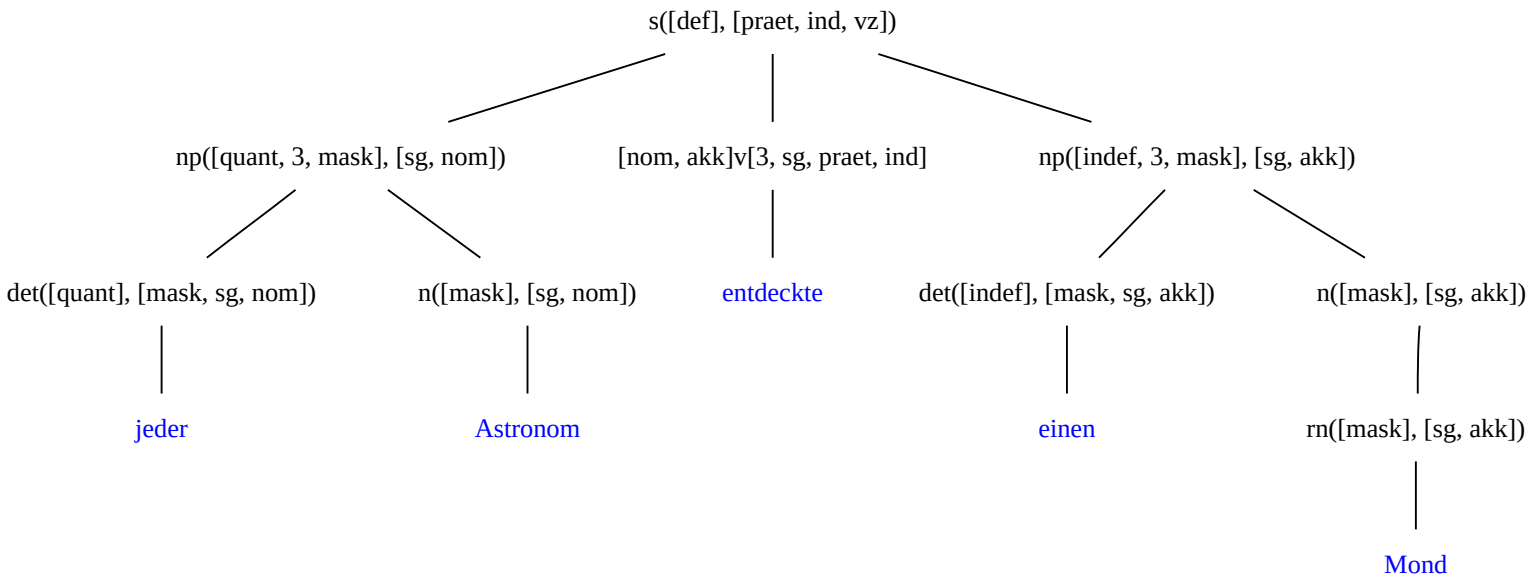
Formel: `entdecken(kepler,uranus).`

Dasselbe sollte bei anderen Wortstellungen herauskommen.

Quantifizierte Nominalphrasen:

Aussage: `Jeder Astronom entdeckte einen Mond.`

Baum:



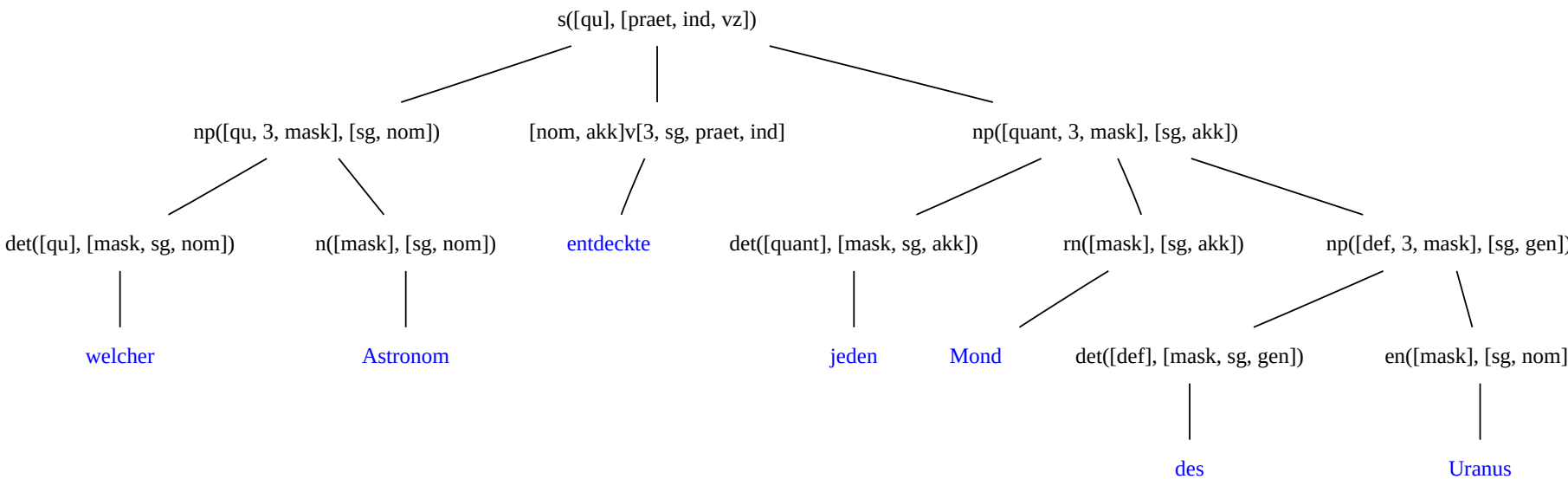
2 Formeln, da der Satz wegen der Quantorenskopi zweideutig ist:

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

W-Fragen und NP-Attribute:

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

Baum:



Formel: `qu(X, astronom(X) &`

`all(Y,mond(Y,uranus) => entdecken(X,Y))`

Was ist der schwierige Teil der Übersetzung?

Natürliche Sprache:

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

Prädikatenlogik (erster Stufe):

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

Adverbiale „oft/überall“ quantifizieren über Zeit/Orts-Individuen

Bedeutung einer quantifizierten NP?

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

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

Einfache Sätze sind nach A. eine Verbindung von Subjekt und Prädikat, (Chomsky: $S = NP\ VP$); die Quantoren im Subjekt “sind” Aussagemodi: allgemeine vs. partikuläre Aussageweise:

$$\begin{aligned}
(\textit{jeder } M \textit{ arbeitet}) &\mapsto M\ a\ A \equiv M \subseteq A,\\
(\textit{eín } M \textit{ arbeitet}) &\mapsto M\ i\ A \equiv M \cap A \neq \emptyset,\\
\textit{nicht (jeder } M \textit{ arbeitet}) &=\\
(\textit{ein } M \textit{ (arbeitet nicht)}) &\mapsto M\ o\ A \equiv M \not\subseteq A,\\
\textit{nicht (eín } M \textit{ arbeitet}) &=\\
(\textit{kein } M \textit{ arbeitet}) &\mapsto M\ e\ A \equiv M \cap A = \emptyset.
\end{aligned}$$

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

Heutige Auffassung: verallgemeinerte Quantoren Q in einer $NP = Q\,N$ drücken eine Beziehung zwischen der Nomenbedeutung N und der VP-Bedeutung aus:

$$[\![S]\!] = [\![NP\ VP]\!] = [\![Q\ N\ VP]\!] = [\![Q]\!]([\![N]\!], [\![VP]\!])$$

Quantoren vom Typ $\langle 1, 1\rangle$: die Bedeutung $Q^{\mathcal{I}}$ bei einer Interpretation $\mathcal{I}$ ist eine Beziehung zwischen Individuenmengen

$$\mathcal{I} \models Qx(\varphi, \psi) : \Longleftrightarrow\ Q^{\mathcal{I}}(\{\, x \mid \mathcal{I} \models \varphi \,\}, \{\, x \mid \mathcal{I} \models \psi \,\})$$

Beispiel: Aristotelische und weitere $\langle 1, 1\rangle$-Quantoren:

- $all(A, P) : \Longleftrightarrow\ A \subseteq P, \quad no(A, P) : \Longleftrightarrow\ A \cap P = \emptyset,$
- $most(A, P) : \Longleftrightarrow\ |A \cap P| > |A \setminus P|$

Beispiel $\langle 1, 1, 1\rangle$-Quantor: *Mehr Männer als Frauen lieben Fußball*

- $more\_than(A, B, P) : \Longleftrightarrow\ |A \cap P| > |B \cap P|$

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

Für einfache Aussagen (Verb mit Komplementen):

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

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

$$\begin{aligned}((\mathit{jeder}\ M)\ \mathit{singt}\ (\mathit{ein}\ L)) \quad &\mapsto \quad \forall x \in M\ (x\ \mathit{singt\ ein}\ L) \\ &\mapsto \quad \forall x \in M\,(\exists y \in L\ (x\ \mathit{singt}\ y))\end{aligned}$$

Mathematisch ausgedrückt:

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

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

Daher wollen wir übersetzen:

$$\begin{aligned} (\textit{alle}\ N) &\mapsto (P \mapsto \forall x(N(x) \rightarrow P(x))) \\ (\textit{ein}\ N) &\mapsto (P \mapsto \exists x(N(x) \wedge P(x))) \\ (\textit{die meisten}\ N) &\mapsto (P \mapsto \mathit{Most}\ x(N(x), P(x))) \end{aligned}$$

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

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

λ-Terme

λ-Terme sind eine Schreibweise für Funktionen und Daten:

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

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

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

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

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

Termvereinfachung $s \to t$

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

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

Weitere Regeln legen den Umgang mit Konstanten c fest.

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

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.

Syntaktische Einsetzung $t[x/s]$

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

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

Die *Ersetzung der freien Vorkommen von x in t durch s*, kurz: $t[x/s]$, definiert man induktiv über den Aufbau von t:

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

Beispiele für $t[x/s]$

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

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: in

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

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

Die „Vereinfachung“ kann sogar divergieren!

Aber wenn man Variablen mit *Typen*

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

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:

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

Quantoren vom Typ $\langle 1, 1 \rangle$ werden auf ihre Argumente *nacheinander* angewendet (nach H.B.Curry): $(Q \cdot N) \cdot P$ statt $Q(N, P)$.

Reduktion von λ-Termen

Wir vereinfachen λ-Termen nach folgender Reduktionsstrategie:

1. Versuche einen Reduktionsschritt, $t \to s$ (mit `beta/3` unten), und wenn das ging, reduziere s weiter; sonst ist t das Ergebnis (Normalform).
2. Im Reduktionsschritt wird eine Anwendung $t \cdot s$ der Form $\lambda x\, r \cdot s$ zu $r[x/s]$ vereinfacht; bei anderen Anwendungen wird möglichst t, sonst 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 in neue umbenannt (α-Reduktion), bevor x durch s ersetzt wird.

⟨*Semantik/lambdaTerme.pl*⟩≡

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

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

normalize(T,Nf) :-
    beta(T,ConT,Tag),     % Einen Reduktionsschritt
    ( Tag = chd           % machen; falls er zu ConT=/=T
    -> normalize(ConT,Nf) % führt, weiter vereinfachen,
    ; Nf = ConT ).        % sonst ist T die Normalform
                          % ConT für lam unter Quantor
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(T*S, T*ConS, Tag) :-
    var(T),
    !, beta(S,ConS,Tag).

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

⟨*Semantik/lambdaTerme.pl*⟩+≡

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

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

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

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

α-Reduktion: Umbenennung gebundener Variablen

⟨*Semantik/lambdaTerme.pl*⟩+≡

```
% Alpha-Reduktion: alpha(+Term,-Umbenannt),
%                   alpha(+Term,+VarPaare,-Umbenannt)
alpha(T, TDup) :-
    alpha(T, [], TDup), !.

alpha(V, L, VDup) :-
    var(V), !, rename(V,L,VDup).
alpha(lam(V,R), L, lam(New,RDup)) :-
    var(V), !, alpha(R,[(V,New)|L],RDup).
alpha(ex(V,R,S), L, ex(New,RDup,SDup)) :-
    var(V), !, alpha_seq([R,S],[(V,New)|L],[RDup,SDup]).
alpha(all(V,R,S), L, all(New,RDup,SDup)) :-
    var(V), !, alpha_seq([R,S],[(V,New)|L],[RDup,SDup]).
alpha(qu(V,R,S), L, qu(New,RDup,SDup)) :-
    var(V), !, alpha_seq([R,S],[(V,New)|L],[RDup,SDup]).
```

⟨*Semantik/lambdaTerme.pl*⟩+≡

```
alpha(T*S, L, TDup*SDup) :-
    !, alpha(T,L,TDup),
    alpha(S,L,SDup).
alpha(C, _, C) :-
    atomic(C), !.
```

⟨*Umbenennung von Paar-Quantoren*⟩

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

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

Umbenennung von Variablen

⟨*Semantik/lambdaTerme.pl*⟩+≡

```
% rename(+Var, +Variablenpaare, -VarUmbenannt)

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

Beispiele zur α- und β-Reduktion

Nur die λ-gebundenen Variablenvorkommen werden umbenannt:

⟨*Beispiel zur α-Reduktion*⟩≡

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

Vereinfachung durch Einsetzungen für gebundene Variable:

⟨*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)
```

⟨*Korrekte Auswertung, statt t[X/s] durch Unifikation X = s(X,Y)*⟩≡

```
?- normalize(lam(X,t(X,Y))*s(X,Y),Q).
Q = t(s(X,Y),Y)
```

Wenn in t die Variable x im Wirkungsbereich einer Bindung $\lambda y(\ldots x \ldots)$ steht und bei $t[x/s]$ solche y frei in s vorkommen, muß vor der Einsetzung die Bindungsvariable y in t in eine unbenutzte Variable umbenannt werden. Das Programm nennt einfach *alle* gebundenen Variablen in t um.

⟨*Beispiel zur Umbenennung gebundener Variablen*⟩≡

```
?- normalize(lam(X, X * lam(Y, X * Y)) * (Y * c), Nf).
Nf = Y*c*lam(_G343, Y*c*_G343).

?- normalize(lam(X, X * lam(Y, X * Y)) * lam(Z, Y * Z),
             Nf).
Nf = Y*lam(_G361, Y*_G361).
```

Berechnung der Semantik

Wir wollen zu jeder natürlichsprachlichen Aussage eine logische Formel berechnen, die ihre Bedeutung repräsentiert. Dazu verwenden wir den Syntaxbaum und berechnen

- zu jedem Blatt des Syntaxbaums eine Bedeutung,
- zu jedem inneren Knoten des Syntax eine Bedeutung aus den Bedeutungen an den Wurzeln der direkten Teilbäume („Kompositionsprinzip“)

Eigennamen werden in Konstante c übersetzt, Nomina und Verben in Grundprädikate $\lambda x.\, n(X)$ und $\lambda x \lambda y.\, p(X, Y)$, und Artikel und Quantoren in λ-Terme, die aus Prädikaten Formeln aufbauen.

Beispiel: Semantik zu einem NP-Baum

Der Aufbau der Formel wird *nach* der Syntaxanalyse durch `sem(+Syntaxbaum,-LamTerm)` berechnet.

Zum Syntaxbaum von *jeder Astronom* muß ein λ-Term aus den λ-Termen der Teilbäume konstruiert werden:

⟨*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))).
```

Die Bedeutung hängt i.a. nicht von den Formmerkmalen ab.

⟨*Beispiel einer Analyse (mit lesbaren Variablen)*⟩≡

```
?- ['Beispiele/semNP.pl','Semantik/lambdaTerme.pl'].

?- Baum = [np([quant,3,mask],[sg,nom]),
            [det([quant],[mask,sg,nom]), [jeder]],
            [n([mask],[sg,nom]), ['Astronom']]],
   sem(Baum,SemTerm),
   normalize(SemTerm,Normalform).

SemTerm = lam(N, lam(P, all(X, N*X=>P*X)))
          * lam(Z, astronom(Z))
Normalform = lam(P, all(X, astronom(X)=>P*X))
```

Die Bildung der Normalform kann auch in sem/2 erfolgen.

Übersetzung NL nach PL

Wir brauchen zuerst die Operatordeklarationen für die logischen Junktoren und eine Hilfsfunktion:

⟨*Semantik/sem.pl*⟩≡

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

kleinschreibung(Wort,Klein) :-
    downcase_atom(Wort,Klein).
```

Alle Quantoren werden mit Beschränkungsbereich formuliert, als 3-stellige Prädikate `Quantor(Variable,Beschränkung,Formel)`.

A. Bedeutung der Wörter

Von der Vollform eines Worts im Syntaxbaum gehen wir zur Stammform und von dort zu seiner Bedeutung:

1. Eigennamen bedeuten entsprechende Konstanten der Datenbank:

   ⟨*Semantik/sem.pl*⟩+≡

```
sem([en(Art,Form), [Vollform]],LamTerm) :-
    wort(Stammform, en(Art,Form), Vollform),
    kleinschreibung(Stammform,LamTerm).
```

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

⟨*Semantik/sem.pl*⟩+≡

```
sem([rn(Art,Form), [Vollform]],LamTerm) :-
    wort(Stammform, rn(Art,Form), Vollform),
    kleinschreibung(Stammform,Stamm),
    Formel =.. [Stamm,X,Y],
    LamTerm = lam(X,lam(Y,Formel)).

sem([v([nom,akk],Form),[Vollform]],LamTerm) :-
    kleinschreibung(Vollform,VollformKl),
    wort(Stammform,v([nom,akk],Form),VollformKl),
    Formel =.. [Stammform,X,Y],
    LamTerm = lam(X,lam(Y,Formel)).
```

3. Absolute Nomen bedeuten Eigenschaften von Objekten:

⟨*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).
```

Werden sie aus Relationsnomen abgeleitet, bedeuten sie die Projektion des Relationsnomens: $N(X) = \exists Y.RN(X, Y)$

⟨*Semantik/sem.pl*⟩+≡

```
sem([n(Art,Form),[rn(Art,Form),[Vollform]]],LamTerm)
    sem([rn(Art,Form),[Vollform]],SemRN),
    normalize(SemRN * X * Y, Formel),
    LamTerm = lam(X,ex(Y,true,Formel)).
```

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

$$\begin{aligned} \textit{ein } N\ VP &= (N \cap VP \neq \emptyset) \\ \textit{der } N\ VP &= (|N| = 1 \wedge N \cap VP \neq \emptyset) \\ \textit{jeder } N\ VP &= (N \subseteq VP) \end{aligned}$$

Ihre λ-Terme bilden aus 2 Eigenschaften die jeweilige Formel:

⟨*Semantik/sem.pl*⟩+≡

```
sem([det([indef],[_,sg,_]),_Vollform],
    lam(N,lam(P,ex(X, N*X, P*X)))).      % ex/3

sem([det([def], [_,sg,_]),_Vollform],
    lam(N,lam(P, iota(X, N*X, P*X)))).   % iota/3

sem([det([quant],[_,sg,_]),[Vollform]],
    lam(N,lam(P,all(X, N*X, P*X)))) :-   % all/3
    concat_atom(['jede'|_],Vollform).
```

5. Zahlquantoren („Mindestens k“, für Zahlen $k \geq 2$)

$$k\ N\ VP \quad = \quad (|N \cap VP| \geq k)$$

werden durch Auszählen (mit `anzahl/2`) ausgewertet:

⟨*Semantik/sem.pl*⟩+≡

```
sem([det([def],[_,pl,_]),[Vollform]],
    lam(N,lam(P, anzahl(X,N*X & P*X,Zahl)))) :-
    anzahl(Vollform,Zahl), 2 =< Zahl, !.
```

6. 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)))) :-
    concat_atom(['welche'|_],Vollform).
```

7. Artikel und Quantoren im Plural brauchen drei Bedeutungen:
   a) eine für distributive Prädikate über Individuen, *die meisten Menschen arbeiten*
   b) eine für distributive Prädikate über Individuenpaare (X, Y), *die meisten Geraden schneiden einander* (vereinfacht um $X \neq Y$):
   c) und eine wie b), aber mit Beschränkung durch ein Relationsnomen: *die meisten Brüder gleichen einander*.

   Wir formulieren a) und lassen die beiden anderen Fälle für später (mit $\langle Plural \rangle$), wenn die λ-Terme typisiert werden.

   Erst mit den Typen können wir unterscheiden, ob bei `lam(N,lam(P,Formel))` mit `N` und `P` über ein- oder zweistellige Prädikate abstrahiert wird.

Der Numerus beeinflußt i.a. die Bedeutung einer NP, je nachdem, mit welcher Art Prädikat sie verbunden wird.

Bedeutung b), c) ist für *symmetrische Prädikate* gedacht:

- *Emil und Maria heiraten (einander).*
- *Sie stellt die Gäste einander vor.*

Es fehlt eine Bedeutung für *kollektive Prädikate*:

- *Die Abgeordneten beschließen das Gesetz.*

Distributive individuelle Lesart a) von *die*, *alle* und *die meisten*:

⟨*Semantik/sem.pl*⟩+≡

```
sem([det([def], [_,pl,_]),_Vollform],
    lam(N,lam(P, iota(X, N*X, P*X)))).  % iota/3

sem([det([quant],[_,pl,_]),[Vollform]],
    lam(N,lam(P,all(X,N*X,P*X)))) :-
    concat_atom(['alle'|_],Vollform).

sem([det([quant],[_,pl,_]),[_DefArt],[meisten]],
    lam(N,lam(P,most(X,N*X,P*X)))).
```

Distributive Paar-Lesart b), c) von *alle* und *die meisten*:

⟨*Semantik: Plural-Determiner*⟩≡

```
sem([det([quant],[_,pl,_]),[Vollform]],
    lam(N,lam(P,all((X,Y),N*X & N*Y,P*X*Y)))) :-
    concat_atom(['alle'|_],Vollform).

sem([det([quant],[_,pl,_]),[Vollform]],
    lam(N,lam(P,all((X,Y),N*X*Y,P*X*Y)))) :-
    concat_atom(['alle'|_],Vollform).

sem([det([quant],[_,pl,_]),[_DefArt],[meisten]],
    lam(N,lam(P,most((X,Y),N*X & N*Y,P*X*Y)))).

sem([det([quant],[_,pl,_]),[_DefArt],[meisten]],
    lam(N,lam(P,most((X,Y),N*X*Y,P*X*Y)))).

sem([det([quant],[_,pl,_]),[die]],
    lam(N,lam(P,iota((X,Y),N*X & N*Y,P*X*Y)))).
```

Distributive Paar-Lesart b), c) von Interrogativartikel *welche*:

⟨*Semantik: Plural-Determiner*⟩+≡

```
sem([det([qu],[_,pl,_]),[welche]],
    lam(N,lam(P,qu((X,Y),N*X & N*Y,P*X*Y)))).

sem([det([qu],[_,pl,_]),[welche]],
    lam(N,lam(P,qu((X,Y),N*X*Y,P*X*Y)))).
```

8. 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)))).
sem([a([als(nom)],[komp]),[größer]],
    lam(X,lam(Y,(Y < X)))).
```

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

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

⟨*Semantik/sem.pl*⟩+≡

```
sem([pron([rel],_Form),_Rel], lam(P,lam(X,P*X))).
sem([pron([qu],_Form),[Wer]], lam(P,qu(X,true,P*X)))
     ( Wer = 'wer' ; Wer = wen ).
```

10. Relativierende Possessivpronomen bedeuten auf ein korreliertes Individuum „verschobene“ Eigenschaften:

⟨*Semantik/sem.pl*⟩+≡

```
sem([poss([rel],[_Gen,_Num,gen]),_Poss],
     lam(RN,lam(P,lam(Y,ex(X, RN*X*Y, P*X))))).
     % nur bei RNsg
```

Z.B.: *dessen Mond aufgeht* = λy(*der Mond von y geht auf*)

$$P(\textit{dessen } RN) \mapsto \lambda y \exists x(RN(x, y) \wedge P(x))$$

11. Reflexiv- und Rexiprokpronomen reduzieren die Stelligkeit von Prädikaten: (erst noch in die Grammatik einbauen)

12. Hilfsverben `v([sein],Form)` haben keine eigenständige (nur „synkategorematische“) Bedeutung; keine Klausel.

Laden der Grammatik und Semantik

Mit `?- [semantik,astronomie]` werden die nötigen Dateien geladen.

⟨*semantik.pl*⟩≡

```
:- style_check(-discontiguous).
:- [grammatik],
   ['Semantik/lambdaTerme.pl',
    'Semantik/sem.pl'
    ].
```

Die `sem/2`-Klauseln für zusammengesetzte Ausdrücke folgen unten.

Zum Testen dient parses, ein Parseraufruf mit Ausgabe der aus dem Syntaxbaum berechneten Lambda-Terme:

⟨*semantik.pl*⟩+≡

```
parses :-
    write('Beende die Eingabe mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    startsymbol(S),
    addTree:translate(S,Term,[Baum]),
    addDifflists:translate(Term,ExpTerm,Atomlist,[]),
    nl,write('Aufruf: '), portray_clause(ExpTerm),
    call(ExpTerm), write('Baum:   '),
    nl,writeTrLam(Baum,4),nl,
    fail.
parses.
```

Die Dateien 'semantik.pl','astronomie.pl' seien geladen.

Ausgabe des Baums mit den Lambda-Termen

Mit `writeTrLam` schreiben wir den Syntaxbaum formatiert und dabei unter die Knoten die entsprechenden Lambda-Terme:

⟨*Parser/showTree.pl*⟩+≡

```
/* nur erste Lesart:
writeTrLam([Wurzel|Bs],Einrueckung) :-
    (sem([Wurzel|Bs],LamTerm) % 2.sem.Lesarten?
     -> (debugging(typisieren)
         -> type([],LamTerm,LamTermGetypt,Typ),
             Term = (LamTermGetypt:Typ)
         ; Term=LamTerm),
         tab(Einrueckung),writeq(Wurzel),
         nl,tab(Einrueckung),write('+ '),writen(Term)
    ; tab(Einrueckung),writeq(Wurzel)), % Wörter ohne Bed
    (Bs = [[Blatt]]
    -> tab(1), writeq(Blatt)
    ; Einrueckung2 is Einrueckung + 3,
      writeTrsLam(Bs,Einrueckung2)
    ). */
```

Ausgabe des Baums mit den Lambda-Termen

Mit `writeTrLam` schreiben wir den Syntaxbaum formatiert und dabei unter die Knoten die entsprechenden Lambda-Terme:

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLam([Wurzel|Bs],Einrueckung) :-
    ( sem([Wurzel|Bs],_)  % Ausdruck mit Bedeutungen
    ->  % schreibe alle Bedeutungen unter die Wurzel
        tab(Einrueckung),writeq(Wurzel),
        (Bs = [[Blatt]] -> tab(1), writeq(Blatt) ; true),
        (sem([Wurzel|Bs],LamTerm),
            (nl,tab(Einrueckung),write('+ '),writen(LamTe
            fail
        ; true)
    ; % schreibe das bedeutungslose Wort
      tab(Einrueckung),writeq(Wurzel) ),
    ( Bs = [[Blatt]] -> true % schon geschrieben
    ; Einrueckung2 is Einrueckung + 3,
      writeTrsLam(Bs,Einrueckung2) ).
```

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLam([],Einrueckung) :-
    tab(Einrueckung), writeq([]).

writeTrsLam([Baum|Baeume],Einrueckung) :-
    nl,writeTrLam(Baum,Einrueckung),
    writeTrsLam(Baeume,Einrueckung).
writeTrsLam([],_).
```

Variable schreiben wir lesbar als `X,Y,Z,A1,B1,...` durch:

⟨*Parser/showTree.pl*⟩+≡

```
writen(Term) :- \+ \+((numbervars(Term,23,_),
            write_term(Term,[numbervars(true)]))).
```

Beispiel: Syntaxbaum mit Semantik

Mit den unten folgenden Klauseln von sem/2 wird dann die Analyse mit semantischer Information so ausgegeben:

⟨*Beispiel einer Analyse*⟩≡

```
?- [semantik,astronomie].
?- parses. jeder Astronom.

Aufruf: np([A,B,C],[D,E],F,[jeder,'Astronom'],[]).
Baum:
    np([quant,3,mask],[sg,nom])
    + lam(X, all(Y, astronom(Y), X*Y))
       det([quant],[mask,sg,nom])
       + lam(X, lam(Y, all(Z, X*Z, Y*Z))) jeder
       n([mask],[sg,nom])
       + lam(X, astronom(X)) 'Astronom'
```

Für jede Startkategorie wird eine Analyse versucht, und wenn die gelingt, wird dazu die (erste) Bedeutung berechnet.

B. Bedeutung zusammengesetzter Ausdrücke

Kompositionsprinzip: Die Bedeutung eines zusammengesetzten Ausdrucks hängt vom Syntaxbaum B und den Bedeutungen der direkten Teilausdrücke ab und wird durch einen λ-Term angegeben.

Diesen λ-Term t (ggf. mehrere) berechnet man *rekursiv*:

1. bestimme die Verzweigungsform R an der Wurzel von B,
2. berechne die λ-Terme t_i seiner direkten Teilbäume B_i, für $i = 1, \ldots, n$,
3. konstruiere aus $t_1, \ldots, t_n$ den λ-Term t für B je nach R.

Die Bedeutung wird von den Blättern zur Wurzel des Baums berechnet.

Sie ist daher nur von der Form, nicht vom Kontext des Ausdrucks abhängig; die λ abstrahieren u.a. vom Kontext (in einem Satz).

Wir definieren nun `sem(+Syntaxbaum,-LamTerm)` nach der Form

```
[Kategorie(Art,Form), Teilbaum1, ..., TeilbaumN]
```

des Syntaxbaums des zusammengesetzten Ausdrucks:

1. Nominalphrasen (Kopf `n`, nicht `rn`) ohne Relativsatz:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([_,3,_,_],[_]),
        Det, [n(Art,Form),Vf]], SemNP) :-
    Vf \= km, sem(Det,SemDet),
    sem([n(Art,Form),Vf],SemN),
    normalize(SemDet * SemN,SemNP).
```

⟨*Beispiel* ⟩≡

```
?- parses. jeder Astronom.
lam(X, all(Y, astronom(Y), X*Y))
?- parses. die Sonne.
lam(X, iota(Y, sonne(Y), X*Y))
```

Im Spezialfall, wo das Nomen aus einem Relationsnomen abgeleitet wurde (mit `n --> rn` aus `lexikon_nomenverb.pl`, können wir die Bedeutung des eingebetteten Relationsnomens so benutzen:

⟨*Semantik: Plural*⟩≡

```
sem([np([_,3,_,pl],[Kas]),
        Det, [n(Art,Form),[rn(Art,Form),Vf]]], SemNP) :-
    Form = [pl,Kas], % nur für Plural
    sem(Det,SemDet),
    sem([rn(Art,Form),Vf],SemRN),
    normalize(SemDet * SemRN,SemNP).
```

⟨*Beispiel* ⟩+≡

```
?- parses. die meisten Planeten.
Baum:
    np([quant, 3, mask, pl], [nom])
    + lam(X, most(Y, lam(Z, planet(Y, Z)), X*Y))
       det([quant], [mask, pl, nom])
       + lam(X, lam(Y, most(Z, X*Z, Y*Z)))
          die
          meisten
       n([mask], [pl, nom])
       + lam(X, ex(Y, planet(X, Y)))
          rn([mask], [pl, nom])
          + lam(X, lam(Y, planet(X, Y))) 'Planeten'

?- parses. welche Sterne.
lam(X, qu(Y, stern(Y), X*Y))
```

2. Eigennamen als Nominalphrase bedeuten „die Anwendung von Prädikaten auf die Namenskonstante“:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([def,3,Gen,Num],[Kas]),
        [en([Gen],[Num,Kas]),EN]], SemNP) :-
     sem([en([Gen],[Num,Kas]),EN],SemEN),
     SemNP = lam(P,P*SemEN).
```

⟨*Beispiel* ⟩+≡

```
?- parses. Kepler.
np([def, 3, mask,sg], [nom])
+ lam(X, X*kepler)
   en([mask], [sg, nom])
   + kepler 'Kepler'
```

Ebenso für Eigennamen mit bestimmtem Artikel:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([def,3,Gen,Num],[Kas]),
      [det([def],[Gen,Num,Kas]),_],
      [en([Gen2],[Num,nom]),EN]], SemNP) :-
     sem([en([Gen2],[Num,nom]),EN], SemEN),
     SemNP = lam(P, P*SemEN).
```

⟨*Beispiel* ⟩+≡

```
?- parses. des Uranus.
lam(X, X*uranus)
```

3. Relativpronomen als Nominalphrasen: Ein Relativsatz drückt die *Eigenschaft* aus, die bei Abstraktion vom Relativpronomen entsteht; das Relativpronomen als NP verändert sie nicht:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np(_Art,_Form),[pron([rel],Form),VF]],SemNP) :-
    sem([pron([rel],Form),VF],SemNP).

sem([np(_Art,_Form),[pron([qu],Form),VF]],SemNP) :-
    sem([pron([qu],Form),VF],SemNP).
```

⟨*Beispiel* ⟩+≡

```
parses. der.
Baum:
    np([rel(mask, sg), 3, mask, sg], [nom])
    + lam(X, lam(Y, X*Y))
       pron([rel], [mask, sg, nom]) der
       + lam(X, lam(Y, X*Y))
```

4. Nominalphrase mit Genitivattribut:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([Def,3,Gen,Num],[Kas]),
      [det([Def],[Gen,Num,Kas])|Det],
      [rn([Gen],[Num,Kas]),RN],
      [np(Art,[gen])|Const]],SemNP) :-
    sem([det([Def],[Gen,Num,Kas])|Det],SemDet),
    sem([rn([Gen],[Num,Kas]),RN],SemRN),
    sem([np(Art,[gen])|Const],SemNPGen),
    normalize(SemDet *
       lam(X, SemNPGen*lam(Y,(SemRN*X*Y))), SemNP).
```

⟨*Beispiel* ⟩+≡

```
?- parses. die meisten Monde des Uranus.
Baum:
    np([quant, 3, mask, pl], [nom])
    + lam(X, most(Y, mond(Y, uranus), X*Y))
       det([quant], [mask, pl, nom])
       + lam(X, lam(Y, most(Z, X*Z, Y*Z)))
          die
          meisten
       rn([mask], [pl, nom]) 'Monde'
       + lam(X, lam(Y, mond(X, Y)))
       np([def, 3, mask, sg], [gen])
       + lam(X, X*uranus)
          det([def], [mask, sg, gen]) des
          + lam(X, lam(Y, iota(Z, X*Z, Y*Z)))
          en([mask], [sg, nom]) 'Uranus'
          + uranus
```

Es fehlen noch die die Bedeutungen für Nominalphrasen

- im Plural ohne Artikel und Quantor (*Monde des Jupiter*)
- aus Artikel, Nomen und Eigennamen (*der Mond Io*)
- aus reinem Interrogativpronomen (*wer*, *was*)
- aus Reflexiv- oder Reziprokpronomen (*sich*, *einander*)
- mit Relativsätzen (*die Monde, die Galilei entdeckte,*)

Übungsaufgabe: Ergänze die Bedeutungen für die ersten beiden Fälle und teste sie an Beispielen.

Bedeutung von Sätzen

7. Sätze mit Vollverb in vz, Vorfeld-NP mit weitem Skopus:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([_Def],[_,_,vz]),
       NP1, [v([nom,akk],[3,Num,_,ind]),V], NP2],
     SemS)
:-   not(debugging(typisieren)), % ohne Typen!
     sem(NP1,SemNP1), sem(NP2,SemNP2),
     sem([v([nom,akk],[3,Num,_,ind]),V], SemV),
     NP1 = [np([_,3,_,Num1],[Kas1])|_Konst],
     (Kas1 = nom, Num1 = Num,
         Sem = SemNP1 * lam(X,SemNP2 *
                            lam(Y,SemV * X * Y))
     ; Kas1 = akk,
         Sem = SemNP1 * lam(Y,SemNP2 *
                            lam(X,SemV * X * Y))
     ), normalize(Sem,SemS).
```

⟨ *Beispiel* ⟩+≡

```
parses. Galilei entdeckte einen Stern.
Baum:
    s([def], [praet, ind, vz])
    + ex(X, stern(X), entdecken(galilei, X))
       np([def, 3, mask, sg], [nom])
       + lam(X, X*galilei)
          en([mask], [sg, nom]) 'Galilei'
          + galilei
       v([nom, akk], [3, sg, praet, ind]) entdeckte
       + lam(X, lam(Y, entdecken(X, Y)))
       np([indef, 3, mask, sg], [akk])
       + lam(X, ex(Y, stern(Y), X*Y))
          det([indef], [mask, sg, akk]) einen
          + lam(X, lam(Y, ex(Z, X*Z, Y*Z)))
          n([mask], [sg, akk]) 'Stern'
          + lam(X, stern(X))
```

- Die NP im Vorfeld erhält weiten Wirkungsbereich; die Stellung der NPs wirkt sich also auf die Bedeutung aus:

⟨*Beispiel:* ⟩≡

```
?- parses. jeder Astronom entdeckte einen Stern.
+ all(X, astronom(X), ex(Y, stern(Y),
                          entdecken(X, Y)))

?- parses. einen Stern entdeckte jeder Astronom.
+ ex(X, stern(X), all(Y, astronom(Y),
                        entdecken(Y, X)))
```

- Das ist richtig für Fragen (Aussagen enthalten keine Fragen!)

⟨*Beispiel* ⟩+≡

```
?- parses. welcher Astronom entdeckte einen Mond.
+ qu(X, astronom(X), ex(Y, ex(Z, true, mond(Y, Z)),
                          entdecken(X, Y)))
```

8. Relativsätze: die Bedeutung wird wie bei Aussagen berechnet, nur entsteht durch das Relativpronomen eine Eigenschaft:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([rel(_Gen,Num)],_Form),
            NP1, NP2,
            [v([nom, akk], FormV),V]], SemRelS)
:-  not(debugging(typisieren)), % ohne Typen!
    NP1 = [np([_,3,_,Num1],[Kas1])|_],
    sem(NP1,SemNP1), sem(NP2,SemNP2),
    sem([v([nom,akk],FormV),V], SemV),
    (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,SemRelS).
```

⟨ *Beispiel* ⟩+≡

```
?- parses. den Galilei entdeckte.
+ lam(X, entdecken(galilei, X))

?- parses. der einen Planeten entdeckte.
+ lam(X, ex(Y, ex(Z, true, planet(Y, Z)),
               entdecken(X, Y)))
```

9. Ja/Nein-Fragesätze mit Vollverb

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([_Def],[_,_,ve]),
       [v([nom,akk],[3,Num,_,ind]),V], NP1, NP2],
     SemS)
:-   sem(NP1,SemNP1), sem(NP2,SemNP2),
     sem([v([nom,akk],[3,Num,_,ind]),V], SemV),
     NP1 = [np([_,3,_,Num1],[Kas1])|_Konst],
     (Kas1 = nom, Num1 = Num,
         Sem = SemNP1 * lam(X,SemNP2 *
                                lam(Y,SemV * X * Y))
     ; Kas1 = akk,
         Sem = SemNP1 * lam(Y,SemNP2 *
                                lam(X,SemV * X * Y))
     ), normalize(Sem,SemS).
```

10. Prädikativsätze

11. Nominalphrasen mit Relativsatz (Teil der Beschränkung):

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([_Def,3,Gen,Num],[Kas]),
     Det, [n([Gen],[Num,Kas]),N],
     [s([rel(Gen,Num)],[Tmp,ind,vl])|Const]],
    SemNP) :-
    sem(Det,SemDet),
    sem([n([Gen],[Num,Kas]),N],SemN),
    sem([s([rel(Gen,Num)],[Tmp,ind,vl])|Const],SemS),
    normalize(SemDet * lam(X, SemN*X & SemS*X),
              SemNP).
```

⟨*Beispiel* ⟩+≡

```
?- parses. ein Stern, den Galilei entdeckte.
+ lam(X, ex(Y, stern(Y)&entdecken(galilei, Y), X*Y))
```

Um die Berechnung der Lambda-Terme für alle Ausdrücke einer Datei zu testen, erweitern wir parses/0 zu parses/1:

⟨*semantik.pl*⟩+≡

```
parses(Dateiname) :-   % Bedeutung für alle Ausdrücke in
   open(Dateiname,read,Strom),     % der Datei berechnen
   parses2(Strom),
   close(Strom), nl, write('Fertig.').
parses2(Strom) :-
   read_sentence(Strom, Satz, []),
   (Satz = [] -> true  % Dateiende
   ; tokenize(Satz,Atome),nl,writeq(Atome),nl,
       (setof(Baum,tokenizer:parse(Atome,Baum),Trees)
       -> writeTrsLam(Trees,3), nl
       ; nl, tab(3),
           write('* keine Analysen gefunden *'),nl),
       parses2(Strom)   % weitere Sätze
   ).
```

Die Ausgabe kann man durch `tell(Ausgabedatei)` und anschließendes `told` in eine Datei umlenken:

⟨*Berechnen der Semantik für alle Ausdrücke einer Datei*⟩≡

```
?- tell('Semantik/nominalphrasen.test'). % Ausgabedatei
?- parses('Grammatik/nominalphrasen.txt').  % Beispiele
true.
?- told.                          % Ausgabe wieder am Bildschirm
```

Typisierung der Bedeutungsterme

Bei Artikeln und Quantoren im Plural hatten wir die distributive individuelle Lesart implementiert, aber die distributive paarweise Lesart zurückgestellt. Wir brauchen mindestens zwei Bedeutungen:

⟨*Bedeutungen eines Quantors im Plural*⟩≡

```
sem([det([quant],[_,pl,_]),[alle]],
    lam(N,lam(P,all(X,N*X,P*X)))).  % individuell distrib
sem([det([quant],[_,pl,_]),[alle]],
    lam(N,lam(P,all((X,Y),N*X & N*Y,P*X*Y)))). % paarweis
```

Welche Lesart bei `alle N` richtig ist, hängt *vom Kontext* der Verwendung der Nominalphrase ab. Das Kompositionsprinzip für Bedeutungen stimmt hier nicht: wir brauchen die Stelligkeit von `P`.

Aus Information im Lexikon sollen nun *Typen* für die abstrahierten Variablen ermittelt werden, sodaß wir bei `alle N` zwischen `lam(P:e->t,..P*X..)` und `lam(P:e*e->t,..P*X*Y..)` am Typ unterscheiden und syntaktisch inkorrekte `P*X*Y` vermeiden können.

Getypte λ-Terme

Eigennamen, Nomen und Verben erhalten im Lexikon Typen

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

und Anwendungsterme $(t \cdot s)$ müssen jetzt „typkorrekt“ sein, wo t von einem Funktionstyp $(\sigma \to \tau)$ und s vom Argumenttyp σ ist.

Ein Typkontext Γ ist eine Liste von Annahmen $x : \tau$ von Typen τ für Variable x. Ein Term t hat im Kontext Γ den Typ τ, wenn $\Gamma \vdash t : \tau$ nach folgenden Regeln herleitbar ist:

$$\frac{}{x : \sigma, \Gamma \vdash x : \sigma}\,(\mathit{Var}) \qquad \frac{x \not\equiv y, \quad \Gamma \vdash x : \sigma}{y : \tau, \Gamma \vdash x : \sigma}\,(\mathit{Var})$$

$$\frac{\Gamma \vdash t : \sigma \to \tau, \quad \Gamma \vdash s : \sigma}{\Gamma \vdash (t \cdot s) : \tau}\,(\mathit{App}) \qquad \frac{x : \rho, \Gamma \vdash t : \tau}{\Gamma \vdash \lambda x\, t : (\rho \to \tau)}\,(\mathit{Abs})$$

Jede Objektkonstante c habe einen variablenfreien Typ τ_c. Es sei

$$\frac{}{\Gamma \vdash c : \tau_c}\,(\mathit{Const}).$$

Manche Terme haben im selben Kontext viele Typen, z.B.

$$\frac{x : \mathsf{int} \vdash x : \mathsf{int}}{\vdash \lambda x.x : (\mathsf{int} \to \mathsf{int})}\,(Abs) \qquad \frac{x : \sigma \vdash x : \sigma}{\vdash \lambda x.x : (\sigma \to \sigma)}\,(Abs)$$

Andere Terme haben keinen Typ, z.B. die „Selbstanwendung“:

$$\frac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha \to \beta}\,(*Var,\ \alpha =?(\alpha \to \beta)) \qquad \dfrac{}{x : \alpha \vdash x : \alpha}\,(Var)}{x : \alpha \vdash (x \cdot x) : \beta}\,(App)}{\vdash \lambda x(x \cdot x) : (\alpha \to \beta)}\,(Abs)$$

Um zu Γ und t ein τ mit $\Gamma \vdash t : \tau$ zu finden, versucht man eine Typherleitung mit unbekanntem Typ α, sammelt bei den Regelanwendungen Gleichungen über die Typvariablen und löst sie:

$$\frac{}{x : \sigma, \Gamma \vdash x : \alpha}\,(\mathit{Var}) \qquad \alpha = \sigma$$

$$\frac{\Gamma \vdash t : \beta, \quad \Gamma \vdash s : \gamma}{\Gamma \vdash (t \cdot s) : \alpha}\,(\mathit{App}) \qquad \beta = (\gamma \to \alpha)$$

$$\frac{x : \beta, \Gamma \vdash t : \gamma}{\Gamma \vdash \lambda x\, t : \alpha}\,(\mathit{Abs}) \qquad \alpha = (\beta \to \gamma)$$

$$\frac{}{\Gamma \vdash c : \alpha}\,(\mathit{Const}). \qquad \alpha = \tau_c \text{ (Lexikon)}$$

Man kann eine „allgemeinste Typisierung“ für t *suchen*, indem ein Γ mit Annahmen $x : \alpha_x$ für freie Variable von t annimmt und die nötigen Typgleichungen durch Unifikation („symbolisch“) löst:

$$\frac{f : \alpha, x : \beta \vdash f : \gamma_1 \qquad f : \alpha, x : \beta \vdash x : \gamma_2}{f : \alpha, x : \beta \vdash (f \cdot x) : \gamma} \qquad \begin{array}{l} \gamma_1 = \alpha, \\ \gamma_2 = \beta, \\ \alpha = (\beta \to \gamma) \end{array}$$

Die bzw. eine allgemeinste Typisierung für $(f \cdot x)$ ist daher

$$f : (\beta \to \gamma),\ x : \beta \ \vdash \ (f \cdot x) : \gamma.$$

Alle anderen Typisierung kann man durch Einsetzung in die allgemeinste erhalten, z.B. mit $[\beta/\mathsf{nat}, \gamma/\mathsf{bool}]$

$$f : (\mathsf{nat} \to \mathsf{bool}),\ x : \mathsf{nat} \ \vdash \ (f \cdot x) : \mathsf{bool}.$$

Einfachheitshalber notiert man die Typen als oberen Index, wie in

$$(f^{\beta\to\gamma} \cdot x^{\beta})^{\gamma}, \qquad \lambda x^{\alpha\to\beta}\lambda y^{(\alpha\to\beta)\to\alpha}(x \cdot (y \cdot x)^{\alpha})^{\beta}.$$

Beachte: Nach (Abs) haben im Term $\lambda x^{\sigma} t$ alle freien Vorkommen von x in t denselben Typ σ. Insbesondere gilt:

$$\Gamma \vdash (\lambda x t \cdot s) : \tau \quad \Longrightarrow \quad \Gamma \vdash t[x/s] : \tau.$$

Man kann zeigen:

1. Typerhaltung unter Subjektreduktion Ist $\Gamma \vdash t : \tau$ und $t \to^* s$ eine „Vereinfachung“ von t, so ist $\Gamma \vdash s : \tau$.
2. Starke Normalisierung: Ist $\Gamma \vdash t : \tau$, so terminiert *jede* Reduktionsfolge $t \to t' \to t'' \to \ldots$ (bis auf α-Reduktionen) in derselben Normalform $nf(t)$.

Interpretation getypter λ-Terme

Zur Interpretation der getypten λ-Terme nimmt man

- Individuenbereiche D_σ für die Grundtypen σ, z.B. $D_{\mathsf{bool}} := \{0, 1\}$, $D_{\mathsf{nat}} = \mathbb{N}$,
- den „vollen“ Raum aller mengentheoretischen Funktionen,

$$D_{(\sigma \to \tau)} := \{ f \mid f : D_\sigma \to D_\tau \},$$

- als Applikation $\cdot$ die „wirkliche“ Funktionsanwendung,

$$f \cdot a := f(a) \qquad \text{für } f \in D_{(\sigma \to \tau)}, a \in D_\sigma$$

- für Konstante $c : \tau$ passende Elemente $c^{D_\tau} \in D_\tau$.

Terme wie die Selbstanwendung $\lambda x(x \cdot x)$ sind nicht typisierbar und erhalten keine Interpretation.

Implementierung der Typrekonstruktion

Wir wollen zu ungetyptem Term t die allgemeinste Typisierung berechnen und die Typen der abstrahierten Variablen bei der Bindung vermerken, z.B.

$$\lambda f(f \cdot 0) \;\mapsto\; \lambda f^{(\mathsf{nat}\to\alpha)}(f \cdot 0) \;:(\mathsf{nat}\to\alpha)\to\alpha.$$

⟨*Semantik/type.pl*⟩≡

```
% type(+Kontext,+LamTerm,-LamTermGetypt,-Typ)

% Lambda terms:
type([V:Ty|Kontext],X,X,Typ) :-
    var(X),
    !, ( X == V -> Typ = Ty ; type(Kontext,X,X,Typ) ).
type([],X,_,_) :-
    var(X), !, fail.
```

⟨*Semantik/type.pl*⟩+≡

```
type(Kontext,(R * S),(RTy * STy),Typ) :-
    !, type(Kontext,S,STy,ArgT),
    type(Kontext,R,RTy,FunT),
    (unify_with_occurs_check(FunT,(ArgT -> Typ))
    -> true  % write errors to output-file?
    ; % format('\nFehler: Argument ~w hat Typ ~w',[S,ArgT
      % format('\n        Funktion ~w hat Typ ~w',[R,FunT
      fail).
type(Kontext,lam(X,Term),lam(X:Ty,TermTy),Typ) :-
    var(X),
    !, type([X:Ty|Kontext],Term,TermTy,ResultTy),
    Typ = (Ty -> ResultTy).
type(Kontext,lam((X,Y),Term),lam((X,Y):Ty,TermTy),Typ) :-
    !, type([X:TyX,Y:TyY|Kontext],Term,TermTy,ResultTy),
    Ty = (TyX * TyY),
    Typ = (Ty -> ResultTy).
```

⟨*Beispiele zur Typisierung reiner Lambda-Terme*⟩≡

```
?- ['Semantik/type.pl'].

?- type([X:nat, F:TyF],(F * X),TermTy,Type).
TyF = (nat->Type),
TermTy = F*X.

?- type([X:nat],lam(F,(F * X)),TermTy,Type).
TermTy = lam(F: (nat->_G351), F*X),
Type = ((nat->_G351)->_G351).

?- type([X:nat,Y:bool,X:bool],X,TermTy,Type).

X = TermTy,
Type = nat.  % der linkeste für X angenommene Typ!
```

Den Typ der Wahrheitswerte nennen wir `t`, nicht `bool`.

⟨*Semantik/type.pl*⟩+≡

```
% Boolean combinations of formulas:
type(Kontext,neg(Fml),neg(FmlTy),t) :-
    !, type(Kontext,Fml,FmlTy,t).
type(Kontext,(Fml1 & Fml2),(Fml1Ty & Fml2Ty),t) :-
    !, type(Kontext,Fml1,Fml1Ty,t),
    type(Kontext,Fml2,Fml2Ty,t).
type(Kontext,(Fml1 \/ Fml2),(Fml1Ty \/ Fml2Ty),t) :-
    !, type(Kontext,Fml1,Fml1Ty,t),
    type(Kontext,Fml2,Fml2Ty,t).
type(Kontext,(Fml1 => Fml2),(Fml1Ty => Fml2Ty),t) :-
    !, type(Kontext,Fml1,Fml1Ty,t),
    type(Kontext,Fml2,Fml2Ty,t).
type(Kontext,(Fml1 <=> Fml2),(Fml1Ty <=> Fml2Ty),t) :-
    !, type(Kontext,Fml1,Fml1Ty,t),
    type(Kontext,Fml2,Fml2Ty,t).
```

⟨*Semantik/type.pl*⟩+≡

```
type(_,true,true,t) :- !.  % Bereich: true=unbeschränkt

% Verallgemeinerte Quantoren Qu(Var,Restriction,Body)
type(K,ex(X,R,Term),ex(X:Ty,RTy,TermTy),t) :-
    var(X), !,
    type(K,lam(X,R),lam(X:Ty,RTy),(Ty -> t)),
    type(K,lam(X,Term),lam(X:Ty,TermTy),(Ty -> t)).
type(K,all(X,R,Term),all(X:Ty,RTy,TermTy),t) :-
    var(X), !,
    type(K,lam(X,R),lam(X:Ty,RTy),(Ty -> t)),
    type(K,lam(X,Term),lam(X:Ty,TermTy),(Ty -> t)).
type(K,most(X,R,Term),most(X:Ty,RTy,TermTy),t) :-
    var(X), !,
    type(K,lam(X,R),lam(X:Ty,RTy),(Ty -> t)),
    type(K,lam(X,Term),lam(X:Ty,TermTy),(Ty -> t)).

type(K,iota(X,R,Term),iota(X:Ty,RTy,TermTy),t) :-
```

Für den Plural brauchen wir analoge Typisierungen von Paaren:

⟨*Semantik/type.pl*⟩+≡

```
% Paar-Quantoren Qu((Var,Var),Restriction,Body)
type(K,all((X,Y),R,Term),all((X,Y):Ty,RTy,TermTy),t) :-
    !, type(K,lam((X,Y),R),lam((X,Y):Ty,RTy),(Ty -> t)),
    type(K,lam((X,Y),Term),lam((X,Y):Ty,TermTy),(Ty -> t)
type(K,ex((X,Y),R,Term),ex((X,Y):Ty,RTy,TermTy),t) :-
    !, type(K,lam((X,Y),R),lam((X,Y):Ty,RTy),(Ty -> t)),
    type(K,lam((X,Y),Term),lam((X,Y):Ty,TermTy),(Ty -> t)
type(K,most((X,Y),R,Term),most((X,Y):Ty,RTy,TermTy),t) :-
    !, type(K,lam((X,Y),R),lam((X,Y):Ty,RTy),(Ty -> t)),
    type(K,lam((X,Y),Term),lam((X,Y):Ty,TermTy),(Ty -> t)

type(K,iota((X,Y),R,Term),iota((X,Y):Ty,RTy,TermTy),t) :-
    !, type(K,lam((X,Y),R),lam((X,Y):Ty,RTy),(Ty -> t)),
    type(K,lam((X,Y),Term),lam((X,Y):Ty,TermTy),(Ty -> t)
```

⟨*Semantik/type.pl*⟩+≡

```
type(K,anzahl(X,Fml,N),anzahl(X:Ty,FmlTy,NTy),t) :-
    !, type(K,lam(X,Fml),lam(X:Ty,FmlTy),(Ty -> t)),
    type(K,N,NTy,n).  % Typ n := number

% Questions: unary/binary Wh(Vars,Restriction,Formula)
type(K,qu(X,R,Term),qu(X:Ty,RTy,TermTy),list(Ty)) :-
    var(X), !, type(K,lam(X,R),lam(X:Ty,RTy),(Ty -> t)),
    type(K,lam(X,Term),lam(X:Ty,TermTy),(Ty -> t)).
type(K,qu((X,Y),R,Tm),qu((X,Y):Ty,RTy,TmTy),list(Ty)) :-
    !, type(K,lam((X,Y),R),lam((X,Y):Ty,RTy),(Ty -> t)),
    type(K,lam((X,Y),Tm),lam((X,Y):Ty,TmTy),(Ty -> t)).
```

Der Typ einer Frage ist `list(Ty)`, wenn nach Objekten vom Typ `Ty` gefragt wird: die Antwort ist eine Liste von Namen vom Typ `Ty`.

Die Typen der Grundprädikate (Verben, Nomen) und Konstanten (Namen) müssen in der Anwendung festgelegt werden, einem Modul datenbank.

⟨*Semantik/type.pl*⟩+≡

```
% Algebraic and atomic terms:
type(_Kontext,Modul:Term,Modul:Term,Typ) :-
    !, Typ = t.          % Grundaussagen der Datenbank
type(_Kontext,N,N,Typ) :-
    number(N), !, Typ = n.
type(Kontext,(X,Y),(XTy,YTy),Typ) :- % Variablentupel
    !, type(Kontext,X,XTy,TyX),
    type(Kontext,Y,YTy,TyY),
    Typ = (TyX * TyY).
```

⟨*Semantik/type.pl*⟩+≡

```
type(Kontext,Term,Term,Typ) :-
    Term =.. [F|Args],
    types(Kontext,Args,[],Typen),
    ( Typen = [],          TypF = Typ
    ; Typen = [T],         TypF = (T -> Typ)
    ; Typen = [T1,T2],     TypF = (T1*T2 -> Typ) ),
    functor(Term,F,Arity),
    (setof(Ty,datenbank:type(F/Arity,Ty),TypDecs)
    -> member(TypF,TypDecs) /* --- keine Fehlermeldung
       (\+ \+ member(TypF,TypDecs) % F hat passende Typen
       ->  member(TypF,TypDecs)    % wähle einen
       ; format('\nFehler: datenbank:~w hat Typen ~w, ',
                [F/Arity,TypDecs]),
         format('\nKontext erwartet ~w: ~w\n',
                [F/Arity,TypF]), fail) ---- */
    ; format('\ndatenbank:~w hat keinen Typ\n',[F/Arity]),
      fail
```

⟨*Semantik/type.pl*⟩+≡

```
% Typing a list of arguments
types(Kontext,[A|Args],Acc,TypArgs) :-
    type(Kontext,A,_,TypA),
    types(Kontext,Args,[TypA|Acc],TypArgs).
types(_Kontext,[],Typen,TypArgs) :-
    reverse(Typen,TypArgs).
```

In `type.pl` werden keine Typen für Prädikate und Namen der Anwendung vorgegeben. Damit man Formeln typisieren kann, müssen Typen der Prädikate und Namen bekannt sein.

Das überlassen wir der Anwendung, d.h. `datenbank:type/2`.

Voraussetzung für die Verwendung von getypten Verben, Nomen, Eigennamen sind Typdeklarationen in der Anwendung.

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

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

⟨*Beispiel: Typisierung einer Formel*⟩≡

```
?- [semantik,astronomie,'Semantik/type.pl'].
?- type([],all(X,astronom(X),ex(Y,mond(Y,jupiter),
                                    entdecken(X,Y))),
        Getyped,Ty).
Getyped = all(X:m, astronom(X),
              ex(Y:s, mond(Y, jupiter), entdecken(X, Y)))
Ty = t
```

⟨*Beipspiel: Erkennung eines Typfehlers*⟩≡

```
?- type([],all(X,astronom(X),
                ex(Y,mond(Y,X),entdecken(X,Y))),
        Getyped,Ty).

Fehler: datenbank:mond/2 hat Typen [ (s*s->t)],
Kontext erwartet mond/2: _G635*m->_G683
false.
```

Typüberladung, aber keine Subtypen

Im reinen λ-Kalkül wird durch die Typrekonstruktion eine „allgemeinste“ Typisierung (minimal vieler Annahmen) ermittelt:

Satz (R.Hindley 1978):

- Von jedem λ-Term t ist feststellbar, ob er typisierbar ist.
- Für typisierbares t gibt es ein (Γ, τ), sodaß die Typisierungen $\Delta \vdash t : \sigma$ von t genau die Instanzen $(\Delta, \sigma) = (S\Gamma, S\tau)$ von (Γ, τ) durch Typeinsetzungen $S : \mathrm{TypVar} \to \mathrm{Typ}$ sind.

Man nennt $\Gamma \vdash t : \tau$ dann eine „allgemeinste Typisierung“ von t.
Sie wird durch `type([X1:Alpha1,...],t,_,Alpha)` berechnet.

Wir erlauben im Lexikon *mehrere Typannahmen* für die Konstanten, z.B. für die Artikel und Quantoren im Plural. Die Typisierungen von t sind dann die Instanzen endlich vieler $(\Gamma_1, \tau_1), \ldots, (\Gamma_n, \tau_n)$.

Um Haupttypen zu haben, schließen wir Subtypen $\sigma \leq \tau$ aus.

Ausgabe des Baums mit getypten λ-Termen

Die Mehrdeutigkeiten, die durch den Plural entstehen, führen zu verschiedenen *getypten* λ-Termen eines Teilbaums, die wir alle unter die Wurzel des Teilbaums schreiben.

Bei der Anwendung werden dann nur typkorrekte Anwendungen zugelassen, sodaß unter den Bedeutungen der Teilterme nur typkorrekte Kombinationen berücksichtigt werden.

⟨*Parser/showTree.pl*⟩+≡

```
writeTrsLamTy([Baum|Baeume],Einrueckung) :-
    nl,writeTrLamTy(Baum,Einrueckung),
    writeTrsLamTy(Baeume,Einrueckung).
writeTrsLamTy([],_).

writeTrLamTy([],Einrueckung) :-
    tab(Einrueckung), writeq([]).
```

Problem: Müssen wir nicht einen Typkontext berücksichtigen?

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLamTy([Wurzel|Bs],Einrueckung) :-
    ( sem([Wurzel|Bs],_)  % Ausdruck mit Bedeutungen
    ->  % schreibe alle Bedeutungen unter die Wurzel
        tab(Einrueckung),writeq(Wurzel),
        (Bs = [[Blatt]] -> tab(1), writeq(Blatt) ; true),
        (sem([Wurzel|Bs],LamTerm),
            type([],LamTerm,LamTermGetypt,Typ),
            Term = (LamTermGetypt:Typ),
            (nl,tab(Einrueckung),write('+ '),writen(Term)
            fail
        ; true)
    ; % schreibe das bedeutungslose Wort
      tab(Einrueckung),writeq(Wurzel) ),
    ( Bs = [[Blatt]] -> true % schon geschrieben
    ; Einrueckung2 is Einrueckung + 3,
        writeTrsLamTy(Bs,Einrueckung2) ).
```

Zum Testen dient parsest, ein Parseraufruf mit Ausgabe der aus dem Syntaxbaum berechneten getypten Lambda-Terme:

⟨*semantik.pl*⟩+≡

```
:- ['Semantik/type.pl'].
parsest :-
    write('Beende die Eingabe mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    startsymbol(S),
    addTree:translate(S,Term,[Baum]),
    addDifflists:translate(Term,ExpTerm,Atomlist,[]),
    nl,write('Aufruf: '), portray_clause(ExpTerm),
    call(ExpTerm), write('Baum:   '),
    nl,writeTrLamTy(Baum,4),nl,
    fail.
parsest.
```

Die Dateien 'semantik.pl','astronomie.pl' seien geladen.

Wir aktivieren die Semantik-Klauseln für Artikel und Quantoren im Plural für die distributive Paar-Lesart:

⟨*Semantik/sem.pl*⟩+≡

```
% Paar-Lesarten der Pluraldeterminationen (Typen!)
⟨Semantik: Plural-Determiner⟩
⟨Semantik: Plural⟩
```

und tragen in `Semantik/lambdaTerme.pl` die Umbenennung von Paarquantoren nach:

⟨*Umbenennung von Paar-Quantoren*⟩≡

```
alpha(ex((U,V),R,S), L, ex((NewU,NewV),RDup,SDup)) :-
    !, alpha_seq([R,S],[(U,NewU),(V,NewV)|L],[RDup,SDup]).
alpha(all((U,V),R,S), L, all((NewU,NewV),RDup,SDup)) :-
    !, alpha_seq([R,S],[(U,NewU),(V,NewV)|L],[RDup,SDup]).
alpha(qu((U,V),R,S), L, qu((NewU,NewV),RDup,SDup)) :-
    !, alpha_seq([R,S],[(U,NewU),(V,NewV)|L],[RDup,SDup]).
```

In Nominalphrasen sollen von den Lesarten des Quantors diejenigen verworfen werden, die nicht zum (Beschränkungs-)Nomen passen.

Ungetypt haben wir drei (eine synt.inkorrekte) Lesarten:

⟨*Beispiel: alle Sterne*⟩≡

```
?- parses. alle Sterne.
Baum:
    np([quant, 3, mask, pl], [nom])
    + lam(X, all(Y, stern(Y), X*Y))
    + lam(X, all((Y, Z), stern(Y)&stern(Z), X*Y*Z))
    + lam(X, all((Y, Z), stern(Y)*Z, X*Y*Z))
       det([quant], [mask, pl, nom]) alle
       + lam(X, lam(Y, all(Z, X*Z, Y*Z)))
       + lam(X, lam(Y, all((Z, A1), X*Z&X*A1, Y*Z*A1)))
       + lam(X, lam(Y, all((Z, A1), X*Z*A1, Y*Z*A1)))
       n([mask], [pl, nom]) 'Sterne'
       + lam(X, stern(X))
```

Zu stern/1 werden Quantorlesarten mit 2-stelliger Beschränkung durch die Typisierung beseitigt: (Fehlermeldung unterdrüken?)

⟨*Beispiel* ⟩+≡

```
?- parsest. alle Sterne.
Baum:
    np([quant, 3, mask, pl], [nom])
    + lam(X:(s->t), all(Y:s,stern(Y),X*Y)): ((s->t)->t)
    + lam(X: (s->s->t), all((Y, Z):s*s, stern(Y)&stern(Z)
                              X*Y*Z)) : ((s->s->t)->t)
Fehler: Argument _G816 hat Typ _G1008
         Funktion stern(_G815) hat Typ t
        det([quant], [mask, pl, nom]) alle
        + ... : ((Y->t)-> (Y->t)->t)
        + ... : ((Y->t)-> (Y->Y->t)->t)
        + ... : ((Y->Z->t)-> (Y->Z->t)->t)
        n([mask], [pl, nom]) 'Sterne' + ...: (s->t)
```

Zu planet/2 (genauer: symmetrischen N/2) sollen Quantorlesarten mit 1-stelliger Beschränkung durch die Typisierung beseitigt werden:

⟨*Beispiel (erste Lesart wegen der Umwandlung zu planet/1)*⟩≡

```
?- parsest. alle Planeten.
Baum:
    np([quant, 3, mask, pl], [nom])
    + lam(X:(s->t), all(Y:s, ex(Z:s,true,planet(Y,Z)),
                          X*Y)) : ((s->t)->t)
    + lam(X:(s->s->t),
          all((Y,Z):s*s, ex(A1:s,true,planet(Y, A1))
                          &ex(B1:s,true,planet(Z, B1)),
              X*Y*Z)): ((s->s->t)->t)
Fehler: Argument _G1184 hat Typ _G1440
        Funktion ex(_G1276,true,planet(_G1183, _G1276))
                 hat Typ t
    + lam(X: (s->s->t), all((Y, Z):s*s, planet(Y, Z),
                            X*Y*Z)) : ((s->s->t)->t)
       det([quant], [mask, pl, nom]) alle
```

Jetzt bleibt die Kombination von Plural-Nominalphrasen und symmetrischen Prädikaten zu behandeln. Das Ziel ist:

- bei symmetrischem Prädikat wird als Bedeutung der Subjekt-NP die (distributive) Paar-Lesart genommen.
- das Reziprokpronomen `einander` macht aus einem transitiven Verb $v/2$ das symmetrische Prädikat $\lambda x \lambda y (v(x,y) \,\&\, v(y,x))$.
- das Reflexivpronomen `sich` macht aus einem transitiven Verb $v/2$ das intransitive Prädikat $\lambda x.v(x,x)$.

Das geht auch mit anderen Argumentstellen des Verbs, z.B. für Dativobjekte, wird hier aber dafür nicht implementiert:

- `Er gönnte sich eine Ruhepause.`
- `Sie gönnten einander die Auszeichnung.`

Syntaxregel NP + intransitives Verb:

⟨*Grammatik/saetze.pl*⟩+≡

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

⟨*Semantik/sem.pl*⟩+≡

```
sem([pron([refl],_Form),[sich]], lam(P,lam(X,P*X*X))).
```

Übungsaufgabe: Implementiere die Bedeutung von reflexiven Nominalphrasen, ihre Verwendung in einfachen Sätzen, und deren Bedeutung.

12. Semantik reflexiver und reziproker Nominalphrasen: die Bedeutung ist die des eingebetteten Pronomens:

⟨*Semantik/sem.pl*⟩+≡

```
sem([np([refl,3,Gen,Num],[Kas]),
      [pron([refl],[Gen,Num,Kas]),VF]], SemPron) :-
    sem([pron([refl],[Gen,Num,Kas]),VF], SemPron).

sem([np([rezi,3,Gen,Num],[Kas]),
      [pron([rezi],[Gen,Num,Kas]),VF]], SemPron) :-
    sem([pron([rezi],[Gen,Num,Kas]),VF], SemPron).

sem([pron([rezi],_Form),[einander]],
    lam(P,lam(X,lam(Y,P*X*Y & P*Y*X)))).
```

13. Semantik einfacher Sätze mit reflexiven/reziproken Objekten:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([_Def],[_,_,vz]),
       NP1, [v([nom,akk],[3,Num,_,ind]),V], NP2],
     SemS)
:-   sem(NP1,SemNP1), sem(NP2,SemNP2),
     sem([v([nom,akk],[3,Num,_,ind]),V], SemV),
     NP1 = [np([_,3,_,Num1],[nom])|_Konst1],
     NP2 = [np([Def,3,_,_Num2],[akk])|_Konst2],
     ( Num1 = Num, Def=refl,
         Sem = SemNP1 * (SemNP2 * SemV)
     ; Num1 = Num, Def=rezi,
         Sem = SemNP1 * (SemNP2 * SemV)
     ), normalize(Sem,SemS).
```

Ebenso für ja/nein-Fragen mit reflexiven/reziproken Objekten:

⟨*Semantik/sem.pl*⟩+≡

```
sem([s([_Def],[_,_,ve]),
       [v([nom,akk],[3,Num,_,ind]),V], NP1, NP2],
     SemS)
:-   sem(NP1,SemNP1), sem(NP2,SemNP2),
     sem([v([nom,akk],[3,Num,_,ind]),V], SemV),
     NP1 = [np([_,3,_,Num1],[nom])|_Konst1],
     NP2 = [np([Def,3,_,_Num2],[akk])|_Konst2],
     ( Num1 = Num, Def=refl,
         Sem = SemNP1 * (SemNP2 * SemV)
     ; Num1 = Num, Def=rezi,
         Sem = SemNP1 * (SemNP2 * SemV)
     ), normalize(Sem,SemS).
```

Wenn man die Typfehlermeldungen unterdrückt, bleiben die sinnvollen Bedeutungsterme übrig: (inkl. `planet/2` als symm.Verb)

⟨*Beispiel* ⟩+≡

```
?- parsest. umkreisen alle Sterne sich.
    + all(X:s, stern(X), umkreisen(X, X)):t
?- parsest. umkreisen alle Sterne einander.
    + all((X, Y):s*s, stern(X)&stern(Y),
                      umkreisen(X, Y)&umkreisen(Y, X)):t

?- parsest. umkreisen alle Planeten sich.
    + all(X:s, ex(Y:s,true,planet(X,Y)), umkreisen(X,X)):
?- parsest. umkreisen alle Planeten einander.
    + all((X, Y):s*s, ex(Z:s, true, planet(X, Z)) &
                      ex(A1:s, true, planet(Y,A1)),
                      umkreisen(X, Y)&umkreisen(Y, X)):t
    + all((X, Y):s*s, planet(X, Y),
                      umkreisen(X, Y)&umkreisen(Y, X)):t
```

Soviel zum Plural. Einige Probleme bleiben:

- Durch Typisierung kann man nicht zwischen transitiven und symmetrischen Verben und Nomen unterscheiden, man braucht dazu syntaktische Kategorien `vsym([nom,akk],Form)`, `rnsym([Gen],[Num,Kas])`.
- Für die kollektive Lesart des Plurals braucht man weitere Typen, `list(Typ)`, und in Modellen eine Liste aller Objekte.
- Höherstufige Quantoren werden nicht behandelt: *viele Eigenschaften von Maria hat Fritz nicht*. Dazu müßte man in Modellen einen zu großen Bereich durchsuchen (exponentiell in der Zahl der Individuen).

Programme zum Testen:

- Parse alle Sätze einer Datei: `tokenizer:parse(+Dateiname)`
- Semantik aller Sätze einer Datei: `parses(+Dateiname)`
- Typsemantik der Sätze einer Datei: `parsest(+Dateiname)`

Testdateien:

- Parsen: `Grammatik/saetze.txt`, `Grammatik/nominalphrasen.txt`
- Semantik mit ungetyptem λ-Term
- Semantik mit getyptem λ-Term
- Semantik mit ungetypter λ-DRS
- Semantik mit getypter λ-DRS
- Semantik mit getypter λ-DRS und Pronomenauflösung:

Nächstes Ziel:

- Auswertung in einem endlichen Modell,

oder

- abstrakte Semantik von absoluten/relativen Adjektiven und
- Vergleichkonstruktionen: (Manche Leute sind klüger als andere.), oder:

oder

- Personalpronomen
- Diskursrepräsentation und Pronomenauflösung

Auswertung in einem endlichen Modell

Ein endliches relationales Modell

$$\mathcal{D} = (\langle D_\sigma \mid \sigma \in Typ \rangle, R^{\mathcal{D}}, \ldots, c^{\mathcal{D}}, \ldots)$$

geben wir durch einen Modul `datenbank` an, der Individuennamen (Prolog-Atome) und Relationsnamen (Prolog-Prädikate) exportiert.

Siehe `Anwendungen/Astronomie/datenbank.pl`.

Es werden aber nur die endlich vielen Bereiche D_σ für Grundtypen σ angegeben, nicht die höherstufigen $D_{(\rho \to \tau)}$.

Die Auswertung von Termen und Formeln erfolgt durch `eval/3` und `eval/2`, die mit einer Belegung ihrer freien Variablen einen Wert (Individuum, Relation, Wahrheitswert) berechnet.

Die Auswertung eines Terms (hier: nur Variable und Konstante) mit `eval/3` sucht den Wert von Variablen in der Belegung:

⟨*Semantik/auswertung.pl*⟩≡

```
% eval(+Belegung,+Term,-Objekt)
eval([(Var,Obj)|Env],X,Val) :-
    var(X),
    !, (Var == X -> Val = Obj ; eval(Env,X,Val)).
eval([],X,_) :-
    var(X),
    !, format('\nFehler: unbelegte Variable').
eval(_Env,C,Val) :-
    functor(C,C,0), !, Val = C.

% evals(+Belegung,+Terme,-Objekte)
evals(Env,[Term|Terms],[Val|Vals]) :-
    eval(Env,Term,Val), evals(Env,Terms,Vals).
evals(_Env,[],[]).
```

Die Auswertung von (neu) *Anzahltermen* hat als Wert die Zahl der betreffenden Objekte (`sort/2` entfernt Duplikate):

⟨*Semantik/auswertung.pl*⟩+≡

```
eval(_Env,anzahl(X,_Restr,_Formel),_N) :-
    var(X),!,fail. % nur getypte Ausdrücke auswerten

eval(Env,anzahl(X:Ty,Restr,Formel),N) :-
    var(X),!,
    findall(O,(objekt(O:Ty),
               eval([(X,O)|Env],(Restr & Formel))),Os),
    sort(Os,Objs), length(Objs,N).

eval(Env,anzahl((X,Y):TyX*TyY,Restr,Formel),N) :-
    var(X),var(Y),!,
    findall((A,B),(objekt(A:TyX),objekt(B:TyY),
               eval([(X,A),(Y,B)|Env],(Restr & Formel))),
            ABs), sort(ABs,Pairs), length(Pairs,N).
```

Die Auswertung einer Formel mit `eval/2` gelingt, wenn die Formel wahr ist, sonst scheitert sie:

⟨*Semantik/auswertung.pl*⟩+≡

```
% eval(+Belegung,+Formel)

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

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

eval(_Env,true) :- !.
```

⟨*Semantik/auswertung.pl*⟩+≡

```
% Prädikatenlogische Quantoren:
eval(Env,all(Var,Restr,Fml)) :-
    !, eval(Env, neg(ex(Var,Restr,neg(Fml)))).

eval(Env,ex(Var,Restr,Fml)) :-
    var(Var), !, % ungetypt
    datenbank:objekt(Obj:_Ty), % suche in der Datenbank
    eval([(Var,Obj)|Env], Restr),
    eval([(Var,Obj)|Env], Fml), !. % nur erste Lösung
```

Wir machen das explizit (nicht über die Belegung durch Prolog),
sodaß Bindungsvariable nicht verschieden sein müssen:

⟨*Wiederverwendung quantifizierter Variable*⟩≡

```
?- eval([],ex(X,astronom(X),
                ex(X,stern(X),umkreisen(X,sonne)))).
true
```

Verallgemeinerte Quantoren (`viele, die meisten`):

⟨*Semantik/auswertung.pl*⟩+≡

```
% Verallgemeinerte Quantoren: (TODO)
⟨Auswertung verallgemeinerter Quantoren⟩
```

Übungsaufgabe: verallgemeinerte Quant., Paar-Quant., getypte Qu.

Arithmetische Prädikate werden mit Hilfe der entsprechenden Prolog-Prädikate ausgewertet:

⟨*Semantik/auswertung.pl*⟩+≡

```
eval(Env,(X < Y))  :-
    !, evals(Env,[X,Y],[VX,VY]), VX < VY.
eval(Env,(X =< Y))  :-
    !, evals(Env,[X,Y],[VX,VY]), VX =< VY.
eval(Env,eq(X,Y))  :-
    !, evals(Env,[X,Y],[VX,VY]), VX == VY.
```

⟨*Semantik/auswertung.pl*⟩+≡

```
% Grundformeln (der Datenbank)
eval(Env,Formel) :-
    functor(Formel,Pn,Arity),
    Arity > 0, !,
    Formel =.. [Pn|Args],
    evals(Env,Args,Vals),
    EFormel =.. [Pn|Vals], call(EFormel).
```

⟨*Semantik/auswertung.pl*⟩+≡

```
% Anzahlaussagen
eval(Env,anzahl(Var,Formel,Zahl)) :-
    !, antworten(Env,qu(Var,true,Formel),Objekte),
    length(Objekte,Zahl). % N >= Zahl ?
```

⟨*semantik.pl*⟩+≡

```
:- ['Semantik/auswertung.pl'].

frage :- write('Beende die Frage mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    Startsymbol = s([qu],[_Temp,ind,Vst]),
    parse(Startsymbol,Atomlist,Baum,ExpTerm),
    (call(ExpTerm) ->
        (sem(Baum,LamTerm) -> % nur erste Bedeutung!
            (Vst = ve -> beantworte(LamTerm,Wert)
            ; antworten(LamTerm,Wert)),
            nl,write('Antwort: '),write(Wert)
        ; write('\nÜbersetzung Frage->Formel fehlt.'),
            nl,nl,write('Baum:   '), writeq(Baum)
        )
    ; nl, write('\nEingabe syntaktisch nicht erkannt.\n')
    ).
```

⟨*semantik.pl*⟩+≡

```
parse(Startsymbol,Atomlist,Baum,ExpTerm) :-
    addTree:translate(Startsymbol,Term,[Baum]),
    addDifflists:translate(Term,ExpTerm,Atomlist,[]),
    nl,write('Aufruf: '), portray_clause(ExpTerm).
```

Da die Typisierung unsinnige Lesarten beseitigen sollte, sollten wir eigentlich nur getypte Fragen beantworten.

Oben sind aber auch Fragen mit ungetypter Semantik nicht ausgeschlossen.

⟨*semantik.pl*⟩+≡

```
% Abkürzung:
semTy(Baum,LamTermTyped) :-
    sem(Baum,LamTerm),
    type([],LamTerm,LamTermTyped,_Type).
```

⟨*semantik.pl*⟩+≡

```
fraget :- write('Beende die Frage mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    Startsymbol = s([qu],[_Temp,ind,Vst]),
    parse(Startsymbol,Atomlist,Baum,ExpTerm),
    (call(ExpTerm) ->
        (semTy(Baum,LamTerm)  % nur erste Bedeutung!
        -> (Vst = ve -> beantworte([],LamTerm,Wert)
            ; antworten([],LamTerm,Wert)),
            nl,write('Antwort: '),write(Wert)
        ; write('\nÜbersetzung Frage->Formel fehlt.'),
            nl,nl,write('Baum:   '), writeq(Baum)
        )
    ; nl, write('\nEingabe syntaktisch nicht erkannt.\n')
    ).
```

Bei der Auswertung von Aussagen hatten wir keinen expliziten Wahrheitswert berechnet; wir antworten auf eine beweisbare Ja/Nein-Aussage mit `ja`, auf unbeweisbare mit `nein`:

⟨*Semantik/auswertung.pl*⟩+≡

```
beantworte(PL_Aussage,Wert) :-
    eval([],PL_Aussage) -> Wert = ja ; Wert = nein.

antworten(qu(V,Restr,Formel), As) :-
    var(V), !,
    findall(O,(objekt(O:_),
               eval([(V,O)],(Restr & Formel))),Ws),
    sort(Ws,As).
antworten(qu(V:Typ,Restr,Formel), As) :-
    var(V), !,
    findall(O,(objekt(O:Typ),
               eval([(V,O)],(Restr & Formel))),Ws),
    sort(Ws,As).
```

⟨*Semantik/auswertung.pl*⟩+≡

```
eval(Env,qu((X,Y):(TyX * TyY),Restr,Formel), Paare) :-
    !, findall((A,B),(objekt(A:TyX), objekt(B:TyY),
                eval([(X,A),(Y,B)|Env],(Restr & Formel))),
                ABs),
    sort(ABs,Paare).
```

Übung: Auswertung verallgemeinerter Quantoren

In Semantik/auswertung.pl fügen wir an passender Stelle ein:

⟨*Auswertung verallgemeinerter Quantoren*⟩≡

```
% all, ex, most, iota

eval(_Env,most(X,_Restr,_Formel)) :-
    var(X), !, fail. % keine Semantik ungetypter Aussagen
eval(Env,most(X:Ty,Restr,Formel)) :-
    var(X), !,
    eval(Env,anzahl(X:Ty,Restr,Formel),NPos),
    eval(Env,anzahl(X:Ty,Restr,neg(Formel)),NNeg),
    NPos > NNeg.
eval(Env,most((X,Y):(TyX * TyY),Restr,Formel)) :-
    eval(Env,anzahl((X,Y):(TyX*TyY),Restr,Formel),NPos),
    eval(Env,anzahl((X,Y):(TyX*TyY),Restr,neg(Formel)),NN
    NPos > NNeg.
```

⟨*Auswertung von Interrogativquantoren*⟩≡

```
% eval(Env,qu(X,Restr,Formel)) :-
% eval(Env,qu(X:Ty,Restr,Formel)) :-
% eval(Env,qu((X,Y):(TyX * TyY),Restr,Formel)) :-
```

Auswertung getypter Fragen

⟨*semantik.pl*⟩+≡

```
beantworte(Env,Aussage,Wert) :-
    eval(Env,Aussage) -> Wert = ja ; Wert = nein.

antworten(Env,qu(V:Ty,Restr,Formel), Objekte) :-
    var(V),!,
    findall(O,(objekt(O:Ty),
               eval([(V,O)|Env],(Restr & Formel))),Os),
    sort(Os,Objekte).

antworten(Env,qu((X,Y):(TyX*TyY),Restr,Formel),Paare) :-
    !, findall((A,B),(objekt(A:TyX),objekt(B:TyY),
         eval([(X,A),(Y,B)|Env],(Restr & Formel))),ABs),
    sort(ABs,Paare).
```

⟨*Auswertung verallgemeinerter Quantoren*⟩+≡

```
eval(Env,qu((X,Y):(TyX * TyY),Restr,Formel), Paare) :-
    !, findall((A,B),(objekt(A:TyX), objekt(B:TyY),
                eval([(X,A),(Y,B)|Env],(Restr & Formel))),
                ABs),
    sort(ABs,Paare).
```

Satzkombinationen und „Diskurse“

R. Montague hatte (1970) gezeigt, wie man einzelne Aussagesätze in logische Formeln übersetzen kann; diese Formeln *repräsentieren* die Bedeutung der Sätze.

Ein *Diskurs* ist eine Folge von Sätzen. Die *Diskursrepräsentationstheorie* (DRT) von H.Kamp und U.Reyle (1986) ist eine Theorie über die Bedeutung von Diskursen. Haupsächlich geht es darum,

- eine Repräsentation von „Diskursen“ zwischen der natürlichen Sprache und einer logischen Formulierung zu finden,
- die auch Querbeziehungen zwischen Sätzen erfaßt, insbesondere die Bezüge von Pronomen auf Nominalphrasen in anderen Sätzen.

In einem Text $T = S_1\ S_2\ \ldots\ S_n$ aus Sätzen S_j wie

> *Ein König hatte zwei Söhne. Den älteren von ihnen schickte er in den Krieg gegen seine Feinde.*

werden Aussagen über Objekte gemacht, die i.a. keinen Namen tragen, auf die man sich in Folgesätzen aber bezieht:

> *(Ein König)$_i$ hatte (zwei Söhne)$_j$. (Den älteren von ihnen$_j$)$_k$ schickte er$_i$ in (den Krieg)$_l$ gegen (seine$_{i/k}$ Feinde)$_m$.*

Die Objekte werden durch *Diskursreferenten* $i, j, k, \ldots$ repräsentiert.

Im einfachsten Fall stehen sie für Individuen; sie können aber auch für „Pluralobjekte“, Fakten, Ereignisse o.ä. stehen, wie n in

> *... Das$_n$ hätte er$_i$ besser nicht getan.*

Um Texte S_1 S_2 ... durch logische Formeln zu repräsentieren, sollte schließlich eine Formel der Gestalt

$$\exists \vec{x}_1 \ldots \exists \vec{x}_r (\varphi_1(\vec{x}_1, \ldots, \vec{x}_r) \wedge \ldots \wedge \varphi_r(\vec{x}_1, \ldots, \vec{x}_r))$$

gebildet werden, wobei

- $\vec{x}_i = x_{i,1} \ldots x_{i,n_i}$ die durch indefinite Nominalphrasen, Eigennamen u.a. von S_i eingeführten Diskursreferenten sind,
- $\varphi_i(\vec{x}_1, \ldots, \vec{x}_i)$ den Bedeutungsbeitrag von Satz S_i ausdrückt.

Problem 1: Da man sich auf einen durch Satz S_j eingeführten Diskursreferenten in Sätzen S_k, $j < k$, beziehen kann, muß man

- Quantoren $\exists \vec{x}_k$ und Bedingungen φ_k in den Wirkungsbereich früherer Quantoren einbauen: $\exists \vec{x}_1 \ldots \exists \vec{x}_k (\varphi_1 \wedge \ldots \wedge \varphi_{k-1} \wedge \varphi_k)$ bzw. $\exists \vec{x}_1 (\varphi_1 \wedge \exists \vec{x}_2 (\ldots \wedge \exists \vec{x}_{k-1} (\varphi_{k-1} \wedge \exists \vec{x}_k \varphi_k) \ldots))$

Problem 2: Welche Ausdrücke erfordern neue Diskursreferenten und auf welche vorhandenen können sich Pronomina beziehen?

- Fritz$_i$ hat (kein gutes Buch)$_?$. Er$_i$ kauft es$_?$.
- Fritz$_i$ kennt (viele Leute)$_j$. Er$_i$ läd sie$_{j?}$ zu seiner$_i$ Feier ein.
- (Ein|Jeder Wal)$_?$ ist (ein Säugetier)$_?$. Seine$_?$ Jagd ist erlaubt.
- (Alle Anfänger)$_?$ reden Unsinn. Ich ignoriere sie$_?$.
- Wenn du (eine gute Flasche Wein)$_i$ findest, bringe sie$_?$ mit.

Man braucht Einschränkungen darüber, auf welche Diskursreferenten sich welche Pronomina beziehen können.

- Wenn jemand$_i$ (ein elektrisches Gerät)$_j$ benutzt, muß er$_i$ es$_j$ beim Landeanflug ausschalten.
- (Manche Leute)$_i$ tragen (die Nase)$_j$ hoch. Ich ignoriere sie$_{i|*j}$.

Grundidee der Diskursrepräsetationstheorie

Der Aufbau einer Diskursrepräsentation (DRS) erfolgt dynamisch:

1. Ein Text (Diskurs) ist eine *Folge* $S_1\ S_2\ \ldots$ von Sätzen.
2. Jedem Anfangsstück $S_1 \cdots S_i$ wird eine DRS $B_i = \langle D_i, C_i \rangle$ zugeordnet, die aus einem „Universum“ D_i von Referenten und einer Menge C_i von Bedingungen besteht.
3. Im *Kontext* B_i wird der nächste Satz S_{i+1} interpretiert und führt zu einer DRS B_{i+1} für $S_1 \ldots S_{i+1}$.
4. Diskursrepräsentanten im „Universum“ von B_i sind Kandidaten zur Auflösung von Pronomina in S_{i+1}.

Die Bedeutung eines einzelnen Satzes S_{i+1} besteht dann in der Veränderung einer DRS; sie ist also eine Übergangs*relation* mit

$$B_i\ [\![S_{i+1}]\!]\ B_{i+1}.$$

Diskursrepräsentationsstrukturen

Eine **Diskursrepräsentationsstruktur** (DRS) ist ein Paar $\langle D, C \rangle$ aus einer Liste D von *Diskursreferenten* (Variablen) und einer Liste C von *Bedingungen* über diese.

DR-Strukturen und *Bedingungen* werden simultan definiert:

1. Variable und Konstante sind Terme.
2. Jede atomare Formel $R(t_1, \ldots, t_n)$ ist eine Bedingung, und $\top$.
3. Jede Gleichung $t = t'$ zwischen Termen ist eine Bedingung.
4. Ist D eine endl.Menge von Variablen und C eine endl.Menge von Bedingungen, so ist $\langle D, C \rangle$ eine Struktur (DRS).
5. Sind B_1, B_2 Strukturen, so sind $B_1 \vee B_2$, $B_1 \Rightarrow B_2$ und $\neg B_1$ Bedingungen.

Beachte: es gibt keine expliziten Quantoren und keine Konjunktion. (Die Bedingungen C in $\langle D, C \rangle$ sind konjunktiv gemeint.)

Darstellung von Strukturen

Eine Struktur $\langle D, C \rangle$ mit $D = \{x_1, \ldots, x_n\}$ und $C = \{\varphi_1, \ldots, \varphi_n\}$ wird oft durch

$x_1, \ldots, x_n$
$\varphi_1, \ldots, \varphi_m$

oder

$x_1 \ldots x_n$
φ_1
$\vdots$
φ_m

dargestellt, zum Beispiel

x	
y, z	
$\varphi(x, y)$ $\chi(y, z)$	$\Rightarrow$ $\psi(x, z)$

Konstruktion einer DRS aus dem Syntaxbaum

Das macht man am besten analog zum Aufbau von λ-Termen in der Montague-Grammatik. (Bottom-Up den Syntaxbaum entlang.)

1. Jedem Nomen n wird im Lexikon die λ-DRS $\lambda x.\boxed{\begin{array}{|c|}\hline \\ \hline n(x) \\ \hline\end{array}}$ zugeordnet. Analog für intransitive Verben und Adjektive.
2. Jedem transitiven Verb v wird im Lexikon die λ-DRS $\lambda P \lambda y.(P * \lambda x.\boxed{\begin{array}{|c|}\hline \\ \hline v(x,y) \\ \hline\end{array}})$ zugeordnet.
3. Ein Pronomen erhält die λ-DRS $\lambda P.(\boxed{\begin{array}{|c|}\hline x_{neu} \\ \hline \\ \hline\end{array}} \otimes P * x_{neu})$.
4. Ein Eigenname c erhält die λ-DRS $\lambda P.(\boxed{\begin{array}{|c|}\hline x_{neu} \\ \hline x_{neu} = c \\ \hline\end{array}} \otimes P * x_{neu})$.

5. dem unbestimmten Artikel *ein* (bzw. dem Existenzquantor) wird die λ-DRT $\lambda N.\lambda P.(\boxed{x \\ \\} \otimes N * x \otimes P * x)$ zugeordnet.
6. dem Determinator *jeder* (bzw. dem Allquantor) wird die λ-DRT $\lambda N.\lambda P.\boxed{\\ \boxed{x \\ \\} \otimes N * x \Rightarrow P * x}$ zugeordnet.

Mit $\otimes$ ist das *Verbinden* (*mergen*) von DRSen gemeint:

$$\boxed{x_1, \ldots, x_n \\ \varphi_1, \ldots, \varphi_m} \otimes \boxed{y_1, \ldots, y_p \\ \psi_1, \ldots, \psi_q} = \boxed{x_1, \ldots, x_n, y_1, \ldots, y_p \\ \varphi_1, \ldots, \varphi_m, \psi_1, \ldots, \psi_q}$$

Mit N, P wird von λ-DRSen abstrahiert.

Beispiel 1. Für die NP *Jeder Stern* errechnet man bei Montague

$$\lambda N.\lambda P \forall x(N * x \to P * x) * \lambda x.\,\mathrm{stern}(x)$$
$$= \quad \lambda P \forall x(\mathrm{stern}(x) \to P * x)$$

Hier konstruiert man entsprechend

$$\lambda N.\lambda P.\boxed{\boxed{\begin{array}{|c|}\hline x \\ \hline \end{array}} \otimes N * x \Rightarrow P * x} * \lambda x.\boxed{\begin{array}{|c|}\hline \\ \hline \mathrm{stern}(x) \\ \hline \end{array}}$$

$$= \quad \lambda P.\boxed{\boxed{\begin{array}{|c|}\hline x \\ \hline \end{array}} \otimes \boxed{\begin{array}{|c|}\hline \\ \hline \mathrm{stern}(x) \\ \hline \end{array}} \Rightarrow P * x}$$

$$= \quad \lambda P.\boxed{\boxed{\begin{array}{|c|}\hline x \\ \hline \mathrm{stern}(x) \\ \hline \end{array}} \Rightarrow P * x}$$

Wendet man diese λ-DRT auf die des intransitiven Verbs *leuchtet* an, so erhält man die DRT für den Satz *Jeder Stern leuchtet.*

$$\lambda P.\boxed{\quad\boxed{\begin{array}{|c|}\hline x \\ \hline \text{stern}(x)\\ \hline\end{array}} \Rightarrow P * x\quad} * \lambda x.\boxed{\begin{array}{|c|}\hline \\ \hline \text{leuchtet}(x)\\ \hline\end{array}}$$

$$= \boxed{\quad\boxed{\begin{array}{|c|}\hline x \\ \hline \text{stern}(x)\\ \hline\end{array}} \Rightarrow \boxed{\begin{array}{|c|}\hline \\ \hline \text{leuchtet}(x)\\ \hline\end{array}}\quad}$$

Beispiel 2: Er leuchtet.

$$\lambda P(\boxed{\begin{array}{|c|}\hline x \\ \hline \\ \hline\end{array}} \otimes P * x) * \lambda x.\boxed{\begin{array}{|c|}\hline \\ \hline \text{leuchtet}(x)\\ \hline\end{array}} = (\boxed{\begin{array}{|c|}\hline x \\ \hline \\ \hline\end{array}} \otimes \boxed{\begin{array}{|c|}\hline \\ \hline \text{leuchtet}(x)\\ \hline\end{array}})$$

DRS für einen Text

- Dem leeren Kontext entspricht die leere DRS, $\langle \emptyset, \emptyset \rangle$.
- Ist B_i die DRS, die dem Kontext $S_1 \ldots S_i$ entspricht, und B die DRS des nächsten Satzes S_{i+1} (isoliert), so entsteht die DRS B_{i+1} für $S_1 \ldots S_{i+1}$ durch *Einbauen von B in B_i*.
- Im wesentlichen heißt das: $B_{i+1} = B_i \otimes B$.

Aber: wie behandelt man die Pronomina x_{neu} aus B?

- Suche einen *erreichbaren* vorhandenden Diskursreferenten x,
- Erweitere die Bedingungen um $x_{neu} = x$.

Beispiel: DRS für einen Text

Als einzelne Sätze:

Fritz kauft ein Buch.

x, y
$\text{kauft}(x, y)$
$x = \text{fritz}$
$\text{buch}(y)$

Er liest es.

u, v
$\text{liest}(u, v)$

Als Text, nach Auflösung der Pronomina (durch die Gleichungen):

Fritz kauft ein Buch. Er liest es.

x, y, u, v
$\text{kauft}(x, y)$
$x = \text{fritz}$
$\text{buch}(y)$
$\text{liest}(u, v)$
$u = x$
$v = y$

Pronomenauflösung: Erreichbarkeit von Referenten

Für Pronomina werden neue Diskursreferenten y eingeführt. Deren Auflösung besteht darin, sie durch eine Bedingung $y = x$ auf einen vorhandenen Diskursreferenten x zu beziehen.

Nicht alle vorhandenen Diskursreferenten kommen dafür in Frage.

Eine DRS B_2 ist (in B) einer DRS B_1 direkt untergeordnet, wenn

- $B_1 \Rightarrow B_2$ eine Bedingung in einer Teilstruktur von B ist, oder
- es eine DRS B_3 gibt, sodaß

$$\neg B_2, \quad B_2 \Rightarrow B_3, \quad B_2 \vee B_3, \quad \text{oder} \quad B_3 \vee B_2$$

eine Bedingung von B_1 ist.

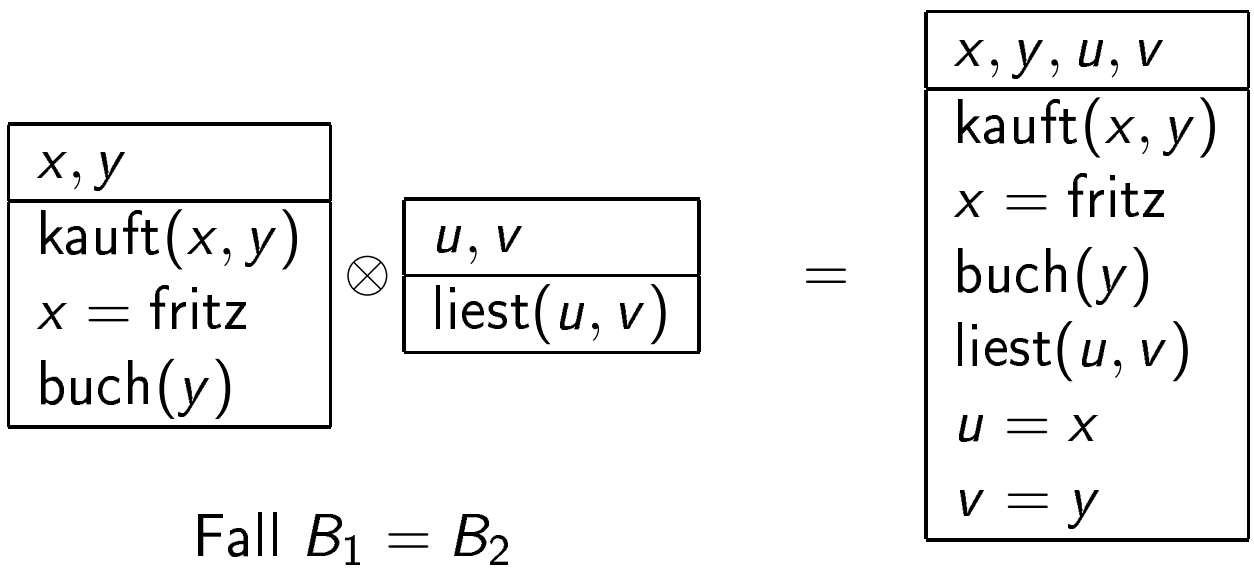
Die Unterordnung ist die transitive Hülle der direkten Unterordnung.

Erreichbarkeit

Eine Struktur B_1 ist von B_2 aus erreichbar (in B), wenn $B_1 = B_2$ ist oder wenn B_2 der Struktur B_1 untergeordnet ist (in B).

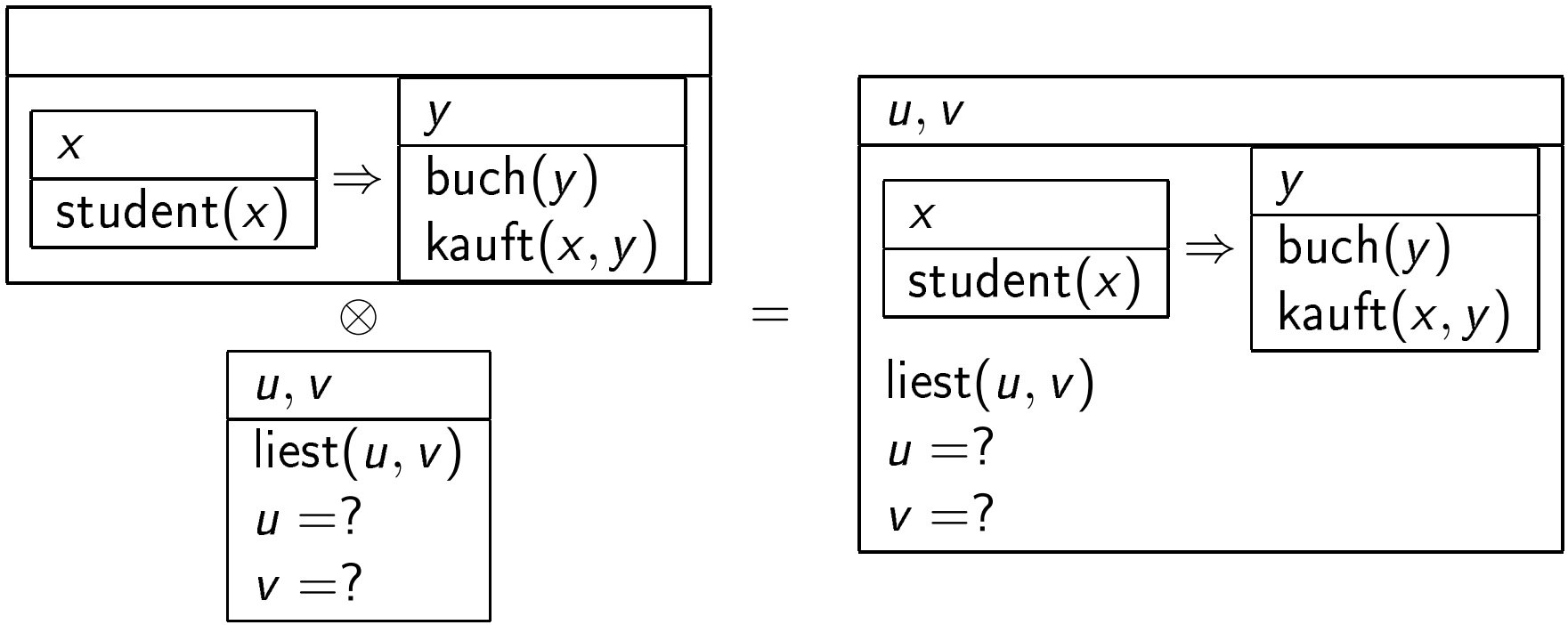
Seien $B_i = \langle D_i, C_i \rangle$ Teilstrukturen von B. Ein Diskursreferent y von D_2 kann zu einem anderen Referenten x aus D_1 aufgelöst werden (durch $y = x$ in C_2), wenn B_1 von B_2 aus erreichbar ist.

Beispiel a) Fritz kauft ein Buch. Er liest es.



Fall $B_1 = B_2$

Beispiel b) Jeder Student kauft ein Buch. Er liest es.



Die Pronomina sind nicht auflösbar, da es zu u, v keine erreichbaren anderen Diskursreferenten (von passendem Genus) gibt.

Übersetzung von DRT nach PL: B^{PL}

1. $\left(\begin{array}{|c|}\hline x_1, \ldots, x_n \\ \hline C_1, \ldots, C_k \\ \hline \end{array}\right)^{\mathrm{PL}} := \exists x_1 \ldots \exists x_n (C_1^{\mathrm{PL}} \wedge \ldots \wedge C_k^{\mathrm{PL}})$
2. $(\neg B)^{\mathrm{PL}} := \neg\, B^{\mathrm{PL}}$,
3. $(B_1 \vee B_2)^{\mathrm{PL}} := B_1^{\mathrm{PL}} \vee B_2^{\mathrm{PL}}$,
4. $\left(\begin{array}{|c|}\hline \vec{x} \\ \hline C_1, \ldots, C_k \\ \hline \end{array}\right) \Rightarrow B)^{\mathrm{PL}} := \forall \vec{x} (C_1^{\mathrm{PL}} \wedge \ldots \wedge C_k^{\mathrm{PL}} \rightarrow B^{\mathrm{PL}})$
5. $C^{\mathrm{PL}} := C$ für atomare Bedingungen $R(t_1, \ldots, t_n)$ und $t = s$.

Übersetzung von PL nach DRT: φ^{drt}

Voraussetzung: Als Terme treten nur Variable und Konstante auf.

Präpossessor: Forme die PL-Formel so um, daß als Junktoren und Quantoren nur $\wedge$, $\neg$ und $\exists$ auftreten, und jede Variable höchstens einmal gebunden wird.

Sei φ die gegebene Formel. Eliminiere alle $(\cdot)^{\mathsf{drt}}$ in $\boxed{\begin{array}{|c|}\hline \\ \hline \varphi^{\mathsf{drt}} \\ \hline \end{array}}$. nach folgenden Regeln:

1. $\boxed{\begin{array}{|c|}\hline \vec{x} \\ \hline \vec{C}, \psi^{\mathsf{drt}}, \vec{C}' \\ \hline \end{array}} \mapsto \boxed{\begin{array}{|c|}\hline \vec{x} \\ \hline \vec{C}, \psi, \vec{C}' \\ \hline \end{array}}$ für atomare Formeln ψ

2. $\boxed{\begin{array}{|c|}\hline \vec{x} \\ \hline \vec{C}, (\varphi \wedge \psi)^{\mathsf{drt}}, \vec{C}' \\ \hline \end{array}} \mapsto \boxed{\begin{array}{|c|}\hline \vec{x} \\ \hline \vec{C}, \varphi^{\mathsf{drt}}, \psi^{\mathsf{drt}}, \vec{C}' \\ \hline \end{array}}$

3. $\dfrac{\vec{x}}{\vec{C}, (\neg\varphi)^{\mathsf{drt}}, \vec{C}'} \mapsto \dfrac{\vec{x}}{\vec{C}, \neg\dfrac{}{\varphi^{\mathsf{drt}}}, \vec{C}'}$

4. $\dfrac{\vec{x}}{\vec{C}, (\exists y\varphi)^{\mathsf{drt}}, \vec{C}'} \mapsto \dfrac{\vec{x}, y}{\vec{C}, \varphi^{\mathsf{drt}}, \vec{C}'}$

Beispiel: $\varphi := \forall x(\mathsf{stern}(x) \rightarrow \mathsf{leuchtet}(x))$.

$$\frac{}{(\neg\exists x(\mathsf{stern}(x) \wedge \neg\mathsf{leuchtet}(x)))^{\mathsf{drt}}} \mapsto$$

$$\frac{}{\neg\dfrac{}{\exists x(\mathsf{stern}(x) \wedge \neg\mathsf{leuchtet}(x))^{\mathsf{drt}}}} \mapsto$$

$$\frac{}{\neg\dfrac{x}{(\mathsf{stern}(x) \wedge \neg\mathsf{leuchtet}(x))^{\mathsf{drt}}}} \mapsto^* \frac{}{\neg\dfrac{x}{\mathsf{stern}(x), \dfrac{}{\neg\mathsf{leuchtet}(x)}}}$$

Implementierung der Diskurse

Um eine Semantik mit Pronomenauflösung à la DRT zu implementieren, brauchen wir:

- Syntax: Satzfolgen als Diskurse, Personalpronomen (und Possessivpronomen?) als Nominalphrasen,
- Semantik: Umstellung der Bedeutungsterme auf λ-DRSen, neue `sem`-Klauseln für Pronomen und Diskurse, λ-DRS-Verschmelzung
- Typisierung: Anpassung der Typisierung auf λ-DRSen
- Pronomenauflösung: Suchen und Unifizieren (gem. Grammatik und Typen) passender Referenten

Die Pronomenauflösung ist also eine *Änderung der DRS des Diskurses*. Statt des Einbauens von Gleichungen verwenden wir die Unifikation von (getypten) Prologvariablen (= Diskursreferenten).

Erweiterung von Syntax und Lexikon

Erweiterung der Syntax um Diskurse, d.h. Folgen von Aussagen:

⟨*Grammatik/saetze.pl*⟩+≡

```
d([def]) -->
    s([def],[_Temp,ind,vz]),
    % ['.'], % wird im Parser/tokenizer.mini.pl enfernt
    d([def]).
d([def]) --> [].
```

Die Konstruktion ist rechtsrekursiv definiert: `d --> s,d`. Bei linksrekursiver Regel (`d --> d,s`) divergiert der Prolog- Parser.

Zum Parsen brauchen wir ein Startsymbol für Diskurse:

⟨*Neues Startsymbol in Grammatik/startsymbole.pl*⟩≡

```
startsymbol(d([def])).
```

Erweiterung des Lexikons um Personalpronomen:

⟨*Grammatik/lexikon_detpron.pl: Personalpronomen*⟩≡

```
pron([per],[mask,sg,nom]) --> [er].
pron([per],[mask,sg,dat]) --> [ihm].
pron([per],[mask,sg,akk]) --> [ihn].

pron([per],[fem, sg,nom]) --> [sie].
pron([per],[fem, sg,dat]) --> [ihr].
pron([per],[fem, sg,akk]) --> [sie].

pron([per],[mask,pl,nom]) --> [sie].
pron([per],[mask,pl,dat]) --> [ihnen].
pron([per],[mask,pl,akk]) --> [sie].

pron([per],[fem, pl,nom]) --> [sie].
pron([per],[fem, pl,dat]) --> [ihnen].
pron([per],[fem, pl,akk]) --> [sie].
```

Erweiterung des Lexikons um definite Possessivpronomen:

⟨*Grammatik/lexikon_detpron.pl: Possessivpronomen*⟩≡

```
poss([def(mask,sg)],[mask,sg,nom]) --> [sein].
poss([def(mask,sg)],[fem, sg,nom]) --> [seine].
poss([def(mask,sg)],[mask,sg,akk]) --> [seinen].
poss([def(mask,sg)],[fem, sg,akk]) --> [seine].

poss([def(fem,sg)],[mask,sg,nom]) --> [ihr].
poss([def(fem,sg)],[fem, sg,nom]) --> [ihre].
poss([def(fem,sg)],[mask,sg,akk]) --> [ihren].
poss([def(fem,sg)],[fem, sg,akk]) --> [ihre].

poss([def(mask,sg)],[mask,pl,nom]) --> [seine].
poss([def(mask,sg)],[fem, pl,nom]) --> [seine].
poss([def(mask,sg)],[mask,pl,akk]) --> [seine].
poss([def(mask,sg)],[fem, pl,akk]) --> [seine].
```

Definite Possessivpronomen im Plural:

⟨*Grammatik/lexikon_detpron.pl: Possessivpronomen*⟩+≡

```
poss([def(fem,sg)],[mask,pl,nom]) --> [ihre].
poss([def(fem,sg)],[fem, pl,nom]) --> [ihre].
poss([def(fem,sg)],[mask,pl,akk]) --> [ihre].
poss([def(fem,sg)],[fem, pl,akk]) --> [ihre].
```

Erweiterung der Nominalphrasen um definite Possessivpronomen (definite Personalpronomen sind in (`np --> pron`) erlaubt):

⟨*Grammatik/np.pl*⟩+≡

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

Hier werden neue Defintheitswerte `def(Gen,Num)` vergeben, die in den Satzregeln noch erlaubt werden müssen! (TODO)

Semantik mit λ-DRSen

Eine Diskursrepräsentationsstruktur implementieren wir durch einen Prologterm der Form

```
drs([X1,...Xn],[C1,...,Cm])
```

mit einer Liste von Prologvariablen (den Diskursreferenten) und einer Liste von Formeln (den Bedingungen).

Wir verpacken die neuen `sem/2`-Klauseln in einem Modul `drs`, damit wir beide Semantiken, die mit λ-Termen und die mit λ-DRSen, simultan laden können.

Zuerst modifizieren wir den Parse-Aufruf und die Baumausgabe.

Parsen und Baumausgabe mit DRS-Semantik

⟨*semantik.pl*⟩+≡

```
:- ['Semantik/semDrs.pl']. % Laden der DRS-Semantik sem/2

parsedrs :-
    write('Beende die Eingabe mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    startsymbol(S),
    addTree:translate(S,Term,[Baum]),
    addDifflists:translate(Term,ExpTerm,Atomlist,[]),
    nl,write('Aufruf: '), portray_clause(ExpTerm),
    call(ExpTerm), write('Baum:   '),
    nl,writeTrLamDrs(Baum,4),nl,
    fail.
parsedrs.
```

Hier nicht gezeigt: `parsedrs(+Datei)`, ebenfalls in `semantik.pl`.

Wir formatieren den Syntaxbaum und schreiben unter jeden Knoten seine mit drs:sem/2 berechneten Lambda-DRSen:

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLamDrs([Wurzel|Bs],Einrueckung) :-
    ( drs:sem([Wurzel|Bs],_)  % Ausdruck mit Bedeutungen
    ->  % schreibe alle Bedeutungen unter die Wurzel
        tab(Einrueckung),writeq(Wurzel),
        (Bs = [[Blatt]] -> tab(1), writeq(Blatt) ; true),
         (drs:sem([Wurzel|Bs],LamDrs),
         (nl,tab(Einrueckung),write('+ '),writen(LamDrs))
         fail
        ; true)
    ; % schreibe das bedeutungslose Wort
      tab(Einrueckung),writeq(Wurzel) ),
    ( Bs = [[Blatt]] -> true % schon geschrieben
    ; Einrueckung2 is Einrueckung + 3,
        writeTrsLamDrs(Bs,Einrueckung2) ).
```

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLamDrs([],Einrueckung) :-
    tab(Einrueckung), writeq([]).

writeTrsLamDrs([Baum|Baeume],Einrueckung) :-
    nl,writeTrLamDrs(Baum,Einrueckung),
    writeTrsLamDrs(Baeume,Einrueckung).
writeTrsLamDrs([],_).
```

Berechnung der λ-DRSen

Im Lexikon müssen wir bei den Nomen und Verben statt Formeln DRSen erzeugen, deren Referentenliste leer ist.

⟨*Semantik/semDrs.pl*⟩≡

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

% sem/2 mit lambda-DRS-Termen
:- op(500,yfx,&), op(600,yfx,'\/'), op(600,xfx,'=>').

kleinschreibung(Wort,Klein) :-
    downcase_atom(Wort,Klein).

% EN
sem([en(Art,Form), [Vollform]],LamTerm) :-
    !, wort(Stammform, en(Art,Form), Vollform),
    kleinschreibung(Stammform,EN),
    LamTerm = EN.
```

⟨*Semantik/semDrs.pl*⟩+≡

```
% VI
sem([v([nom],Form),[Vollform]],LamTerm) :-
    !, kleinschreibung(Vollform,VollformKl),
    wort(Stammform,v([nom],Form),VollformKl),
    Formel =.. [Stammform,X],
    LamTerm = lam(X,drs([],[Formel])).

% VT
sem([v([nom,akk],Form),[Vollform]],LamTerm) :-
    !, kleinschreibung(Vollform,VollformKl),
    wort(Stammform,v([nom,akk],Form),VollformKl),
    Formel =.. [Stammform,X,Y],
    LamTerm = lam(X,lam(Y,drs([],[Formel]))).
```

Nomen und Relationsnomen werden mit ant(X,Gen,Num) als mögliche Antezedenzien mit Genus und Numerus markiert:

⟨*Semantik/semDrs.pl*⟩+≡

```
% CN
sem([n([Gen],[Num,Kas]),[Vollform]],LamTerm) :-
    !, wort(Stammform, n([Gen],[Num,Kas]), Vollform),
    kleinschreibung(Stammform,Stamm),
    Formel =.. [Stamm,X],
    LamTerm = lam(X,drs([],[ant(X,Gen,Num),Formel])).

% RN
sem([rn([Gen],[Num,Kas]), [Vollform]],LamTerm) :-
    !, wort(Stammform, rn([Gen],[Num,Kas]), Vollform),
    kleinschreibung(Stammform,Stamm),
    Formel =.. [Stamm,X,Y],
    LamTerm = lam(X,lam(Y,drs([],[Formel,
                                  ant(X,Gen,Num)])))).
```

Relationsnomen als Nomen verwendet:

⟨*Semantik/semDrs.pl*⟩+≡

```
sem([n(Art,Form),[rn(Art,Form),[Vollform]]],LamTerm) :-
    !, sem([rn(Art,Form),[Vollform]],SemRN),
    normalize(SemRN * X * Y, Drs),
    LamTerm = lam(X,drs([Y],[])+Drs).
```

Bei den Determinatoren werden die entsprechenden DRSen erzeugt, wobei + für die Verschmelzung (merge) von DRSen steht:

⟨*Semantik/semDrs.pl*⟩+≡

```
% DET = every % distributive sg-Lesart; 2.pl-Lesart unter
sem([det([quant],_Form),_Vollform],
    lam(N,lam(P,drs([],[((drs([X],[]) + N*X) => P*X)]))))

% DET = a
sem([det([indef],_Form),_Vollform], % 2.pl-Lesart unten
    lam(N,lam(P, drs([X],[]) + N*X + P*X))).

% DET = the
sem([det([def],[_,sg,_]),_Vollform],
    lam(N,lam(P, drs([Y],[(drs([X],[]) + N*X)
                          => drs([],[eq(X,Y)])])
                 + N*Y + P*Y))) :- !.
```

Für Determinatoren im Plural werden auch Lesarten mit Paarquantoren, resp. Paaren von Diskursrepräsentanten, erzeugt:

⟨*Semantik/semDrs.pl*⟩+≡

```
% DET = alle % Paar-Lesart mit symmetrischem P
sem([det([quant],[_,pl,_]),_Vollform],
    lam(N,lam(P,drs([],[((drs([X,Y],[]) + N*X + N*Y)
                         => P*X*Y)])))) :- !.
% DET = einige
sem([det([indef],[_,pl,_]),_Vollform],
    lam(N,lam(P, drs([X,Y],[]) + N*X + N*Y + P*X*Y))) :-
% DET = die
sem([det([def],[_,pl,_]),_Vollform],
    lam(N,lam(P, drs([Y1,Y2],
                     [(drs([X1,X2],[]) + N*X1 + N*X2)
                      => drs([],[eq(X1,Y1),eq(X2,Y2)])])
                + N*Y1 + N*Y2 + P*Y1*Y2))) :- !.
```

Problem: Sollten wir nicht die Klauseln für Funktionswörter beibehalten und erst bei der Normalisierung in DRSen umformen?

⟨*Normalisierung von Quantoren*⟩≡

```
normalize(all(X,Restr,Fml),
          drs([],[drs([X],[])+RestrNf => FmlNf])) :-
  normalize(Restr,RestrNf),
  normalize(Fml,FmlNf).
```

⟨*Semantik/semDrs.pl*⟩+≡

```
% TODO: drs?, sg/pl-Lesarten?
sem([det([qu], _Form),[Vollform]],
    lam(N,lam(P,qu(X,N*X & P*X)))) :-
    concat_atom(['welche'|_],Vollform), !.
```

⟨*Semantik/semDrs.pl*⟩+≡

```
sem([det([def],[_,pl,_]),[Vollform]],
    lam(N,lam(P, drs([],[anzahl(X,N*X & P*X,Zahl)])))) :-
    anzahl(Vollform,Zahl), !.
```

TODO: Überprüfen und testen:

- die Anzahldeterminatoren (Plural mit Paaren?)
- die Pluralsemantik der Determinatoren mit Paaren von Referenenten.

Definite Personalpronomen erhalten als Bedeutung eine Abstraktion von Individueneigenschaften eines neuen Diskursreferenten, der für die Pronomenauflösung als Anapher markiert wird:

⟨*Semantik/semDrs.pl*⟩+≡

```
% PRON [per]
sem([pron([per],[Gen,Num,_Kas]),_Pron],
    lam(P, drs([X],[anp(X,Gen,Num)]) + P*X)) :- !.
```

Relativ- und Interrogativpronomen sind noch zu überlegen:

⟨*Semantik/semDrs.pl*⟩+≡

```
% TODO
sem([pron([rel],_Form),_Rel], lam(P,lam(X,P*X))) :- !.
sem([pron([qu],_Form),_Pron], lam(P,qu(X,P*X))) :- !.
```

Possessivpronomen erhalten ebenfalls einen Diskursreferenten; er wird als possessive Anapher markiert:

⟨*Semantik/semDrs.pl*⟩+≡

```
% POSS (ohne Eindeutigkeitsbedingung) (HL)
sem([poss([def(Gen2,Num2)],_Form),_Vollform],Sem):-
    !, Sem = lam(RN,lam(P, drs([X,Y],
                               [anposs(Y,Gen2,Num2)])
                           + RN*X*Y + P*X)).
```

TODO:

- in der Syntax nachtragen: `np([def(Gen,Num)],Form)` als Subjekt und Objekt
- in der Semantik nachtragen: `sem/2` für `(np --> poss, rn)`.

Reflexiv- und Reziprokpronomen: die Klauseln sind wie bei der λ-Semantik, nur konstruiert P jetzt eine DRS, keine Formel.

⟨*Semantik/semDrs.pl*⟩+≡

```
sem([pron([refl],_Form),[sich]],
    lam(P,lam(X, P*X*X))) :- !.

sem([pron([rezi],_Form),[einander]],
    lam(P,lam(X,lam(Y, P*X*Y + P*Y*X)))) :- !.
```

Bei den Grammatikregeln wird die Bedeutung oft analog zu der mit reinen Lambda-Termen konstruiert.

⟨*Semantik/semDrs.pl*⟩+≡

```
% NP = DET+CN
sem([np([_,3,_,_],[_]),
        Det, [n(Art,Form),Vf]], SemNP) :-
    sem(Det,SemDet), Vf \= km,
    !, sem([n(Art,Form),Vf],SemN),
    normalize(SemDet * SemN,SemNP).
```

Eigennamen als Nominalphrasen: sie abstrahieren von Eigenschaften eines Diskursreferenten, der als Antezedens markiert wird:

⟨*Semantik/semDrs.pl*⟩+≡

```
% NP = EN (mit neuem Diskursreferenten)
sem([np([def,3,Gen,Num],[Kas]),
        [en([Gen],[Num,Kas]),EN]], SemNP) :-
    !, sem([en([Gen],[Num,Kas]),EN],SemEN),
    SemNP = lam(P,drs([X],[eq(X,SemEN),ant(X,Gen,Num)])
                    + P*X).
% NP = DET EN
sem([np([def,3,Gen,Num],[_Kas]),
        [det([def],_),_],
        [en([Gen],[Num,nom]),EN]],
    SemNP) :-
    !, sem([np([def,3,Gen,Num],[nom]),
            [en([Gen],[Num,nom]),EN]], SemNP).
```

Ein Nomenattribut beim Eigennamen wird in die Bedingung eingebaut; es führt schon eine Antezedensmarkierung ein:

⟨*Semantik/semDrs.pl*⟩+≡

```
% NP = DET+N+PN
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, drs([X],[eq(X,SemEN)])
                        + SemN*X + P*X), SemNP).
```

Pronomen als Nominalphrasen: sie haben die Pronomenbedeutung

⟨*Semantik/semDrs.pl*⟩+≡

```
% NP = PRON (personal, reflexiv, reziprok, interrogativ)
sem([np([def,3,_Gen,_Num],[_Kas]),
        [pron([per],Form),Pron]], SemPron) :-
    !, sem([pron([per],Form),Pron],SemPron).

sem([np([DefP,3,Gen,Num],[Kas]),
      [pron([DefP],[Gen,Num,Kas]),Pron]], SemNP) :-
    (DefP = rezi; DefP = refl; DefP = qu),
    !, sem([pron([DefP],[Gen,Num,Kas]),Pron], SemNP).

sem([np([Def,3,Gen,Num],[Kas]),
      [pron([DefP],[Gen,Num,Kas]),Pron]], SemNP) :-
    DefP = rel, Def = rel(Gen,Num),
    !, sem([pron([DefP],[Gen,Num,Kas]),Pron], SemNP).
```

⟨*Semantik/semDrs.pl*⟩+≡

```
% NP = POSS+RN
sem([np([def(Gen2,Num2),3,Gen,Num],[Kas]),
     [poss([def(Gen2,Num2)],[Gen,Num,Kas]),Poss],
     [rn([Gen],[Num,Kas]),RN]], SemNP) :-
    !, sem([poss([def(Gen2,Num2)],
                 [Gen,Num,Kas]),Poss], SemPoss),
    sem([rn([Gen],[Num,Kas]),RN], SemRN),
    normalize(SemPoss * SemRN, SemNP).

% NP = Zahl km (TODO: testen)
sem([np([_,3,_,_],[_]),
        [det([def],_),[Zahlatom]],[n(_,_),[km]]],
    lam(P,drs([],[])+P*Zahl)) :-
    anzahl(Zahlatom,Zahl), !. % Ausnahme: Maß
```

⟨*Semantik/semDrs.pl*⟩+≡

```
% NP = DET RN NPgen
sem([np([_,3,_,_],[_]),
       Det,[rn(Art,Form),RN],NPgen], Sem) :-
    NPgen = [np([_,3,_,_],[gen])|_],
    !, 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).
```

Satzregeln:

Sätze mit intransitivem Verb in Verbzweitstellung:

⟨*Semantik/semDrs.pl*⟩+≡

```
% S = NP+VI
sem([s([_Def],[_,_,vz]), NP, VI], SemS) :-
    NP = [np(_,_)|_], VI = [v([nom],_)|_],
    !, sem(NP,SemNP),
    sem(VI,SemVI),
    Sem = SemNP * SemVI,
    normalize(Sem,SemS).
```

Sätze mit transitivem Verb in Verbzweitstellung:

⟨*Semantik/semDrs.pl*⟩+≡

```
sem([s([_Def],[_,_,vz]),
       NP1, [v([nom,akk],[3,Num,_,ind]),V], NP2], SemS)
:-  NP1 = [np([_,3,_,_],[Kas1])|_],
    NP2 = [np([Def2,3,_,_],[_])|_],
    !, sem(NP1,SemNP1), sem(NP2,SemNP2),
    sem([v([nom,akk],[3,Num,_,ind]),V], SemV),
    (Kas1 = nom,
        ( (Def2 = rezi ; Def2 = refl)
        -> Sem = SemNP1 * (SemNP2 * SemV)
        ; Sem = SemNP1 * lam(X,SemNP2 *
                                lam(Y,SemV * X * Y)))
    ; Kas1 = akk, % ausser: rezi, refl-Pronomen
        Sem = SemNP1 * lam(Y,SemNP2 *
                              lam(X,SemV * X * Y))
    ), normalize(Sem,SemS).
```

Die Normalisierung muß noch so verbessert werden, daß sie Verschmelzungen `drs(Ref1,Fml1)+drs(Ref2,Fml2)` ausführt.

⟨ *Beispiel* ⟩+≡

```
?- parsedrs. der Uranus umkreist einen Stern.
Baum:
    s([def], [praes, ind, vz])
    + drs([X], [eq(X, uranus), ant(X, mask, sg)])+
       (drs([Y], [])+drs([], [ant(Y, mask, sg), stern(Y)]
        +drs([], [umkreisen(X, Y)]))
       np([def, 3, mask, sg], [nom])
       + lam(X, drs([Y], [eq(Y, uranus), ant(Y,..)])+X*Y)
       ...
       v([nom, akk], [3, sg, praes, ind]) umkreist
       + lam(X, lam(Y, drs([], [umkreisen(X, Y)])))
       np([indef, 3, mask, sg], [akk])
       + lam(X, drs([Y], [])+drs([], [ant(Y, mask, sg),
                                      stern(Y)])+X*Y)
       ...
```

⟨*Semantik/semDrs.pl*⟩+≡

```
sem([s([rel(GenR,NumR)],[Tmp,Mod,vl]),
       NP1, NP2, [v([nom,akk],[3,Num,Tmp,Mod]),V]],
     SemS)
:-
     NP1 = [np([rel(GenR,NumR),3,GenR,_],[Kas1])|_],
     NP2 = [np([Def2,3,_,_],[_])|_],
     !, sem(NP1,SemNP1), sem(NP2,SemNP2),
     sem([v([nom,akk],[3,Num,Tmp,Mod]),V], SemV),
     ( Kas1 = nom, ((Def2 = refl ; Def2 = rezi)
        -> Sem = SemNP1 * (SemNP2 * SemV)
        ; 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).
```

Nominalphrasen mit Relativsätzen:

⟨*Semantik/semDrs.pl*⟩+≡

```
% NP = DET N RelS
sem([np([_,3,_,_],[_]),
        Det,[n(Art,Form),N],RelS], Sem) :-
    RelS = [s([rel(_,_)],_)|_],
    !, sem(Det,SemDet),
    sem([n(Art,Form),N],SemN),
    sem(RelS,SemRelS),
    SemNS = lam(X, drs([],[]) + SemN*X + SemRelS*X),
    normalize(SemDet*SemNS,Sem).
```

Analog für `NP = Det RN NPgen RelS`

Schließlich die Regeln für die Bedeutung von Diskursen:

⟨*Semantik/semDrs.pl*⟩+≡

```
% D = S D
sem([d([def]),S,D], Sem) :-
    !, sem(S,SemS),
    sem(D,SemD),
    ( debugging(resolve)
    -> resolveDrs:resolve(SemS,SemD,Sem) % später
    ; normalize(SemS + SemD, Sem)
    ).

% D = [] (Diskursende)
sem([d([def])], drs([],[])) :- !.
```

Vorläufig wird nur SemS+SemD normalisiert; Die Pronomenauflösung wird später mit `?- (no)debug(resolve).` ein(aus)-geschaltet.

⟨*Beispiel eines 2-Satz-Diskurses (ohne Verschmelzungen)*⟩≡

```
?- parsedrs. ein Astronom entdeckt Uranus. er leuchtet.
Baum:
    d([def])
    + drs([X], [])+drs([], [ant(X,mask,sg),astronom(X)])
      +(drs([Y], [eq(Y,uranus), ant(Y,mask,sg)])
       +drs([], [entdecken(X,Y)]))+(drs([Z], [anp(Z,mask,
         +drs([], [leuchten(Z)])+drs([], []))
       s([def], [praes, ind, vz])
       + drs([X], [])+drs([], [ant(X,mask,sg),astronom(X)
         +(drs([Y], [eq(Y, uranus), ant(Y, mask, sg)])
          +drs([], [entdecken(X, Y)]))
          ...
       d([def])
       + drs([X], [anp(X, mask, sg)])
         +drs([], [leuchten(X)])+drs([], [])
          ...
          d([def]) + drs([], [])
```

Verschmelzung der DRSen

Beim Normalisieren der DRS-Bedeutungen sollen Verschmelzungen `drs(R1,F1)+drs(R2,F2)` schon ausgeführt werden. Beachte:

- eine Verschmelzung `(DRS1 + DRS2)` erzeugt keine β-Redexe
- also können alle β-Reduktionen *vor* allen Verschmelzungen ausgeführt werden.

Wir benutzen in `drs` ein umdefiniertes `normalize/2`:

⟨*Semantik/semDrs.pl*⟩+≡

```
normalize(Term,Reduced):-
    lambdaDrs:normalize(Term,Reduced).
```

und laden ab jetzt für die Semantik auch `lambdaDRS.pl`:

⟨*semantik.pl*⟩+≡

```
% DRS-Semantik: normalisieren und mergen
:- ['Semantik/lambdaDrs.pl'].
```

Die neue Definition wird in einem neuen Modul implementiert:

⟨*Semantik/lambdaDrs.pl*⟩≡

```
:- module(lambdaDrs,[reduceTerm/2,mergeTerm/2]).
:- op(500,yfx,&), op(600,yfx,'\/'), op(600,xfx,'=>').

normalize(Term,MergedTerm):-
    reduceTerm(Term,ReducedTerm),
    mergeTerm(ReducedTerm,MergedTerm).
```

Hier ist `reduceTerm/2` das bisherige `normalize/2` und `mergeTerm/2` die Verschmelzung von DRSen:

⟨*Semantik/lambdaDrs.pl*⟩+≡

```
reduceTerm(T,Nf) :-
    beta(T,ConT,Tag),
    ( Tag = chd -> reduceTerm(ConT,Nf) ; Nf = ConT ).
```

Vorsicht: `beta/3` ruft `normalize/2` auf, nicht `reduceTerm/2` !???

⟨*Semantik/lambdaDrs.pl*⟩+≡

```
reduceTerms([T|Ts],[Nf|Nfs]) :-
    reduceTerm(T,Nf), reduceTerms(Ts,Nfs).
reduceTerms([],[]).

% mergeTerm(+Term,-mergedTerm)

mergeTerm(T,Nf) :-
    mergeDrs(T,MT,Tag),
    ( Tag = chd -> mergeTerm(MT,Nf) ; Nf = MT ).

mergeTerms([T|Ts],[Nf|Nfs],Tag) :-
    mergeDrs(T,MT,TagT),
    ( TagT = chd -> mergeTerm(MT,Nf) ; Nf = MT ),
    mergeTerms(Ts,Nfs,TagTs),
    ( TagT = not -> Tag = TagTs ; Tag = chd ).
mergeTerms([],[],not).
```

Das Verschmelzen von DRSen erfolgt rekursiv nach ihrer Form:

⟨*Semantik/lambdaDrs.pl*⟩+≡

```
% mergeDrs(+ReducedTerm, -mergedTerm, -geaendert?)

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

mergeDrs(T*S, Mrg, Tag) :-
    !, mergeDrs(T,ConT,TagT),
    ( TagT = chd -> Mrg = ConT*S, Tag = chd
    ; mergeDrs(S,ConS,Tag), Mrg = T*ConS ).

mergeDrs(lam(X,D),lam(X,MD),Tag):-
    !, mergeDrs(D,MD,Tag).

mergeDrs(drs(D,F),drs(D,MF),Tag):-
    !, mergeTerms(F,MF,Tag).
```

⟨*Semantik/lambdaDrs.pl*⟩+≡

```
mergeDrs(neg(D),neg(MD),Tag):-
    !, mergeDrs(D,MD,Tag).

mergeDrs((D1 => D2),Mrg,Tag):-
    !, mergeDrs(D1,MD1,Tag1),
    ( Tag1 = chd -> Mrg = (MD1 => D2), Tag = chd
    ; mergeDrs(D2,MD2,Tag),Mrg = (D1 => MD2)).

mergeDrs(or(D1,D2),Mrg,Tag):- % TODO: \/ statt or
    !, mergeDrs(D1,MD1,Tag1),
    ( Tag1 = chd -> Mrg = or(MD1,D2), Tag = chd
    ; mergeDrs(D2,MD2,Tag),Mrg = or(D1,MD2)).
```

Bem: es gibt noch keine (linksrekursive!) Regel `(s -> s oder s)`.

Falls bei (DRS1 + DRS2) zwei konkrete DRSen vorliegen, werden sie verschmolzen; andernfalls behandelt man rekursiv die Teile DRSi:

⟨*Semantik/lambdaDrs.pl*⟩+≡

```
mergeDrs((drs(R1,F1) + drs(R2,F2)),drs(R,F),chd):-
    !, append(R2,R1,R), append(F2,F1,F). % (Wu)
    % !, append(R1,R2,R), append(F1,F2,F). % (HL)

mergeDrs((D1 + D2), Mrg, Tag):-
    !, mergeDrs(D1,MD1,Tag1),
    ( Tag1 = chd -> Mrg = (MD1 + D2), Tag = chd
    ; mergeDrs(D2,MD2,Tag), Mrg = (D1 + MD2) ).

mergeDrs(F,F,not):-!. % einfache Bedingung
```

Bei der Verschmelzung wurde die Formelreihenfolge invertiert, um bei der Pronomenauflösung die Antezedens-NP im Text nach links laufend zu suchen (anders für satzinterne Possessivauflösung HL?).

⟨*Beispiel eines 2-Satz-Diskurses (Verschmelzung ausgeführt)*⟩≡

```
?- parsedrs. ein Astronom entdeckt Uranus. er leuchtet.
Baum:
    d([def])
    + drs([X, Y, Z],
          [leuchten(X), anp(X, mask, sg),
           entdecken(Z, Y), eq(Y, uranus), ant(Y,mask,sg)
           ant(Z, mask, sg), astronom(Z)])
       s([def], [praes, ind, vz])
       + drs([X, Y],
             [entdecken(Y,X), eq(X,uranus), ant(X,mask,sg
              ant(Y, mask, sg), astronom(Y)])
          ...
       d([def])
       + drs([X], [leuchten(X), anp(X, mask, sg)])
          ...
          d([def])
          + drs([], [])
```

Typisierung der DRSen

Die Pronomenauflösung `Pron` $\rightsquigarrow$ `NP` soll die Typen der Pronomen und Nominalphrasen berücksichtigen, um semantisch unsinnige (hier: nicht typkorrekte) Auflösungen auszuschließen.

- jedes Pronomen*vorkommen* erhält einen eigenen Referenten.
- ein Pronomen an einer Verbargumentstelle muß zum Typ des Verbs passen: daher erhält der Referent des Pronomenvorkommens den Argumenttyp des Verbs.
- jede mögliche Bezugsnominalphrase erhält durch ihre Konstruktion einen Typ; dabei auch der Referent in ihrer Antezedensmarkierung.

Die Pronomenauflösung sucht dann zu `anp(X:TyX,GenX,NumX)` ein Antezedens `ant(Y:TyY,GenY,NumY)`, unifiziert beide und entfernt die Anaphermarkierung.

Jetzt implementieren wir zuerst die Typisierung der DRSen.

Eine *getypte* DRS sollte die Form

$x_1 : \sigma_1, \ldots, x_n : \sigma_n$
$\varphi_1 : t, \ldots, \varphi_m : t$

haben: eine Liste von Referenten mit Typen und eine Liste von Formeln (Termen vom Typ `t` = `bool`; die Typen freier Variablen müssen[1] einem Typkontext

$$\Gamma = y_1 : \rho_1, \ldots, y_k : \rho_k$$

entnommen werden. Als Typ der DRS nehmen wir die das Paar

$$\langle [\sigma_1, \ldots, \sigma_n], t \rangle$$

aus den Typen der Referenten und dem Typ der Konjunktion der Bedingungen. Solche Paare nennen wir DRS-Typen.

[1] sofern man nicht mit getypten Variablen y^σ arbeitet

Die Typrekonstruktion sollte die ungetypten DRS so typisieren:

$$\Gamma \vdash \boxed{\begin{array}{|c|}\hline x_1, \ldots, x_n \\ \hline \varphi_1, \ldots, \varphi_m \\ \hline \end{array}} : \langle [\sigma_1, \ldots, \sigma_n], t \rangle \qquad (1)$$

Im Typ werden also nur die Typen der Top-Level-Referenten vermerkt. Aus Γ und dem Typ der DRS sollten sich die Typen der Referenten in Teil-DRSen ermitteln lassen.

Beachte, daß die typisierten Referenten selber einen Typkontext bilden: in den Bedingungen sollen die Referenten von dem Typ sein, der in der Referentenliste genannt wird: aus (1) soll folgen:

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n, \Gamma \vdash \varphi_i : t, \quad \text{für } i = 1, \ldots, n.$$

Die Typisierung von DRSen wird nun implementiert durch

```
type(+Typkontext,+DRS,-GetypteDRS,-Typ).
```

Zum Parsen dient parsedrst/0, ein Parseaufruf mit Ausgabe der aus dem Syntaxbaum mit `drs:sem/2` berechneten DRS und ihrer Typisierung.

⟨*semantik.pl*⟩+≡

```
% Typisierung von (merge-freien) Lambda-DRSen
:- ['Semantik/typeDrs.pl'].
```

⟨*semantik.pl*⟩+≡

```
parsedrst :-
    write('Beende die Eingabe mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    tokenize(Sentence,Atomlist),
    startsymbol(S),
    addTree:translate(S,Term,[Baum]),
    addDifflists:translate(Term,ExpTerm,Atomlist,[]),
    nl,write('Aufruf: '), portray_clause(ExpTerm),
    call(ExpTerm), write('Baum:   '),
    nl,writeTrLamDrsTy(Baum,4),nl,
    fail.
parsedrst.
```

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLamDrsTy([Wurzel|Bs],Einrueckung) :-
    ( drs:sem([Wurzel|Bs],_)  % Ausdruck mit Bedeutungen
    ->  % schreibe alle Bedeutungen unter die Wurzel
        tab(Einrueckung),writeq(Wurzel),
        (Bs = [[Blatt]] -> tab(1), writeq(Blatt) ; true),
        (drs:sem([Wurzel|Bs],LamTerm),
            drs:type([],LamTerm,LamTermGetypt,Typ),
            Term = (LamTermGetypt:Typ),
            (nl,tab(Einrueckung),write('+ '),writen(Term)
            fail
        ; true)
    ; % schreibe das bedeutungslose Wort
      tab(Einrueckung),writeq(Wurzel) ),
    ( Bs = [[Blatt]] -> true % schon geschrieben
    ; Einrueckung2 is Einrueckung + 3,
        writeTrsLamDrsTy(Bs,Einrueckung2) ).
```

Es fehlen noch die Endklausel und das Schreiben einer Liste von Sub-DRSen:

⟨*Parser/showTree.pl*⟩+≡

```
writeTrLamDrsTy([],Einrueckung) :-
    tab(Einrueckung), writeq([]).

writeTrsLamDrsTy([Baum|Baeume],Einrueckung) :-
    nl,writeTrLamDrsTy(Baum,Einrueckung),
    writeTrsLamDrsTy(Baeume,Einrueckung).
writeTrsLamDrsTy([],_).
```

Wie bei der Normalisierung definieren wir die Typisierung um

⟨*Semantik/semDrs.pl*⟩+≡

```
% type(+TypKontext,+Drs,-DrsGetypt,-Typ) :-
type(Kontext,Drs,DrsGetypt,Typ) :-
    typeDrs:type(Kontext,Drs,DrsGetypt,Typ).
```

und implementieren sie in einer neuen Moduldatei `typeDrs.pl`.

Siehe dazu: `Semantik/typeDrs.pl`.

Die Typregeln zum Typisieren von Variablen und Merge-DRSen:

⟨*Auszug aus Semantik/typeDrs.pl*⟩≡

```
% drs2
type(Kontext,drs([X|R],Fs),drs([(X:Xtyp)|RT],FsT),
     ([Xtyp|Rtyp],t)):-
    var(X), !,
    type([X:Xtyp|Kontext],drs(R,Fs),drs(RT,FsT),(Rtyp,t)).

% merge (+)
type(Kontext,(D1 + D2),(D1T + D2T),((Typ1+Typ2),t)) :-
    !, type([D2:(Typ2,t)|Kontext],D1,D1T,(Typ1,t)),
    type([D1:(Typ1,t)|Kontext],D2,D2T,(Typ2,t)).
```

Ein DRS-Typ hat die Form `(Typ der Refs,t)`.

Bei `(D1+D2)` werden die `Di` wechselseitig als Typkontexte benutzt.

⟨*Beispiel einer getypten DRS*⟩≡

```
?- parsedrst. jeder Stern leuchtet.
Baum:   % Klammern: (Y->Z, t) = (Y -> (Z,t)) usw.
    s([def], [praes, ind, vz])
    + drs([], [drs([X:s], [ant(X:s, mask, sg), stern(X)])
                =>drs([], [leuchten(X)])]): ([], t)
       np([quant, 3, mask, sg], [nom])
       + lam(X: (s->Y, t),
             drs([], [drs([Z:s], [ant(Z:s, mask, sg),
                                  stern(Z)])=>X*Z]))
          det([quant], [mask, sg, nom]) jeder
          + lam(X: (Y->Z, t), lam(A1: (Y->B1, t),
               drs([], [drs([C1:Y], [])+X*C1=>A1*C1])))
          n([mask], [sg, nom]) 'Stern'
          + lam(X:s,drs([],[ant(X:s,mask,sg),stern(X)]))
       v([nom], [3, sg, praes, ind]) leuchtet
       + lam(X:s, drs([], [leuchten(X)])): (s->[], t)
```

⟨*Beispiel* ⟩+≡

```
?- parsedrst. er entdeckte ihn.
Baum:
    s([def], [praet, ind, vz])
    + drs([X:s, Y:m], [entdecken(Y, X), anp(X:s, mask, sg
                       anp(Y:m, mask, sg)]): ([s, m], t)
       np([def, 3, mask, sg], [nom])
       + lam(X: (Y->Z, t),
             drs([A1:Y],[anp(A1:Y, mask, sg)])+X*A1)
          : ((Y->Z, t)->[Y]+Z, t)
          pron([per], [mask, sg, nom]) er
          + ... : ((Y->Z, t)->[Y]+Z, t)
       v([nom, akk], [3, sg, praet, ind]) entdeckte
       + lam(X:m, lam(Y:s, drs([], [entdecken(X, Y)])))
         : (m->s->[], t)
       np([def, 3, mask, sg], [akk])
       + ... : ((Y->Z, t)->[Y]+Z, t)
```

Die Typisierung gibt jedem Diskursreferenten einen Typ. Die Typen müssen bei der Pronomenauflösung `Pron` $\rightsquigarrow$ `NP` passen: `X:s` $\not\rightsquigarrow$ `Z:m`

⟨*Beispiel eines Diskurses, getypt*⟩≡

```
?- parsedrst. ein Astronom entdeckt Uranus. er leuchtet.
Baum:
  d([def])
  + drs([X:s, Y:s, Z:m],
        [leuchten(X), anp(X:s,mask,sg), entdecken(Z,Y),
         eq(Y, uranus), ant(Y:s, mask, sg),
         ant(Z:m, mask, sg), astronom(Z)]): ([s,s,m],t)
    s([def], [praes, ind, vz])
    + drs([X:s, Y:m],
          [entdecken(Y,X), eq(X,uranus), ant(X:s,mask,sg
           ant(Y:m,mask,sg), astronom(Y)]): ([s,m],t)
    d([def])
    + drs([X:s],
          [leuchten(X), anp(X:s, mask, sg)]): ([s], t)
      ...
```

Pronomenauflösung

Wir beschränken uns auf *anaphorische* Pronomenauflösung: die Bezugsnominalphrase muß im *Vortext* liegen. Wir wollen

- Personalpronomen gegen NPs in vorangegangenen Sätzen,
- Possessivpronomen gegen vorangegangene NPs im selben Satz

auflösen und die Erreichbarkeitsbedingungen der DRT beachten.

In der DRS eines Diskurses muß die Pronomenauflösung

- zu einer Anapher `anp(X:TyX,GenX,NumX)` ein erreichbares Antezendens `ant(Y:TyY,GenY,NumY)` im Vortext suchen,
- sie durch `(X:TyX,GenX,NumX)=(Y:TyY,GenY,NumY)` unifizieren und die Anaphermarkierung und Duplikate der Referenten in der DRS entfernen.

Wir verwenden eine neue Ladedatei:

⟨*anaphern.pl*⟩≡

```
% Anaphorische Pronomenauflösung
:- [grammatik],  % Tokenisierung und Grammatik
   [astronomie], % Beispiellexikon, Datenbank
   [semantik],   % LambdaTerm-Semantik, zum Vergleich
   ['Semantik/semDrs.pl',    % LambdaDrs-Semantik
    'Semantik/lambdaDrs.pl', % Reduktion + Mergen
    'Semantik/typeDrs.pl',   % DRS-Typisierung
    'Semantik/resolveDrs.pl' % Pronomenauflösung
   ].
```

Eine Satzfolge wurde mit `(d --> s, d)` rechtsrekursiv konstruiert, damit der Parser nicht divergiert. Für die Pronomenauflösung (im Linkskontext!) wäre es besser, wenn Diskurse mit `(D --> D, S)` linksrekursiv aufgebaut wären; dann könnte man

- im getypten Diskursanfang `D` die Pronomen intern auflösen,
- dann den nächsten Satz `S` analysieren und typisieren,
- und anschließend die Pronomen von `S` in `D` auflösen.

Um die Pronomenauflösung linksrekursiv vornehmen zu können, invertieren wir beim Einlesen die Reihenfolge der Sätze: statt `D = S1 ... Sn` analysieren wir `D' = Sn ... S1` mit `(s --> d,s)`.

Beim Lesen gibt `tokenizer:read_discourse/4` die invertierte Satzreihe aus, die dann der Parser als Diskurs analysieren soll:

⟨*anaphern.pl*⟩+≡

```
parsedrsr :-
    write('Beende die Eingabe mit <Punkt><Return>'),
    nl,tokenizer:read_discourse(user,[],[],Discourse),
    tokenize(Discourse,Atomlist),
    startsymbol(d([def])),
    addTree:translate(d([def]),Term,[Baum]),
    addDifflists:translate(Term,TermExp,Atomlist,[]),
    nl,write('Aufruf: '), portray_clause(TermExp),
    call(TermExp),
    write('Baum:   '), nl, writeTrLamDrs(Baum,8),nl,
    fail.
parsedrsr.
```

Interaktiv wird durch

⟨*Ein- bzw. Ausschalten der Pronomenauflösung*⟩≡

```
?- debug(resolve).   bzw. nodebug(resolve).
```

der Test `debugging(resolve)` auf `true` bzw. `false` gesetzt und die Pronomenauflösung der Diskurssemantik eingeschaltet:

⟨*Auszug aus Semantik/semDrs.pl*⟩≡

```
% D = S D
sem([d([def]),S,D], Sem) :-
    !, sem(S,SemS), sem(D,SemD),
    ( debugging(resolve)
    -> resolveDrs:resolve(SemS,SemD,Sem)
    ; normalize(SemS + SemD, Sem)
    ).
```

Wegen der Invertierung ist hier `S` der *letzte* Satz, `D` der *Anfang*.

`resolve(+SemS,+SemD,-Sem)` typisiert den letzten Satz und lösen seine Pronomina gegen die Nominalphrasen des Diskursanfangs auf:

⟨*Semantik/resolveDrs.pl*⟩≡

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

resolve(SemS,SemD,Sem) :-
    drs:type([],SemS,GetyptS,_TypS),
    reverse_drs(SemD,SemDrev),
    resolve_drs([seq(SemDrev,GetyptS)],[SemRev]),
    reverse_drs(SemRev,Sem).

% :- debug(resolve).  % Pronomenauflösung einschalten.
```

Die Reihenfolge der DRS des Links(!)kontexts wird invertiert, damit man die rechteste Auflösungsmöglichkeit als erste bekommt.

⟨*Semantik/resolveDrs.pl*⟩+≡

```
reverse_drs(drs(R,F),drs(Rrev,Frev)):-
    reverse(R,Rrev),
    reverse(F,Frev).
```

Das Prädikat `resolveDrs/2` löst die Pronomen in den Eingabe-DRSen auf und liefert aufgelöste Ergebnis-DRSen:

⟨*Semantik/resolveDrs.pl*⟩+≡

```
% resolve_drs(+DRSen, -resolvedDRSen)
resolve_drs([drs(R,F)|Ds1],Ds2):-
    resolve_fml(F,[drs(R,[])|Ds1],Ds2).
```

In der ersten DRS werden die Formeln (und deren DRSen) mit `resolve/3` behandelt; im (vorläufigen) Zwischenergebnis `drs(R,[])` werden behandelte Formeln gesammelt (eingefügt).

Im Sonderfall, wo die erste Eingabe-DRS von der Form `seq(D,S)` ist, werden TODO

⟨*Semantik/resolveDrs.pl*⟩+≡

```
/*
resolve_drs([seq(D,S)],[drs(R,F)]):-
    resolve_drs([S,D],[drs(RS,FS),drs(RD,FD)]),
    append(RS,RD,R),
    append(FS,FD,F).
*/
resolve_drs([seq(D,S)|Ds1],[drs(R,F)|Ds3]):-
    resolve_drs([D|Ds1],Ds2),
    resolve_drs([S|Ds2],[drs(RS,FS),drs(RD,FD)|Ds3]),
    append(RS,RD,R),
    append(FS,FD,F).
```

Die Auflösung einer Anapher `anp(X:Ty,Gen,Num)` gegen ein Antezedens ist einfach: man nimmt eine DRS und sucht, ob ihre Formelliste ein Antezedenz `ant(X:Ty,Gen,Num)` enthält:

⟨*Semantik/resolveDrs.pl*⟩+≡

```
% resolve_anp(+Ref,+Gen,+Num,+DRSen)
resolve_anp(Ref,Gen,Num,Ds):-
    member(drs(_Refs,Fs),Ds),
    member(ant(Ref,Gen,Num),Fs).
```

Hierbei wird der Referent `X` in `anp(X:Ty,Gen,Num)` mit dem einer passenden Antezedensmarkierung unifiziert, was `+DRSen` *ändert*.

⟨*Semantik/resolveDrs.pl*⟩+≡

```
delete_ref(X,[R|Rs],NewRs) :- % entferne aufgelösten
    (X==R -> NewRs = Rs       % Referenten
    ; delete_ref(X,Rs,RsX),NewRs = [R|RsX]).
% delete_ref(_X,[],[]). % (HL) Doppelloesung verhindern!
```

Ist unter den Formeln eine Anapher `Ref`, so löst man sie in den erreichbaren DRSen `Ds1` auf und entfernt `Ref` aus den Referenten der Ergebnis-DRS `drs(R,F)`. Kann man sie nicht auflösen, so kommt die Anaphermarkierung in die Bedingungen der Ergebnis-DRS, und man bearbeitet die restlichen Formeln:

⟨*Semantik/resolveDrs.pl*⟩+≡

```
% resolve_fml(+Fmls,[?resultDRS|accessDRSen],
%                                        -resolvedDRSen)
resolve_fml([anp(Ref,Gen,Num)|Fmls],
            [drs(R,F)|Ds1], Ds2) :-
    !,
    ( resolve_anp(Ref,Gen,Num,Ds1),
      delete_ref(Ref,R,NewR),% Ref-Duplikate weglassen
      NewD = drs(NewR,F)     % anp(Ref,..) weglassen
    ; NewD = drs(R,[anp(Ref,Gen,Num)|F])
    ),
    resolve_fml(Fmls,[NewD|Ds1],Ds2).
```

Ist die nächste Formel aus DRSen aufgebaut, so löst man zuerst deren Anaphern auf. Dabei wird die Liste der zur Auflösung erreichbaren DRSen angepaßt:

⟨*Semantik/resolveDrs.pl*⟩+≡

```
resolve_fml([neg(D1)|Fmls],Ds1,Ds3):-
    !, resolve_drs([D1|Ds1],[D2,drs(R,F)|Ds2]),
    resolve_fml(Fmls,[drs(R,[neg(D2)|F])|Ds2],Ds3).

resolve_fml([(D1 => D2)|Fmls],Ds1,Ds4):-
    !, resolve_drs([D1|Ds1],Ds2),
    resolve_drs([D2|Ds2],[D4,D3,drs(R,F)|Ds3]),
    resolve_fml(Fmls,[drs(R,[(D3 => D4)|F])|Ds3],Ds4).

resolve_fml([or(D1,D2)|Fmls],Ds1,Ds4):-
    !, resolve_drs([D1|Ds1],[D3|Ds2]),
    resolve_drs([D2|Ds2],[D4,drs(R,F)|Ds3]),
    resolve_fml(Fmls,[drs(R,[or(D3,D4)|F])|Ds3],Ds4).
```

Andere Formeln, z.B. Antezedenzmarkierungen und Grundformeln, erfordern keine Pronomenauflösung und werden in die Ergebnis-DRS übertragen:

⟨*Semantik/resolveDrs.pl*⟩+≡

```
resolve_fml([Fml|Fmls],[drs(R,F)|Ds1],Ds2):-
    !, resolve_fml(Fmls,[drs(R,[Fml|F])|Ds1],Ds2).

resolve_fml([],[drs(R,F)|Ds],[drs(R,Frvs)|Ds]):-
    !, reverse(F,Frvs).
```

Da die Formeln invertiert wurden, wird das nach der Auflösung rückgängig gemacht.

Bem. Die Auflösung der Possessivpronomen muß man vor diese zwei Fälle noch einbauen!

⟨*Beispiel zur typkonformen Pronomenauflösung*⟩≡

```
?- parsedrsr. Galilei entdeckt einen Stern. er leuchtet.
Baum:
        d([def])
        + drs([X:m, Y:s],
              [ant(X:m, mask, sg), eq(X, galilei),
               stern(Y), ant(Y:s, mask, sg),
               entdecken(X, Y), leuchten(Y)])
        + drs([X:m, Y:s, Z:s],
              [ant(X:m, mask, sg), eq(X, galilei),
               stern(Y), ant(Y:s, mask, sg),
               entdecken(X, Y),
               anp(Z:s, mask, sg), leuchten(Z)])
          s([def], [praes, ind, vz])
          + ...
```

Hier wird `er = anp(Z:s,mask,sg)` nicht gegen `galilei = ant(X:m,mask,sg)` aufgelöst, da die `m` (Mensch) $\neq$ `s` (Stern).

⟨*Beispiel zur Pronomenauflösung*⟩≡

```
?- parsedrsr. ein Stern umkreist den Jupiter. er leuchtet
Aufruf: d([def], _, [er, leuchtet, ein, 'Stern',
Baum:                          umkreist, den, 'Jupiter'], []).
        d([def])
        + drs([X:s, Y:s],
              [stern(X), ant(X:s, mask, sg),
               ant(Y:s, mask, sg), eq(Y, jupiter),
               umkreisen(X, Y), leuchten(Y)])
        + drs([X:s, Y:s],
              [stern(X), ant(X:s, mask, sg),
               ant(Y:s, mask, sg), eq(Y, jupiter),
               umkreisen(X, Y), leuchten(X)])
        + drs([X:s, Y:s, Z:s],
              [stern(X), ant(X:s, mask, sg),
               ant(Y:s, mask, sg), eq(Y, jupiter),
               umkreisen(X, Y),
               anp(Z:s, mask, sg), leuchten(Z)])
```