Grammatikimplementierung
Vorlesung mit Seminar
WS 2001/2002

Hans Leiß, CIS

3. Juli 2002

Zusammenfassung

Dieses Skriptum enthält die in der Vorlesung besprochenen Programme zur Tokenisierung, lexikalischen Analyse und syntaktischen Analyse von Eingabesätzen, sowie eine Grammatik für Sätze des Deutschen. Die Grammatikregeln wurden von den Seminarteilnehmern Johannes Stiehler (Nominalphrasen), Angelica Hassel, Sara Tonelli, Yunbei Li (Verbalphrasen), Simone Leonardi, Nadine Lenk (Adjektivphrasen und Prädikativsätze), Paula Barros, Inger Zahrenhusen (Fragesätze) und Christoph Ringlstetter (Relativsätze) geschrieben, mit Anstößen und Ergänzungen von mir (Nominalphrasen, Präpositionalphrasen). Eine weitere Implementierung des Earley-Parsers stammt von Sebastian Hick (vgl. Seminarbeiträge).

Die meist nur unvollständig dokumentierten Teilgrammatiken stimmen nicht immer in den Kategorien und Merkmalen so überein, wie es nötig wäre, damit man sie gemeinsam benutzen kann. Eine Vereinheitlichung und Erweiterung um Stellungsvarianten wurde in Abschnitt 6.10 begonnen, konnte aber in diesem Semester nicht mehr abgeschlossen werden; ebenso fehlt die geplante Behandlung von Metaregeln.

Die Vereinheitlichung erfordert weitere Verbesserungen der Syntaxanalyseprogramme; die Programme zur Vorberechnung von Hilfstabellen zum Parsen sind für größere Grammatiken nicht mehr praktikabel. Zur Zeit ist das Parsen mit der Grammatik `s.grm` aus Abschnitt 6.10 daher so gut wie ausgeschlossen. Das soll im kommenden Semester verbessert werden.

Die Lexikoneinträge der Beispieltexte stammen ursprünglich aus dem CISLEX von Petra Maier-Meyer und sind hier (unvollständig) um die benötigten syntaktischen Informationen angereichert worden.

Inhaltsverzeichnis

1 Auswahl von Programmen und Optionen

Die im Folgenden entwickelten Satzanalyseprogramme sollen in verschiedenen Varianten und an verschiedenen Beispielgrammatiken (mit zugehörigen Lexika) getestet und verwendet werden können. Um das zu ermöglichen, verwenden wir die unten beschriebene Datei `auswahl.pl`. In ihr werden (meist 0-stellige) Prolog-Prädikate definiert, die die Auswahl der Grammatik- und Parserversionen und sonstige Bedienungsmöglichkeiten festlegen.

Die Bedienung wird erleichtert durch Verwenden des Editors Emacs, für den ein Menü „Grammatik“ in `Emacs/.emacs.grammatik.el` definiert ist, in dem die wichtigsen Bedienungsfunktionen von `auswahl.pl` angeboten werden.

Um das Menü in die Menüzeile des Editors einzufügen, sollte man zuerst den Pfadnamen zur Datei `auswahl.pl` in der Datei `Emacs/.emacs.grammatik.el` anpassen. Dann ergänzt man seine `~/.emacs`-Datei um die Zeile `(load-file «Pfad>/Emacs/emacs.grammatik.el")`. Nach dem Starten des Emacs erscheint dann das Menü „Grammatik“ in der Menüzeile.

1.1 Die Auswahldatei `auswahl.pl`

Um das zu ermöglichen, verwenden wir die Datei `auswahl.pl`. In ihr wird festgelegt, welche Programme und Daten es gibt und wie diese durch Prolog angesprochen werden. *Diese Datei legt die Bedeutung von Prädikaten fest, die zum Laden der gewünschten Daten und Programme dienen.*

Damit kann jeder Benutzer festlegen, welche Lexika, Grammatiken, Parsern u.a. statt der Originalversionen geladen werden. Dazu muß er von der hier mitgelieferten Musterdatei eine Kopie `auswahl.pl` machen und an seine Zwecke anpassen!

Die Musterdatei wird bei Neuversionen überschrieben, die Datei `auswahl.pl` bleibt dabei unverändert. Die Musterdatei besteht aus drei Teilen:

⟨*auswahl.muster.pl*⟩≡
  ⟨*Deklaration dateiübergreifender Prolog-Prädikate*⟩
  ⟨*Deklaration dynamischer Prolog-Prädikate*⟩
  ⟨*Programmvarianten*⟩
  ⟨*Hilfsfunktionen*⟩

1.1.1 Deklaration dateiübergreifender Prädikate

Im allgemeinen wird das Lexikon nicht aus einer, sondern aus mehreren Dateien bestehen, z.B. einem Lexikon für Nomina und einem anderen für Verben. Lexikoneinträge haben in allen Dateien des Vollformenlexikons die Form `word(<Wortform>,<Lexikalische Kategorie>)`.[1]

⟨*Deklaration dateiübergreifender Prolog-Prädikate*⟩≡

```
% Zu folgenden Prädikaten kann es Klauseln in verschiedenen Dateien geben:
:- multifile word/3, word/2, rule/2, '-->'/2, auxiliary_predicates/1, restrict/2.
```

[1] Die Einträge werden in Zukunft in die Form `word(<Wortform>,<Lexikalische Kategorie>,<Stammform>)` umgerechnet; aber bei den von Hand erstellten Einträgen muß man das dann nachtragen, oder `word/2` auf diese Sonderfälle und `word(Vollform,Kategorie,_)` zurückführen.

(Das Prädikat `word/2` braucht man noch, solange die von Hand ergänzten Lexikoneinträge nicht ebenfalls geändert worden sind.)

Ohne diese Deklaration würde Prolog beim Laden weiterer Lexikondateien die schon geladenen Fakten zu `word/3` überschreiben! Erst wenn man die Datei, in der die `multifile`-Deklaration steht, neu lädt, werden vorhandene Klauseln zu `word/3` überschrieben.

Auch die Grammatik besteht nicht aus einer einzigen Datei, sondern i.a. aus getrennten Teilgrammatiken für Nominalphrasen, Sätze, Adverbialphrasen usw. Grammatikregeln werden ursprünglich mit dem zweistelligen Prädikat `'-->'` geschrieben. Für manche Parser ist es zweckmäßig, die Regeln zuerst zu untersuchen und zu verändern und dann mit einem neuen Prädikat, `rule/2`, zu schreiben.

Weiter enthält jede Teilgrammatik Hilfsprädikate, die in den aus ihr erzeugten Parser exportiert werden müssen. Diese Hilfsprädikate werden in den Teilgrammatiken durch `auxiliary_predicates/1` angegeben; die verschiedenen Angaben von Hilfsprädikaten ergänzen einander. Daher muß auch `auxiliary_predicates/1` eine dateiübergreifende Definition haben.

Entsprechendes gilt für `restrict/2`, wodurch in den Teilgrammatiken (und bei der Beschreibung der Wortkategorien) festgelegt wird, welche Merkmale der Kategorien bei der Erzeugung von Hilfstabellen des Parsers ignoriert werden können.

1.1.2 Deklaration dynamischer Prädikate

Prolog-Prädikate, die zur Laufzeit der Programme geändert werden können, müß man zu *dynamischen* Prädikaten erklären, damit die Übersetzung in Maschinencode darauf Rücksicht nimmt.

Wir benutzen dynamische Prädikate für drei Zwecke.

(i) Programmoptionen sollen man zur Laufzeit ändern können: dazu verwenden wir einen *Schalter*. Durch `einschalten(<Option>).` wird dann ein Faktum `ein(<Option>).` zur Prolog-Datenbank hinzugenommen. Durch `ausschalten(<Option>)` wird dieses Faktum aus der Datenbank entfernt und durch `aus(<Option>)` ersetzt.

⟨*Deklaration dynamischer Prolog-Prädikate*⟩≡

```
% Folgende Prädikate können während der Laufzeit verändert werden:
:- dynamic ein/1, aus/1.
```

Wie das Umschalten erfolgt, ist in der vorher geladenen Datei `schalter.pl` definiert.

Damit man zwischen Varianten, die sich gegenseitig ausschließen, von Prolog aus eine wählen kann, erklären wir auch Prädikate als dynamisch, die die jeweilige Auswahl festhalten:

⟨*Deklaration dynamischer Prolog-Prädikate*⟩+≡

```
:- dynamic parser/1, grammar/1.
```

(ii) Teilprogramme sollen ihr Scheitern dokumentieren: Wenn ein Teilprogramm auf einen Fall trifft, den es nicht behandeln kann, soll es ein Faktum der Form `gescheitert(<Teilprogramm>)` in die Prolog-Datenbank schreiben. Dann braucht man nachfolgende Programme gar nicht erst zu starten.

⟨*Deklaration dynamischer Prolog-Prädikate*⟩+≡

```
:- dynamic gescheitert/1.
```

Beispielsweise kann die lexikalische Analyse vielleicht nicht alle Token im Lexikon finden; der Parser kann dann gleich abbrechen.

(iii) Programme können zur Laufzeit Hilfsdaten oder Programme berechnen: das jeweilige Programm legt die Namen der von ihm berechneten Hilfsprädikate selbst fest. Beispiele sind das für die Analyse eines Satzes benötige Sublexikon, oder die vor der Parsezeit berechneten Eigenschaften einer Grammatik.

1.1.3 Festlegung der Quelldateien

Für die Quelldateien der Grammatiken werden mit `grammar/2` Kurzbezeichnungen und die (relativen) Pfadnamen festgelegt:

⟨*Programmvarianten*⟩≡
```
% Dateinamen und ihre Abkuerzungen: grammar/2 und parser/3

grammar(1,  'Grammatiken/cfg.1.grm').
grammar(1.2,'Grammatiken/cfg.1.2.grm').
grammar(2,  'Grammatiken/cfg.2.grm').
grammar(del,'Grammatiken/deletions.grm').
grammar(np, 'Grammatiken/np.grm').
grammar(pp, 'Grammatiken/pp.grm').
grammar(pp1,'Grammatiken/pp1.grm').
grammar(app,'Grammatiken/app.grm').
grammar(vp, 'Grammatiken/vp.grm').
grammar(saetze,'Grammatiken/saetze.grm').
grammar(s,  'Grammatiken/s.grm').
grammar(qu, 'Grammatiken/qu.grm').
grammar(sh, 'Seminarbeitraege/Hick/synHick.grm').
```

Für die Quelldateien der Parser werden durch `parser/3` ebenfalls Kurzbezeichnungen und die Pfadnamen festgelegt; dazu wird ein Suffix angegeben, mit dem später der Name der für den jeweiligen Parser vorberechneten Grammatik aus dem Namen der Grammatik gebildet wird.

⟨*Programmvarianten*⟩+≡
```
parser(earley_doerre,      ed,  'Parser/earley.doerre.pl').   % Terme als Kategorien
parser(earley_doerre_0,    edg, 'Parser/earley.doerre.0.pl'). % Grundterme als Kateg.
parser(earley_pratt_leiss, epl, 'Parser/earley.pratt.leiss.pl').
parser(earley_hick,        eh,  'Seminarbeitraege/earleyHick.pl').
```

Vorauswahl von Grammatik, Lexikon, Parser und Optionen

Unter den möglichen Varianten der Daten und Programme muß jeweils eine ausgewählt werden. Das geschieht durch `auswaehlen/0` (was beim Starten von Prolog aus dem Grammatik-Menü des Emacs aufgerufen wird):

⟨*Programmvarianten*⟩+≡
```
% ------- Voreingestellte Auswahl und Optionen --------
% Kann man von Prolog aendern mit
% 'auswahl(parser(<Parser>)).' und 'einschalten(<Option>).' usw.

auswaehlen :-
        consult('Programmparameter/schalter.pl'),
        auswahl(basefirst(restrict)),  % zum Testen der First-
        auswahl(tc(warshall-4)),       % Vorberechnungen
        auswahl(predictor(uniform)),
        auswahl(parser(earley_pratt_leiss)),
        auswahl(grammar(app)),
        einschalten(predictUniform),   % einfacherer Predictor
        einschalten(pruefeLexWord),
        einschalten(omitEmptyTrees),
        einschalten(boundedChains),
```

```
        einschalten(showTrees),
        einschalten(asciiTrees).
```

Der Benutzer ist selbst dafür verantwortlich, daß seine Auswahl durch die folgenden Angaben, wo entsprechende Dateien zu finden sind, unterstützt wird.

Diese voreingestellte Auswahl kann von Prolog aus durch Eingeben von z.B. `auswahl(parser(earley_doerre)).` geändert werden, ohne daß man diese Auswahldatei neu laden muß. Allerdings muß man danach i.a. den Parser und die dafür vorberechnete Grammatik neu laden (mit `parser, grammatiken.`).

Laden der Programme und Grammatiken

Die folgenden 0-stelligen Prädikate dienen dazu, bestimmte Programmdateien oder Daten wie Lexika und Grammatiken in einfacher Weise (gegebenenfalls erneut) zu laden. Da für die Grundaufgaben wie Tokenisierung, lexikalische Analyse u.ä. vorerst keine Varianten geplant sind, treten Bedingungen nur beim Ladeaufruf für den Parser, die Lexika und die Grammatiken auf.

Je nach der getroffenen Auswahl werden der Parser, eine dafür vorberechnete Grammatik und die zur Grammatik passenden Lexika geladen.

⟨*Programmvarianten*⟩+≡

```
laden :-
        tokenizer, lexer, lexika, grammatiken, parser.

tokenizer :- consult('Tokenizer/tokenizer.pl').

satztrenner :- consult('Tokenizer/satztrenner.pl').

lexer :- consult('Tokenizer/lexer.pl').

umformatierer :- consult('Lexika/umformatieren.pl').

parser :-
        consult('Parser/main.pl'),
        consult('Parser/subsumption.pl'),
        consult('Parser/phrasenerkenner.pl'),
        ((parser(Parser), parser(Parser,_,Datei))
        ->  consult(Datei)
        ; write(user,'Der gewählte Parser ist nicht implementiert.\n'), fail).
```

Der Benutzer muß hier festlegen, welche Lexika zu welcher Grammatik passen und zu laden sind:

⟨*Programmvarianten*⟩+≡

```
% Welche Varianten werden (je nach der Auswahl) aus welchen Dateien geladen?

lexika :-
        (grammar(1) -> consult('Lexika/lexikon.1.wrd') ; true),
        (grammar(1.2) -> consult('Lexika/lexikon.1.wrd') ; true),
        (grammar(2) -> consult('Lexika/lexikon.2.wrd') ; true),
        (grammar(pp)
        -> lexika_wissenschaftsreligion,
            consult('Lexika/praepositionen.hilfslexikon.wrd')
        ; true),
        (grammar(pp1) -> grundlexika, lexika_wissenschaftsreligion ; true),
```

```
        (grammar(app) -> grundlexika, lexika_wissenschaftsreligion, lexika_ap ; true),
        (grammar(apn) -> grundlexika, lexika_wissenschaftsreligion, lexika_ap ; true),
        (grammar(np)  -> grundlexika, lexika_wissenschaftsreligion ; true),
        (grammar(vp)  -> grundlexika, lexika_vp ; true),
        (grammar(qu)  -> grundlexika, lexika_qu ; true),
        (grammar(saetze) -> lexika_wissenschaftsreligion,lexika_vp ; true),
        (grammar(s) ->
            grundlexika,lexika_wissenschaftsreligion,lexika_vp,lexika_qu ; true),
        (grammar(sh) -> consult('Seminarbeitraege/Hick/lexSynHick.swi') ; true).

grundlexika :-
        consult(['Lexika/en.wrd',
                 'Lexika/pronomen.wrd',
                 'Lexika/part.ddm.wrd',
                 'Lexika/verben.aux.wrd',
                 'Lexika/verben.modal.wrd',
                 'Lexika/praepositionen.wrd',
                 'Lexika/wissenschaftsreligion.lem.det.wrd',
                 'Lexika/wissenschaftsreligion.lem.n.wrd',
                 'Lexika/verben.ddm.wrd']).

lexika_wissenschaftsreligion :-
        consult(['Testbett/wissenschaftsreligion.lem.na.tmp.wrd',
                 'Testbett/wissenschaftsreligion.lem.a.tmp.wrd',
                 'Testbett/wissenschaftsreligion.lem.adv.tmp.wrd',
                 'Testbett/wissenschaftsreligion.lem.v.tmp.wrd',
                 'Testbett/wissenschaftsreligion.lem.part.tmp.wrd',
                 'Testbett/wissenschaftsreligion.lem.conj.tmp.wrd'
                 ]).

lexika_vp :-
        consult(['Seminarbeitraege/HasselLiTonelli/det.ddm.wrd.2',
                 'Lexika/n.ddm.wrd',
                 'Lexika/pron.ddm.wrd',
                 'Lexika/konjunktionen.ddm.wrd'
                 ]).

lexika_qu :- % was nicht per _wissenschaftsreligion oder _vp geladen wird:
        consult(['Seminarbeitraege/PaulaInger/en.fragen.wrd',
                 'Seminarbeitraege/PaulaInger/n.fragen.wrd',
                 'Seminarbeitraege/PaulaInger/praep.fragen.wrd'
                 ]).

lexika_ap :-
        consult(['Seminarbeitraege/LenkLeonardi/adverben.wrd',
                 'Seminarbeitraege/LenkLeonardi/praedikative.wrd',
                 'Seminarbeitraege/LenkLeonardi/partikel.wrd',
                 'Seminarbeitraege/LenkLeonardi/adjektive.wrd']).

% Vorberechnung und Umformung der ausgewaehlten Grammatik:

vorberechnungen :-
        write(user,'Vorberechung der ausgewaehlten Grammatik mit: vorberechnen/0.\n').
```

```
vorberechnen :-
        consult(['Parser/transclos.pl','Parser/vorberechnen.pl']),
        originalgrammatik(G),
        precomputations(G).

% Laden der vorberechneten und umgeformten Grammatik:

grammatiken :-  % Sonderfaelle, durch Namensaenderungen etc noch unter den allg. Fall bringen!
        (grammar(pp) ->
            consult('Grammatiken/np.cat.pl'),
            ((parser(earley_doerre) ; parser(earley_doerre_0))
            -> consult('Grammatiken/pp.grm.ed.pl') ; true),
            (parser(earley_pratt_leiss) -> consult('Grammatiken/pp.grm.epl.pl') ; true)
        ; true),
        (grammar(vp) ->
            (parser(earley_doerre)
            -> consult(['Seminarbeitraege/HasselLiTonelli/np.cat.ddm.pl',
                        'Seminarbeitraege/HasselLiTonelli/pp.grm.ddm.pl',
                        'Seminarbeitraege/HasselLiTonelli/vp.grm.ed.pl'])
            ; true),
            (parser(earley_pratt_leiss)
            -> consult(['Seminarbeitraege/HasselLiTonelli/np.cat.ddm.pl',
                        'Seminarbeitraege/HasselLiTonelli/pp.grm.ddm.pl',
                        'Grammatiken/vp.grm.epl.pl'])
            ; true)
        ; fail).

grammatiken :-  % allgemeiner Fall
        uebersetztegrammatik(G),
        (consult(G) -> true
        ; write(user,'Die Grammatik ist noch nicht vorberechnet.'), fail).
```

Die Sonderfälle, bei denen mehrere Dateien zu laden sind oder der Dateiname nicht aus dem der Originalgrammatik und dem Parsersuffix hervorgeht, sind noch auf dieses Schema zu bringen, indem man eine Hauptdatei mit dem passenden Namen anlegt, die die anderen Dateien lädt.

⟨*Hilfsfunktionen*⟩≡

```
% ------ Hilfsfunktionen zur Bestimmung der Dateinamen: ---------

originalgrammatik(G) :-
        (grammar(B)
        -> grammar(B,G)
        ; write(user,'Zuerst eine Grammatik im Menü auswählen!'), fail).

uebersetztegrammatik(G) :-
        originalgrammatik(Grammatik),
        parser(P),
        !,
        parser_suffix(GP),
        concat_atom([Grammatik,GP,'.pl'],G).
uebersetztegrammatik(G) :-
        write(user,'Zuerst Parser auswählen oder Grammatik vorberechnen!.'), fail.
```

```
grammar_suffix(Suffix) :-
        grammar(G),
        atom_concat('.',G,Suffix).
parser_suffix(Suffix) :-
        parser(X),parser(X,P,_),
        atom_concat('.',P,Suffix).
datum_suffix(Suffix) :-
        get_time(Time),
        convert_time(Time,Year,Month,Day,Hour,Min,_,_),
        concat_atom(['.',Day,'.',Month,'.',Year,':',Hour,':',Min],Suffix).

grammar_parser_datum_suffix(Suffix) :-
        grammar_suffix(G),
        parser_suffix(P),
        datum_suffix(D),
        concat_atom([G,P,D],Suffix).

% ------------------- Ende -----------------
```

Bem. Die Datei ist inzwischen verändert, was die Lexikonauswahl angehtd; siehe `auswahl.hl.pl`.

1.2 Ein- und ausschaltbare Optionen

Zum Testen der Programme ist es zweckmäßig, wenn man verschiedene Varianten durch einfaches Ein- und Ausschalten miteinander vergleichen kann. Dazu wird in den Programmen überprüft, ob ein bestimmter „Schalter" eingeschaltet ist; wenn dies der Fall ist, wird z.B. ein bestimmter Teil des Programms (nicht) ausgeführt.

Das Ein- und Ausschalten der Optionen wird durch Einträge in die Prolog-Datenbank verwirklicht:

⟨*Programmparameter/schalter.pl*⟩≡

```
einschalten(Schalter) :-
        aus(Schalter),!,
        retract(aus(Schalter)),
        einschalten(Schalter).
einschalten(Schalter) :-
        ein(Schalter),!.
einschalten(Schalter) :-
        assert(ein(Schalter)).

ausschalten(Schalter) :-
        ein(Schalter),!,
        retract(ein(Schalter)),
        ausschalten(Schalter).
ausschalten(Schalter) :-
        aus(Schalter),!.
ausschalten(Schalter) :-
        assert(aus(Schalter)).
```

Was gerade ein- und was ausgeschaltet ist, kann man durch Aufrufen von `schalter/0` abfragen:

⟨*Programmparameter/schalter.pl*⟩+≡

```
schalter :-
        nl,write('Optionen (ein): '),
```

```
        (setof(Opt,ein(Opt),OptionenEin) -> write(OptionenEin) ; write('keine ')),
        nl,write('Optionen (aus): '),
        (setof(Opt,aus(Opt),OptionenAus) -> write(OptionenAus) ; write('keine ')).
```

Folgende Optionen bzw. Schalter sind bisher vorgesehen:

(i) Anzeigeschalter:

(a) `showProfile`: Für bestimmte Aufrufe soll mit `profile/3` von SWI-Prolog der Aufwand berechnet werden.

(b) `showLexAnalyse`: Das Ergebnis der lexikalischen Analyse soll am Bildschirm angezeigt werden.

(c) `showTrees`: Nach der Syntaxanalyse sollen die gefundenen Strukturen als Bäume angezeigt werden, mit `asciiTrees` oder mit `texTrees`.

(d) `verbose`: Gewisse Programme sollen Informationen zur Berechnung ihrer Ergebnisse anzeigen (zum Verständnis und zur Fehlersuche)

(ii) Veränderung der Funktionalität:

(a) `satzzeichen`: Die lexikalische Analyse soll Satzzeichen entfernen, bis auf das am Ende der Eingabe.

(b) `filterLexAnalyse`: Nach der lexikalischen Analyse sollen (durch Befragen des Benutzers) unnötige Einträge aus dem Sublexion `lexword/3` entfernt werden, bevor die Syntaxanalyse startet.

(c) `omitEmptyTrees`: Der Parser soll keine Teilbäume konstruieren, die ein leeres Stück der Eingabe abdecken, d.h. keine Löschungen. Das reduziert die Ambiguität, garantiert aber noch nicht die Terminierung: es wird nur verhindert, daß die *Breite* von Analysebäumen die Länge der Wortfolge überschreitet; die *Tiefe* kann immer noch unbeschränkt wachsen.

(d) `boundedChains`: Der Parser soll die Länge von Kettenableitungen dadurch beschränken, daß aufeinanderfolgende Anwendungen von Kettenregeln nur paarweise verschiedene Kategorien enthalten. Damit wird die Tiefe von Analysebäumen durch die Länge der Eingabe abschätzbar.

(e) `pruefeLexWord`: Nach der lexikalischen Analyse soll geprüft werden, ob alle lexikalischen Kategorien der Eingabewörter auch der Grammatik bekannt sind.

Die Anzeigen am Bildschirm erlauben es, die Ergebnisse einer Teilanalyse zum Verständnis und zur Fehlersuche von umfassenden Programmen zu nutzen.

Damit man nicht ständig an der Auswahldatei ändern muß, ist es zweckmäßig, daß man die Auswahl eines anderen Beispiels oder eines anderen Parsers interaktiv in Prolog machen kann. Dazu dient

⟨*Programmparameter/schalter.pl*⟩+≡

```
 auswahl(Term) :-
        (functor(Term,F,1)
        -> length(L,1),
            Muster =.. [F|L],
            retractall(Muster),
            assert(Term)
        ; write('ERROR: Das Argument ist nicht von der Form: parser(Atom), grammar(Atom)', basefirst(Atom), tc(Atom))).
```

Mit `auswahl(parser(earley_doerre))` wird z.B. ein Parser ausgewählt (und die Auswahl eines anderen Parsers aufgehoben); die Fallunterscheidungen unter `grammatiken/0` und `parser/0` sorgen dann dafür, daß die richtigen Dateien geladen werden – falls sie existieren!

Falls für den gewählten Parser die vorberechnete Grammatik noch nicht existiert, muß man zuerst `vorberechnungen.` laden und dann mit `precomputations(<Grammatik>.grm)` die vorberechnete Grammatik erzeugen.

Um sich die geltende Auswahl anzeigen zu lassen, rufe man `auswahl.` auf:

⟨*Programmparameter/schalter.pl*⟩+≡

```
auswahl :-
        nl,write('First-Relation: '),
        (setof([B,H],(basefirst(B),tc(H)),[BH]) -> write(BH) ; write('keine Wahl getroffen')),
        nl,write('Predictor: '),
        (setof(P,predictor(P),Predictor) -> write(Predictor) ; write('keine Wahl getroffen')),
        nl,write('Parser: '),
        (setof(P,parser(P),Parser) -> write(Parser) ; write('keine Wahl getroffen')),
        nl,write('Grammatik: '),
        (setof(G,grammar(G),Grammatik) -> write(Grammatik) ; write('keine Wahl getroffen')).
```

Wenn neue Auswahlmöglichkeiten eingebaut werden, muß dieses Prädikat angepaßt werden!

2 Einleitung: Prinzip der Satzanalyse

Bevor man Sätze syntaktisch analysieren kann, braucht man

(i) ein *Lexikon*, das die Wörter, d.h. die atomaren Ausdrücke der Sprache, bestimmt und zu jedem Wort bestimmte Informationen festlegt,

(ii) eine *Grammatik*, die festlegt, wie die zusammengesetzten Ausdrücke aus Teilausdrücken aufgebaut werden dürfen,

(iii) ein *Analyseverfahren*, nach dem man mit Hilfe der Grammatik die verschiedenen Möglichkeiten, einen vorgegebenen Ausdruck zu konstruieren, ermittelt.

Bei der Analyse kann man verschieden Phasen unterscheiden:

(i) die *Tokenisierung*: ein zu analysierender Ausdruck wird aus einer Datei oder von der Tastatur eingelesen und wird dann in eine *Liste von Prolog-Atomen* zerlegt, der Eingabefolge. Jedes Element oder *Token* der Eingabefolge sollte ein Wort der Sprache sein.

(ii) die *lexikalischen Analyse*: für jedes Element der Eingabefolge werden Informationen im Lexikon nachgeschlagen; gegebenenfalls wird festgestellt, daß das Token kein zulässiges Wort ist.

Die Einträge werden in ein Teillexikon extrahiert (*Lemmatisierung*) und, wenn möglich, auf jeweils eine Information pro Token eingeschränkt (*Tagging*).

(iii) die *syntaktische Analyse*: aus den Informationen zu den Token wird die syntaktische Struktur des Eingabesatzes bestimmt. Dazu gibt es verschiedene *Analyseverfahren* oder *Parser*.

Die Ausgabe der syntaktischen Analyse ist eine Baumstruktur (in der Form eines Prolog-Terms), die auch graphisch angezeigt werden kann.

3 Tokenisierung

Die Eingabe eines Ausdrucks kann auf zwei Weisen erfolgen:

(i) durch Lesen von der Konsole, oder

(ii) durch Lesen aus einer Textdatei.

In beiden Fällen muß zuerst das Ende der Eingabe erkannt werden, bevor man die Eingabe in eine Folge von Token zerlegen kann. Die unten folgenden Programme stehen im Verzeichnis `Tokenizer`.

⟨*Tokenizer/tokenizer.pl*⟩≡
  ⟨*Tokenisierung*⟩
  ⟨*Satzende-Erkenner*⟩
  ⟨*Zerlegung in Token*⟩
  ⟨*skipComment*⟩

3.1 Ende der Eingabe

Da für die Satzanalyse die einzelnen Sätze als Eingabe dienen, geht es hier darum, beim Lesen einer Zeichenfolge (von der Konsole oder aus einer Datei) das erste Anfangsstück zu erkennen, das einen Satz bilden könnte.

Ein *Satzende-Erkenner* soll, –ohne Grammatik– (mögliche) *Satzenden* erkennen, d.h. gewisse Vorkommen von Sonderzeichen wie `.`, `?` und `!`. Anschließend soll die *Zerlegung in Token* der Satz in einzelen Token, i.e. vermutete Wörter, zerlegen.

Das Lesen aus einer Datei oder von der Konsole sind für Prolog Spezialfälle des Lesens von einem *Strom*. Zum Umgang mit Strömen gibt es spezielle Prolog-Prädikate, z.B.

`open(Datei+,Bearbeitungsmodus+,Stromname-)`

`current_stream(Datei?,Bearbeitungsmodus?,Stromname?)`.

3.1.1 Einlesen von einer Datei

Falls die Datei schon zum Lesen geöffnet ist, wird der nächste Satz –die Zeichenreihe vom zuletzt gelesenen Zeichen bis zur nächsten Satzgrenze– gelesen und in eine Liste von Atomen umgewandelt:

⟨*Tokenisierung*⟩≡
```
read_sentence(File,Atomlist) :-
        current_stream(File,read,Stream),
        !,read_sentence(Stream,Sentence,[]),
        stringToAtoms(Sentence,Atomlist).
```

Andernfalls wird zuerst die Datei zum Lesen geöffnet:

⟨*Tokenisierung*⟩+≡
```
read_sentence(File,Atomlist) :-
        open(File,read,Stream),
        read_sentence(Stream,Sentence,[]),
        stringToAtoms(Sentence,Atomlist).
```

Das aufrufende Programm muß den Strom schließen.

3.1.2 Einlesen von der Konsole

Das Lesen von der Konsole (oder dem Bildschirm), dem Prolog-Strom `user`, soll nach einer Eingabeaufforderung an den Benutzer erfolgen.

⟨*Satzende-Erkenner*⟩≡
```
satzende :-
        write('Beende den Eingabesatz mit <Satzzeichen><Return>'),nl,
        read_sentence(user,Sentence,[]),
        name(Satz,Sentence),
        nl,writeq(Satz).
```

Nach Eingabe von `satzende.` wird die eingetippte Zeichenreihe am ersten Satzende abgeschnitten und als Prolog-Atom am Bildschirm ausgegeben.

⟨*Zerlegung in Token*⟩≡
```
tokenize :-
        write('Beende den Eingabesatz mit <Satzzeichen><Return>'),nl,
        read_sentence(user,Sentence,[]),
        stringToAtoms(Sentence,Atomliste),
        nl,writeq(Atomliste).
```
Nach Eingabe von `tokenize.` wird der erste eingetippte Satz in Token zerlegt und die enstandene Liste von Atomen am Bildschirm gezeigt.

3.1.3 Weiterlesen bis zum Satzende

Durch `read_sentence(Stream+,Sentence-,[])` wird das Lesen vom Strom `Stream` begonnen. Als `Sentence` wird der Anfang des Zeichenstroms bis zum nächsten Satzende ausgegeben.

Im dritten Argument, einer Prolog-Liste `[]` bzw. `Seen`, merkt man sich die schon gelesenen Zeichen in umgekehrter Reihenfolge (Akkumulator). Durch

⟨*Satzende-Erkenner*⟩+≡
```
read_sentence(Stream,Sentence,Seen) :-
        get0(Stream,Char),
        !,read_sentence(Stream,Sentence,Char,Seen),
        (dateiende(Char)
         -> write('End of file reached.')
        ; true).
```

wird das erste Zeichen, `Char`, vom Eingabestrom gelesen. Falls damit das Ende der Datei erreicht war, wird das am Bildschirm gemeldet. Sonst wird mit `read_sentence/4` je nach `Char` und den schon vorher gelesenen Zeichen `Seen` weitergelesen und schließlich ein Satz in `Sentence` ausgegeben.

Das Erkennen des Satzendes

Wie soll nun an `Char` und `Seen` das Ende des zu lesenden Satzes erkannt werden?

(i) Vermutlich liegt ein Satzende vor, wenn `Char` das Ende der Datei markiert; dann kehrt man die Liste `Seen` der gelesenen Zeichen um und gibt den Satz in `Sentence` aus:

⟨*Satzende-Erkenner*⟩+≡
```
read_sentence(_Stream,Sentence,Char,Seen) :-
        dateiende(Char),
        !,reverse(Seen,Sentence).
```
⟨*Ignoriere Kommentare*⟩

Da nicht jeder Punkt vor einem Leerzeichen ein Satzende markiert, fangen wir zuerst zwei Ausnahmen ab:

(ii) Punkte am Ende von Ordinalzahlen (`der 14. Mai`) sind keine Satzenden:

⟨*Satzende-Erkenner*⟩+≡
```
read_sentence(Stream,Sentence,Char,[Punkt,Ziffer|Seen]) :-
        ordinalzahlende([Ziffer,Punkt,Char]),
        !,read_sentence(Stream,Sentence,[Char,Punkt,Ziffer|Seen]).
```

Satzenden wie in `Er kam am 3.12.` werden also *nicht* erkannt. Das Ordinalzahlende wird unter den Hilfsbegriffen unten definiert, die Ordinalzahlen im Anhang 9.

(iii) Punkte am Ende von Abkürzungen (`Dr. Schein z.B. wohnt in der Sinnstr. 6`) werden einfach überlesen:

⟨*Satzende-Erkenner*⟩+≡
```
read_sentence(Stream,Sentence,Char,[R|Seen]) :-
        leerZeichen(Char),punktZeichen(R),
        (abkuerzung(Abk); suffixAbkuerzung(Abk)),
        reverse(Abk,RevAbk),
        append(RevAbk,_,[R|Seen]),
        !,read_sentence(Stream,Sentence,[Char,R|Seen]).
```

In den verbleibenden Fällen hängt es von den beiden letzten Zeichen ab, ob ein Satzende vorliegt. Wenn ein Satzende erreicht ist (vgl. `satzende/1` unten), wird der gefundene Satz in `Sentence` zurückgegeben.

⟨*Satzende-Erkenner*⟩+≡
```
read_sentence(_Stream,Sentence,Char,[C|Seen]) :-
        satzende([C,Char]),
        !,reverse([C|Seen],Sentence).
```

In allen anderen Situationen wird `Char` nicht beachtet und weiter vom Strom gelesen, bis ein Satzende auftritt:

⟨*Satzende-Erkenner*⟩+≡
```
read_sentence(Stream,Sentence,Char,Seen) :-
        read_sentence(Stream,Sentence,[Char|Seen]).
```

Aufgabe 3.1 Erweitere dieses Prädikat so, daß beim Lesen von Dateien alle Kommentare ignoriert werden. Als Kommentar soll das Zeichen `%` und alle folgenden auf derselben Zeile der Datei gelten.

Aufgabe 3.2 Schreibe mit Hilfe von `read_sentence/2` ein Programm, das eine Textdatei liest und ihre Sätze der Reihe nach jeweils in eine eigene Zeile einer Ausgabedatei schreibt.

Hilfsbegriffe der Satzende-Erkennung

Einige Hilfsdefinitionen bleiben nachzutragen.

⟨*Satzende-Erkenner*⟩+≡
⟨*satzende*⟩
⟨*ordinalzahlende*⟩
⟨*Abkürzungen*⟩
⟨*Zeichen und ihre Eigenschaften*⟩

Da Zeichenreihen in Prolog mit Listen von Character-Codenummern gleichgesetzt werden, wird hier bei Sonderzeichen wie `newline` diese Codenummer benutzt:

⟨*satzende*⟩≡
```
satzende(". ").          %  .Char ist kein Satzende.
satzende("? ").
satzende("! ").
satzende(": ").
satzende([46,10]).       %  . newline
satzende([63,10]).       %  ? newline
satzende([33,10]).       %  ! newline
satzende([58,10]).       %  : newline
```

⟨*ordinalzahlende*⟩≡
```
ordinalzahlende([Ziffer,Punkt,Char]) :-
        ziffer(Ziffer),
        punktZeichen(Punkt),
        leerZeichen(Char).
```

Dazu kommen noch die Abkürzungen, also bestimmte Zeichenreihen, die in einem Punkt enden. Für die Tokenisierung bestimmt man die Abkürzungen nach einer fest vorgegebenen Abkürzungs*tabelle* `abkuerzung/1`, die aus einem für die lexikalische Analyse vorgesehenen Abkürzungs*lexikon*, `abkuerzung/2` berechnet wird:

⟨*Abkürzungen*⟩≡
```
abkuerzung(String) :-
        abkuerzung(String,Kategorie).
```

Zusätzlich gibt es noch Abkürzungen, die nicht im Abkürzungslexikon stehen, sondern an einem Suffix erkannt werden:

⟨*Abkürzungen*⟩+≡
```
suffixAbkuerzung("str.").
```

Der Tokenizer erzeugt damit Token wie `Bahnhofstr.`, aber bisher werden diesen Token in der nachfolgende lexikalische Analyse keine syntaktischen Kategorien zugeordnet. (Man sollte sie zu Nomina machen.)

Im Abkürzungs*lexikon* werden lexikalische und grammatischen Kategorien verwendet, die zu der Satzgrammatik aus Abschnitt 6 passen. (Es wird hier beschrieben, damit es aus derselben Datei extrahiert werden kann.)

Alle Abkürzungen, die wie die in `T.S.Eliot` nur aus einem Großbuchstaben mit Punkt bestehen, fassen wir als Abkürzungen von Vornamen auf:

⟨*Abkürzungen*⟩+≡
```
abkuerzung([C,46], vn([unsp],[sg,_Kas])) :-
        grossbuchstabe(C). %, kasus(Kas).
```

Daneben haben wir Abkürzungen für Titel, Anreden, Monatsnamen und ein paar weitere, deren lexikalische Beschreibung *noch zu überlegen und nachzutragen* ist:

⟨*Abkürzungen*⟩+≡
```
abkuerzung("Mag.", titel([unsp],[sg])).
abkuerzung("Dipl.",titel([unsp],[sg])).
abkuerzung("Ing.", titel([unsp],[sg])).
```

```
abkuerzung("Dr.",  titel([unsp],[sg])).
abkuerzung("Prof.",titel([unsp],[sg])).

abkuerzung("Jan.", n([mask],[sg,_Kas])). % :- kasus(Kas).
abkuerzung("Feb.", n([mask],[sg,_Kas])). % :- kasus(Kas).
abkuerzung("Apr.", n([mask],[sg,_Kas])). % :- kasus(Kas).
abkuerzung("Aug.", n([mask],[sg,_Kas])). % :- kasus(Kas).
abkuerzung("Sept.",n([mask],[sg,_Kas])). % :- kasus(Kas).
abkuerzung("Okt.", n([mask],[sg,_Kas])). % :- kasus(Kas).
abkuerzung("Nov.", n([mask],[sg,_Kas])). % :- kasus(Kas).
abkuerzung("Dez.", n([mask],[sg,_Kas])). % :- kasus(Kas).

abkuerzung("Fa.", anrede([neut],[sg])).
abkuerzung("Hr.", anrede([mask],[sg])).
abkuerzung("Fr.", anrede([fem],[sg])).
abkuerzung("Frl.",anrede([neut],[sg])).
abkuerzung("Fam.",anrede([fem],[sg])).

abkuerzung("Nr.").  % wie sollte man das klassifizieren?

abkuerzung("rer."). % wie sollte man das klassifizieren?
abkuerzung("nat.").
abkuerzung("math.").

abkuerzung("bzw.",conj([],[])).  % welche Parameter?
abkuerzung("etc.").
abkuerzung("usw.").

abkuerzung("vgl."). % wie sollte man das klassifizieren?
abkuerzung("s.u.").

abkuerzung("z.B.",  pp([def],[zu(dat)])).  % oder als Adverbiale?
abkuerzung("a.a.O.",pp([def],[an(dat)])).
abkuerzung("m.M.",  pp([poss(1,sg)],[nach])).
abkuerzung("z.T.",  pp([def(3,sg)],[zu])).
```

Bem. Der Kasus bei den Monatsabkürzungen ist auskommentiert, damit man das Prädikat `kasus/1` aus der Grammatik nicht schon bei der Tokenisierung braucht.

Sowohl im Satzende-Erkenner als auch bei der Tokenisierung braucht man Testprädikate, mit denen man spezielle Zeichen an ihrer Codenummer erkennt. Entsprechende Testprädikate baucht man auch für Prolog-Atome, die nur aus diesen Zeichen bestehen:

⟨*Zeichen und ihre Eigenschaften*⟩≡

```
dateiende(-1).                 %  eof
zeilenende(10).                %  newline
worttrennung([45,10]).         %  - newline

grossbuchstabe(C) :-
        member(C,"ABCDEFGHIJKLMNOPQRSTUVWXYZ").
kleinbuchstabe(C) :-
        member(C,"abcdefghijklmnopqrstuvwxyz").
ziffer(Z) :-
        member(Z,"0123456789").
```

```
apostroph_s('''s',"'s").

leer(' ').
leerZeichen(C) :- name(' ',[C]).
komma(',').
kommaZeichen(C) :- name(',',[C]).
punkt('.').
punktZeichen(C) :- name('.',[C]).
trennZeichen(C) :- name('-',[C]).
```

Außer in Zahlausdrücken und Abkürzungen sollen bestimmte Zeichen zu eigenständigen Token werden; man muß dazu ihre Codenummer in das entsprechende Prolog-Atom umwandeln:

⟨*Zeichen und ihre Eigenschaften*⟩+≡

```
sonderzeichen('.',46).  % name('.',[46]).  usw.
sonderzeichen(',',44).
sonderzeichen(';',59).
 sonderzeichen(':',58).
sonderzeichen('!',33).
sonderzeichen('?',63).
sonderzeichen('"',34).  % bei Zitaten
sonderzeichen('(',40).
sonderzeichen(')',41).
```

Aufgabe 3.3 Man lade die Datei `Tokenizer/tokenizer.pl` in SWI-Prolog und teste, ob der Aufruf von `eingabe.` die eingetippte Zeichenreihe am ersten Satzende abschneidet.

Das in `read_sentence/2` verwendete, bisher nicht definierte Prädikat `stringToAtoms` definiere man dazu durch

⟨*Zerlegung in Token (vorläufig)*⟩≡

```
stringToAtoms(Sentence,Atomliste) :-
        name(Atomliste,Sentence).
```

Es wird also bloß die Zeichenreihe (mit Leerzeichen!) als Prolog-Atom ausgegeben. Was ist am Programm noch zu ändern, d.h. welche Satzenden werden nicht oder welche Fälle werden irrtümlich als Satzenden erkannt?

3.2 Zerlegung eines Satzes in Wörter (Token)

Die durch `read_sentence(Stream+,Sentence-)` zurückgegebene Zeichenreihe `Sentence` ist noch nicht in einzelne Wörter zerlegt. Das erfolgt durch den *Tokenizer*, der eine Zeichenfolge in eine Liste von Atomen umwandelt:[2]

⟨*Zerlegung in Token*⟩+≡

```
stringToAtoms(String,Atomlist) :-
        tokenize(String,nt,[],Atomlist).
```

⟨*tokenize*⟩

[2]Wenn man bei `tokenize/4` als Ergebnis nur die invertierte Zeichenfolge ausgibt und diese dann in `stringToAtoms` als Atomfolge in der richtigen Reihenfolge umwandelt, würde man sich eine Invertierung sparen. Dann muß man bei der normalen Umwandlung von Zeichenreihen zuerst die Zeichenreihe invertieren.

Die eigentliche Arbeit erledigt das Prädikat

```
tokenize(+String,+Flag,+GeseheneTokenInvers,-Ergebnis).
```

Im Argument `+GeseheneTokenInvers` merkt man sich schon gefundene Token in einer Liste (von Prolog-Atomen) in invertierter Reihenfolge.

Wenn die schon gelesenen Zeichen *innerhalb* eines Wortes enden, wird das gelesene Präfix des Worts noch nicht in ein Token (Prolog-Atom) umgewandelt. Das *erste* Listenelement von `GeseheneTokenInvers` ist daher i.a. kein Token, sondern Zeichenliste, das invertierte Präfix (vgl. Fall 1,c) und Fall 2,g)).

Man unterscheidet zwei Fälle (dazu dient das Argument `Flag`):

Fall 1 Mit dem nächsten Zeichen beginnt ein *n*eues *T*oken (das zweite Argument `Flag` ist `nt`).

a) Am Anfang eines Tokens werden Leerzeichen und Zeilenumbrüche ignoriert:

⟨*tokenize*⟩≡
```
tokenize([C|String],nt,Rs,Ts) :-
        (leerZeichen(C); zeilenende(C)),
        !,tokenize(String,nt,Rs,Ts).
```

b) Falls das nächste Zeichen ein eigenständiges Token ist, z.B. ein Satzzeichen oder eine Klammer, wird es als Atom in die Liste der gefundenen Token angefügt, und wir sind wieder in Fall 1.

⟨*tokenize*⟩+≡
```
tokenize([C|String],nt,Rs,Ts) :-
        sonderzeichen(Atom,C),
        !,tokenize(String,nt,[Atom|Rs],Ts).
```

c) Sonst wird das nächste Zeichen als Anfang eines gefundenen Token auf die Liste gesetzt, und wir gehen in Fall 2, bis das Ende dieses Tokens erkannt wird:

⟨*tokenize*⟩+≡
```
tokenize([C|String],nt,Rs,Ts) :-
        !,tokenize(String,at,[[C]|Rs],Ts).
```

d) Falls die Eingabezeichenreihe leer ist, wird die Liste der schon gefundenen Token in der richtigen Reihenfolge als Antwort zurückgegeben:

⟨*tokenize*⟩+≡
```
tokenize([],nt,Rs,Ts) :-
        reverse(Rs,Ts).
```

Fall 2 Das letzte gelesene Zeichen lag in einem *a*ngefangenen *T*oken (zweites Argument ist `at`).

Wir behandeln zuerst einige Sonderfälle; falls diese nicht auftreten, ist das Weiterlesen einfach.

a) Worttrennungen, d.h. Zeilenumbrüche innerhalb eines Tokens, werden beseitigt, und zwar sowohl wenn das getrennte Wort direkt am Anfang der folgenden Zeile weitergeht, als auch dann, wenn diese mit Leerzeichen beginnt (z.B. bei eingerückten Abschnitten):

⟨*tokenize*⟩+≡
```
tokenize([N,L|String],at,[[T|Cs]|Rs],Ts) :-
        worttrennung([T,N]),leerZeichen(L),
        !,tokenize([N|String],at,[[T|Cs]|Rs],Ts).
tokenize([N|String],at,[[T|Cs]|Rs],Ts) :-
        worttrennung([T,N]),
        !,tokenize(String,at,[Cs|Rs],Ts).
```

b) Ist das nächste Zeichen ein Zeilenumbruch, beenden wir das angefangene Token und ersetzen es auf der Liste der gefundenen Token (wo es invertiert als Zeichenreihe gespeichert war) durch das entsprechende Prolog-Atom:

⟨*tokenize*⟩+≡
```
tokenize([C|String],at,[R|Rs],Ts) :-
        zeilenende(C),
        !,tokenToAtom(R,Atom),
        tokenize(String,nt,[Atom|Rs],Ts).
```

c) Ist das nächste Zeichen ein Leerzeichen, beenden wir ebenfalls das angefangene Token, außer wenn das angefangene Wort in `'s` endet; in diesem Sonderfall bilden wir zwei neue Token:

⟨*tokenize*⟩+≡
```
tokenize([C|String],at,[[S,A|R]|Rs],Ts) :-
        leerZeichen(C),
        apostroph_s(ES,[A,S]),
        !,tokenToAtom(R,Atom),
        tokenize(String,nt,[ES,Atom|Rs],Ts).

tokenize([C|String],at,[R|Rs],Ts) :-
        leerZeichen(C),
        !,tokenToAtom(R,Atom),
        tokenize(String,nt,[Atom|Rs],Ts).
```

d) Ein Punkt nach einer Ziffer und vor einem Leerzeichen wird mit der Ziffer zu einem Zahlausdruck gerechnet:

⟨*tokenize*⟩+≡
```
tokenize([S,L|String],at,[[C|Cs]|Rs],Ts) :-          % Cs+C._ = Ordinal_
        ziffer(C),punktZeichen(S),leerZeichen(L),
        !,tokenToAtom([S,C|Cs],Ordinalzahl),
        tokenize(String,nt,[Ordinalzahl|Rs],Ts).
```

Ein Komma nach einer Ziffer und vor einem Leerzeichen wird als eigenes Token gewertet:

⟨*tokenize*⟩+≡
```
tokenize([S,L|String],at,[[C|Cs]|Rs],Ts) :-          % Cs+C,_ = Zahl,_
        ziffer(C),kommaZeichen(S),leerZeichen(L),
        !,tokenToAtom([C|Cs],Zahl),
        tokenize([S|String],nt,[Zahl|Rs],Ts).
```

Aber Punkt und Komma zwischen Ziffern gelten als Teile von Zahlausdrücken:

⟨*tokenize*⟩+≡
```
tokenize([S,L|String],at,[[C|Cs]|Rs],Ts) :-          % Cs+C.Z = Zahlteil
        ziffer(C),(punktZeichen(S);kommaZeichen(S)),ziffer(L),
        !,tokenize(String,at,[[L,S,C|Cs]|Rs],Ts).
```

e) Ebenso müssen wir Punkte in Abkürzungen vom Punkt am Satzende unterscheiden:

⟨*tokenize*⟩+≡
```
tokenize([S|String],at,[R|Rs],Ts) :-
        punktZeichen(S),
        suffixAbkuerzung(Abkuerzung),                % XYstr. 45
        reverse(Abkuerzung,[S|RevAbk]),
        append(RevAbk,_,R),
        !,tokenToAtom([S|R],Atom),
```

```
        tokenize(String,nt,[Atom|Rs],Ts).

    tokenize([S|String],at,[R|Rs],Ts) :-                     %  wie z.B.
        punktZeichen(S),
        abkuerzung(Abkuerzung),
        append(Praefix,[S|Suffix],Abkuerzung),
        reverse(Praefix,R),
        append(Suffix,RestString,String),
        !,name(Atom,Abkuerzung),
        tokenize(RestString,nt,[Atom|Rs],Ts).

    tokenize([S|String],at,[R|Rs],Ts) :-
        punktZeichen(S),abkuerzung(Abkuerzung),
        reverse(Abkuerzung,[S|R]),
        !,name(Atom,Abkuerzung),
        tokenize(String,nt,[Atom|Rs],Ts).
```

f) Ist das nächste Zeichen ein Sonderzeichen, das ein eigenes Token bildet – z.B. ein Satzzeichen –, so werden das angefangene Token und das Sonderzeichen als Atome auf die Liste der gefundensen Token gelegt:

⟨*tokenize*⟩+≡
```
    tokenize([S|String],at,[R|Rs],Ts) :-
        sonderzeichen(SAtom,S),
        !,tokenToAtom(R,Atom),
        tokenize(String,nt,[SAtom,Atom|Rs],Ts).
```

g) In allen anderen Fällen erweitern wir das angefangene Token um das nächste Zeichen und lesen in Fall 2 weiter:

⟨*tokenize*⟩+≡
```
    tokenize([C|String],at,[R|Rs],Ts) :-
        !,tokenize(String,at,[[C|R]|Rs],Ts).
```

h) Ist schließlich das Ende der Eingabezeichenreihe erreicht, muß das letzte angefangene Token in ein Atom umgewandelt und die Liste der gefundenen Token, in der richtigen Reihenfolge, als Antwort `Ts` zurückgegeben werden:

⟨*tokenize*⟩+≡
```
    tokenize([],at,[R|Rs],Ts) :-
        tokenToAtom(R,Atom),
        reverse([Atom|Rs],Ts).
```

Da die lexikalische Analyse mit Prolog-Atomen statt Zeichenreichen arbeiten soll, muß jedes gefundene Token, das zunächst nur als invertierte Zeichenreihen `R` auf der Hilfliste `[R|Rs]` steht, in ein Atom umgewandelt werden:

⟨*tokenize*⟩+≡
```
  tokenToAtom(RevToken,Atom) :-
      reverse(RevToken,Token),
      name(Atom,Token).
```

Damit ist die Zerlegung des Eingabesatzes in einzelne Wörter abgeschlossen.

Aufgabe 3.4 Ändere den Satztrenner wie folgt:

(i) Gedankenstriche - wie diese hier - sollen nicht als Worttrenner behandelt werden, sondern im Text stehen bleiben.

(ii) In der Ausgabedatei soll zwischen zwei Sätzen der Eingabedatei *eine* Leerzeile, zwischen zwei Absätzen der Eingabedatei dagegen *zwei* Leerzeilen erscheinen.

4 Lexikalische Analyse

Die lexikalische Analyse bildet aus der Eingabeliste von Atomen ein Lexikon, `lexword/3`, dem die nachfolgende Syntaxanalyse alle benötigten Eigenschaften der vorkommenden Wörter entnehmen kann.

Im wesentlichen ist `lexword/3` ein Sublexikon des vorhandenden Vollformenlexikons `word/2`. Neben dem jeweiligen Wort und seiner lexikalischen Information wird in `lexword/3` aber auch die Position des Worts im Eingabesatz festgehalten.

⟨*Tokenizer/lexer.pl*⟩≡

```
:- use_module('Tokenizer/zahlAtome.pl').

⟨Eingabe an der Konsole⟩
⟨Lexikalische Analyse⟩
⟨Sublexikon⟩
⟨Satzzeichen⟩
⟨Wortflexion⟩
```

4.1 Eingabe an der Konsole

Zum Testen ist es wieder nützlich, wenn man die Eingabe zur lexikalischen Analyse an der Konsole vornehmen kann:

⟨*Eingabe an der Konsole*⟩≡

```
lex :-
        write('Beende den Eingabesatz mit <Satzzeichen><Return>'),nl,
        read_sentence(user,Sentence,[]),
        stringToAtoms(Sentence,Atomlist),
        lexAnalyse(Atomlist),
        (ein(showLexAnalyse) -> true ; nl,listing(lexword/3)).
```

Dabei wird anschließend das erzeugte Sublexikon am Bildschirm angezeigt.

4.2 Aufbau des Sublexikons

Bevor die lexikalische Analyse der Eingabeliste erfolgt, wird das Sublexikon der vorangegangenen Eingabe gelöscht. Falls die Option `satzzeichen` eingeschaltet ist, werden auch die Satzzeichen aus der Eingabe entfernt.

⟨*Lexikalische Analyse*⟩≡

```
lexAnalyse(Wortliste) :-
        (gescheitert(lexAnalyse) -> retract(gescheitert(lexAnalyse)); true),
        retractall(lexword(_,_,_)),
        (ein(satzzeichen)
        -> Ws = Wortliste
        ; entferneSatzzeichen(Wortliste,Ws)),
        lexikalischeAnalyse(Ws,0),
        (ein(showLexAnalyse) -> (nl,listing(lexword/3)) ; true),
        (ein(filterLexAnalyse) -> (nl,filterLexword); true).
```

Dann wird zu jedem Token eine Reihe von Einträgen gebildet, die seine Position im Eingabesatz enthalten:

⟨*Lexikalische Analyse*⟩+≡
```
lexikalischeAnalyse([W|Ws],N) :-
        eintraege(W,N),
        M is N+1,
        lexikalischeAnalyse(Ws,M).
lexikalischeAnalyse([],_).
```

Wenn alle Wörter des Eingabesatzes im Lexikon nachgeschlagen wurden, kann man unnötige Einträge noch ausfiltern.

Mit `filterLexword/0` kann man interaktiv aus den Einträgen von `lexword/3` diejenigen aussondern, die man für die nachfolgende syntaktische Analyse behalten will. Das dient hauptsächlich zur Fehlersuche beim Parsen:

⟨*Lexikalische Analyse*⟩+≡
```
filterLexword :-
        lexword(I,W,Kat),
        write('Keep: '), write(lexword(I,W,Kat)), write(' y/n ? '),
        get(user,Char),  % y = 121
        (Char = 121 -> assert(clause(lexword(I,W,Kat))) ; true),
        retract(lexword(I,W,Kat)),
        fail.
filterLexword :-
        writeLexwords.
writeLexwords :-
        clause(lexword(I,W,Kat)),
        assertz(lexword(I,W,Kat)),
        retract(clause(lexword(I,W,Kat))),
        fail.
writeLexwords :-
        listing(lexword/3), nl.
```

Da in `lexword/3` die lexikalische Information mit der Position eines Wortes verbunden wird, kann man bei verschiedenen Vorkommen desselben Worts unterschiedliche Lesarten löschen.

Weil die lexikalische Mehrdeutigkeit die syntaktische Analyse verlangsamt, wäre es auch nützlich, wenn man aus dem Ergebnis der lexikalischen Analyse *automatisch* alle offenkundig unbrauchbaren Einträge entfernen würde. An bestimmten Vorkommen, z.B. am Satzanfang oder nach einem Komma, sind manche Lesarten eines Worts ja ausgeschlossen. Allgemeiner könnte man gegenseitige Unverträglichkeiten der lexikalischen Lesarten von Wörtern teilweise beseitigen, z.B. Lesarten als abgetrennte Verbpräfixe löschen, wenn kein entsprechendes trennbares Verb in der Eingabe war.

4.2.1 Einträge ins Sublexikon

Die Einträge in das Teillexikon `lexword/3` erfolgen durch

⟨*Sublexikon*⟩≡
```
eintrag(Wortform,Kategorie,Position) :-
        assertz(lexword(Position,Wortform,Kategorie)).
```

Die meisten Einträge werden dem Vollformenlexikon, `word/2`, entnommen. Da es nicht alle Token enthält, muß man für einige Sonderfälle die Einträge ins Sublexikon anders erzeugen:

(i) Zahlausdrücke stehen nicht im Lexikon, da es unendlich viele davon gibt. Falls ein Token eine Zahleigenschaft (vgl. den folgenden Abschnitt Hilfsprädikate) hat, dient diese als lexikalische Information:

⟨*Sublexikon*⟩+≡

```
eintraege(W,N) :-
        kardinalzahl(W),
        !,eintrag(W,kardinalzahl,N).
eintraege(W,N) :-
        ordinalzahl(W),
        !,eintrag(W,ordinalzahl,N).
eintraege(W,N) :-
        dezimalzahl(W),
        !,eintrag(W,dezimalzahl,N).
eintraege(W,N) :-
        punktzahl(W),
        !,eintrag(W,punktzahl,N).
```

Entsprechend müßte man natürlichsprachliche Zahlausdrücke behandeln. Ausdrücken wie `der 900ste` werden ebenfalls bisher ebenfalls keine lexikalsichen Kategorien zugeordnet.

(ii) Abkürzungen stehen in einem abgetrennten Abkürzungslexikon `abkuerzung(String,Kategorie)`, das im Abschntt 3.1.3 beschrieben wurde.

⟨*Sublexikon*⟩+≡

```
eintraege(W,N) :-
        name(W,String),
        abkuerzung(String,Kategorie),
        !,eintrag(W,Kategorie,N).
```

Das erste Argument von `abkuerzung/2` ist eine Zeichenreihe und kein Atom, da aus dem Abkürzungslexikon die Abkürzungstabelle für den Tokenizer gewonnen wird. (Es scheint effizienter, die Umwandlung zwischen Atom und Zeichenreihe hier zu machen als beim Tokenisieren.)

(iii) Satzzeichen stehen (normalerweise) nicht im Lexikon. Jedes Satzzeichen wird (ggf. zusätzlich zu Lexikoneinträgen) als eigene lexikalische Kategorie aufgefaßt:

⟨*Sublexikon*⟩+≡

```
eintraege(W,N) :-
        satzzeichen(W),
        !,(word(W,_) -> lexKats(W,W,N) ; true),
        eintrag(W,W,N).
```

Bei allen anderen Token schlagen wir im Lexikon `word/2` nach[3]:

(iv) Falls das Wort Lexikoneinträge hat, werden nur diese ins Sublexikon übernommen:

⟨*Sublexikon*⟩+≡

```
eintraege(W,N) :-
        word(W,_),
        !,lexKats(W,W,N).
```

[3] Das Vollformenlexikon muß dazu geladen sein oder zumindest einmal zum Lesen geöffnet werden.

(v) Falls das Wort keine Lexikoneinträge hat, mit einem Großbuchstaben beginnt und am Satzanfang steht[4], schlagen wir es mit kleinem Anfangsbuchstaben nach:

⟨*Sublexikon*⟩+≡
```
eintraege(W,0) :-            % N = 0 = Satzanfang
        name(W,[C|Chars]),
        grossbuchstabe(C),
        !,KleinC is C+32,
        name(KleinW,[KleinC|Chars]),
        lexKats(KleinW,W,0).
```

(vi) Falls das Wort keine Lexikoneinträge hat und klein geschrieben ist, wird durch `lexKats(W,W,N)` eine Fehlermeldung und ein Hilfseintrag erzeugt:

⟨*Sublexikon*⟩+≡
```
eintraege(W,N) :-
        lexKats(W,W,N).
```

4.2.2 Suche im Vollformenlexikon

Mit `lexKats(Orig+,W+,Position+)` werden Lexikoneinträge unter einer Variante `Orig`, z.B. der Kleinschreibung von `W`, gesucht und als Einträge zu `W` ins Sublexikon übernommen.

Falls die Suche scheitert, wird das am Bildschirm gemeldet, ein Hilfseintrag erzeugt und festgehalten, daß die lexikalische Analyse gescheitert ist[5]:

⟨*Sublexikon*⟩+≡
```
lexKats(Orig,W,N) :-
        not(word(Orig,_)),
        write(user,'\nUnbekanntes Wort: '),write(user,W),
        !,eintrag(W,W,N),
        (gescheitert(lexAnalyse) -> true
          ; assertz(gescheitert(lexAnalyse))).
```

Sonst werden alle Einträge zu `Orig` durch eine Schleife aus dem Lexikon extrahiert und zu `W` im Sublexikon eingetragen:

⟨*Sublexikon*⟩+≡
```
lexKats(Orig,W,N) :-
        word(Orig,Cat),
        eintrag(W,Cat,N),
        fail.
lexKats(_Orig,_W,_N).
```

Hilfsprädikate der lexikalischen Analyse

Satzzeichen sind hier die Atome, die bei der Tokenisierung als Sonderzeichen behandelt wurden, d.h. die zu den folgenden gehören:

⟨*Satzzeichen*⟩≡
```
satzzeichen('.').
satzzeichen(',').
```

[4]Das ist zu restriktiv, wenn nominalisierte Verben (`das Schweigen`), Pronomina (`das Du`) und Präpositionen (`das Für und Wider`) nicht im Lexikon stehen.

[5]Die nachfolgende syntaktische Analyse sollte das wissen!

```
satzzeichen(';').
satzzeichen(':').
satzzeichen('!').
satzzeichen('?').
satzzeichen('"').
satzzeichen('(').
satzzeichen(')').
```

Da Satzzeichenfehler weit verbreitet sind, ist es sinnvoll, daß man bei der lexikalischen Analyse alle Satzzeichen (ggf.: bis auf das am Ende) aus der Eingabe entfernen kann:

⟨*Lexikalische Analyse*⟩+≡

```
% entferneSatzzeichen([S,A|Atomliste],Atome) :-
%       satzzeichen(S),
%       !,entferneSatzzeichen([A|Atomliste],Atome).
entferneSatzzeichen([S|Atomliste],Atome) :-
        satzzeichen(S),
        !,entferneSatzzeichen(Atomliste,Atome).
entferneSatzzeichen([S|Atomliste],[S|Liste]) :-
        entferneSatzzeichen(Atomliste,Liste).
entferneSatzzeichen([],[]).
```

Ohne Satzzeichen in der Eingabe ist die nachfolgende syntaktische Analyse zwar schwieriger, kann aber dann auch Eingaben mit Satzzeichenfehlern annehmen. Da das Satzzeichen am Satzende normalerweise korrekt gesetzt wird und die Satzart bestimmt, wird es hier nicht entfernt.

Zahlen: Bei der syntaktischen Analyse muß man von Prolog-Atomen feststellen können, ob sie Zahlen darstellen, die z.B. als Adjektive gelten. Bei der lexikalischen Analyse unterscheiden wir folgende Zahleigenschaften oder *Zahlatome*:

⟨*Tokenizer/zahlAtome.pl*⟩≡

```
:- module('zahlatome',[kardinalzahl/1,ordinalzahl/1,dezimalzahl/1,punktzahl/1,zahl/1]).

kardinalzahl(Z) :-
        name(Z,L), kardinalzahlString(L).
ordinalzahl(Z) :-
        name(Z,L), ordinalzahlString(L).
dezimalzahl(Z) :-
        name(Z,L), dezimalzahlString(L).
punktzahl(Z) :-
        name(Z,L), punktzahlString(L).

zahl(Z) :- kardinalzahl(Z),!.
zahl(Z) :- ordinalzahl(Z),!.
zahl(Z) :- dezimalzahl(Z),!.
zahl(Z) :- punktzahl(Z),!.
```

Die Zahleigenschaften der Atome werden auf entsprechende Eigenschaften von Zahlzeichenreihen zurückgeführt:

⟨*Tokenizer/zahlAtome.pl*⟩+≡

```
zahlString(Z) :- kardinalzahlString(Z),!.
zahlString(Z) :- ordinalzahlString(Z),!.
zahlString(Z) :- dezimalzahlString(Z),!.
zahlString(Z) :- punktzahlString(Z).
```

```
kardinalzahlString([Z]) :- member(Z,"0123456789"), !.
kardinalzahlString([Z|Rest]) :-
        member(Z,"0123456789"), kardinalzahlString(Rest).
ziffer(Z) :-
        member(Z,"0123456789").

ordinalzahlString(O) :- kurzeOrdinalzahlString(O),!.
ordinalzahlString(O) :- langeOrdinalzahlString(O).

kurzeOrdinalzahlString(O) :-              % 23.
        append(K,".",O),
        kardinalzahlString(K).
langeOrdinalzahlString(O) :-              % 134.5.78.
        append(Kurz,Rest,O),
        kurzeOrdinalzahlString(Kurz),!,
        ordinalzahlString(Rest).

punktzahlString(P) :-                     % 123.345.67
        append(Z,N,P),
        ordinalzahlString(Z),
        kardinalzahlString(N).

dezimalzahlString(D) :- kurzeDezimalzahlString(D),!.
dezimalzahlString(D) :- langeDezimalzahlString(D).

kurzeDezimalzahlString(Z) :-              % 67,578
        append(N,KM,Z),
        kardinalzahlString(N),
        append(",",M,KM),
        kardinalzahlString(M).
langeDezimalzahlString(Z) :-              % 123.345.67,578
        append(P,KN,Z),
        punktzahlString(P),
        append(",",N,KN),
        kardinalzahlString(N).
```

Beachte: Bruchzahlen wie `3/10` werden nicht erfaßt; Bruchzeichen werden wie normale Symbole behandelt. Natürlichsprachliche Zahl*wörter* wie `dreiundsiebzig` können nicht im Lexikon gesammelt werden, wenn man keine Beschränkung an die Größe der ausgedrückten Zahl machen will. Sie werden hier nicht behandelt.

4.3 Format des Vollformenlexikons

In einem von der Grammatik getrennten Vollformen-*Lexikon* wird durch Prolog-Fakten der Form `word(<Wort>,<Kategorie>)` jedem Wort eine *Wortart* oder *lexikalische Kategorie* zugeordnet.

Die *Wörter* werden durch Prolog-Atome dargestellt, also wie `und` mit kleinen Anfangsbuchstaben oder wie `'Hund'` in einfachen Anführungszeichen.

Beispiel 4.1 Das folgende Lexikon hat als Kategorien Prolog-*Atome*, die keine Aufgliederung nach morphologischen oder anderen Kriterien enthalten:

⟨*Lexika/lexikon.1.wrd*⟩≡
```
word(der,det).
word(ein,det).

word('Lied',n).
word('Hamster',n).
word('Grube',n).

word(pfeift,v).

word(arme,a).

word(in,p).
word(für,p).

word(sehr,grad).

word(und,conj).
word(oder,conj).
```

Im allgemeinen werden aber als lexikalische Kateogorien beliebige Prolog-*Terme* zugelassen, z.B. `n(nom,sg)`.

Die Lexikoneinträge können dann auch *schematisch* sein, z.B.

```
word('Hund',n(Kasus,sg)) :- Kasus = nom ; Kasus = akk.
```

Zugriffsprädikate auf das Stammlexikon

Zum Parsen mit einer größeren Grammatiken braucht man ein größeres Vollformenlexikon. Dafür ist es sinnvoll, syntaktische Information zu den Wörtern bei ihren Stammformen anzugeben und nicht bei jeder Vollform zu wiederholen, damit die Korrektur dieser Information erleichtert wird.

Dies läßt sich dadurch erreichen, daß man das vom Parser erwartete Vollformenlexikon `word/2` aus einem um Stammeinträge angereicherten (und bei den Kategorien um Unterklassenspezifikation vereinfachten) Lexikon `word/3` mit Einträgen `word(Vollform,Pseudokategorie,Stamm)` ableitet.

Die zusätzliche syntaktische Information, z.B. die Angabe der Komplementrahmen, wird dann in einem getrennten Stammlexikon `stamm(Wortkategorie, Stamm, Information)` formuliert.

⟨*Tokenizer/lexer.pl*⟩+≡
```
word(W,Cat) :-
        word(W,CatCIS,Stamm),
        CatCIS =.. [Kat,Art,Form],
        (komplementrahmen(Kat,Stamm,Rahmen)
        -> ((Kat = v, Cat =.. [Kat,[Rahmen|Art],Form])
           ; ((Kat = n; Kat = na),
                 append(Art,[Rahmen],ArtNeu),
                 Cat =.. [Kat,ArtNeu,Form]))
        ; Cat = CatCIS).
```

Durch verschiedene Zugriffsprädikate wie `komplementrahmen/3` soll erreicht werden, daß man durch deren Anpassung den Parser in einfacher Weise an Änderungen der Stammeinträge anpassen kann.

⟨*Tokenizer/lexer.pl*⟩+≡

```
komplementrahmen(Kat,Stamm,Rahmen) :-
        stamm(Kat,Stamm,Rahmen).
```

Das Format der Stammeinträge ist vorläufig; sobald es geändert wird, müssen Zugriffsprädikate wie `komplementrahmen/3` angepaßt werden.

Flexionsprogramm für Wörter

Mit dem Stamm (ohne Komplemente und Klassenparameter) können wir aus dem Vollformenlexikon, das natürlich aus einem Stammformenlexikon durch ein Flexionsprogramm erzeugt wurde, ein Flexionsprogramm zurückgewinnen:

⟨*Lexika/umformatieren.pl*⟩≡

```
flektiere(Vollform1, Kategorie1, Form2, Vollform2) :-
        word(Vollform1, Kategorie1, Stamm),
        Kategorie1 =.. [Kat,Art,Form1],
        Kategorie2 =.. [Kat,Art,Form2],
        word(Vollform2, Kategorie2, Stamm).
```

Das wird später zur Transformation von analysierten Phrasen benutzt, um die darin vorkommenden Wörter umzuwandeln.

5 Syntaktische Analyse

Die syntaktische Analyse versucht, festzustellen, ob der von der Tokenisierung in einzelne Token zerlegte Eingabesatz mit Hilfe des von der lexikalischen Analyse erstellten Teillexikons und einer gegebenen Grammatik nach den Regeln der Grammatik gebildet werden kann.

Jede Möglichkeit, die Eingabe gemäß den Grammatikregeln aus geeigneten lexikalischen Lesarten der Token aufzubauen, ist eine *syntaktische Struktur* dieses Satzes.

5.1 Format der Grammatikregeln

Wir verwenden als Format der Grammatik die *Definite Clause Grammar* von Prolog. Die *Syntaxregeln* haben die Form

`A --> B,...,C.`   (eigentlich 3 mal - geschrieben!)

worin `A`,`B`,..., `C` Prolog-Terme sind, aber `A` keine Liste ist.

Ein Term `A` ist eine *grammatische Kategorie*, wenn er in einer Regel auf der linken Seite von `-->` auftritt, in der rechts nicht nur `[]` steht.

Eine *Löschregel* ist eine Regel der Form `A --> []`, bei der rechts die leere Liste, `[]`, steht.

Ein Term `B` außer `[]`, der höchstens in Löschregeln auf den linken Seiten von `-->` auftritt, ist eine *lexikalische Kategorie* oder *Wortart*.

Eine *Kettenregel* ist eine Regel der Form `A --> B`, bei der `B` eine grammatische oder lexikalische Kategorie ist.

Wir erlauben z.T. folgende *Pseudo-DCG-Regeln*, d.h. Grammatikregeln `A --> B :- C.`, deren rechte Seiten `B` wie folgt aufgebaut sein dürfen:

(i) `B` ist ein Tupel `B1,...,Bn`, d.h. eine Konjunktion `(B1,...,Bn)` von Prolog-Termen (außer Variablen),

(ii) `Bi` ist ein einfacher Prolog-Term `P(T1,...,Tk)` mit einem Funktionszeichen `P` und Termen `T1,...,Tk`, oder

(iii) `Bi` ist eine Disjunktion `D1;...;Dm` von einfachen Prolog-Termen, oder

(iv) `Bi` ist von der Form `{D}` mit einer Disjunktion `D` von einfachen Prolog-Termen.

Hierbei ist `P(T1,...,Tk)` ein *einfacher* Prolog-Term, wenn das Funktionszeichen `P` weder die Konjunkion `,` noch die Disjunktion `;`, noch die Mengenklammer `{.}` ist.

Durch Regeln der Form `A --> B,(C;D;E),F.` können mehrere Grammatikregeln `A --> B,C,F.`,...,`A --> B,E,F.` durch *Linksfaktorisierung* eines gemeinsamen Präfixes `B` in *alternativen Konstituenten* `(C;D;E)` zusammengefaßt werden.

Durch Regeln der Form `A --> B,{C},D.` können *optionale* Konstituenten `C` angegeben werden; diese Regel entspricht der Regel `A --> B,(C;[]),E.` und besagt, daß `C`-Phrasen an *solchen Vorkommen* gelöscht werden können.

Als einen Sonderfall behandeln wir noch Kategorien der Form `[W|Ws]` , mit denen *konkrete Wortfolgen* als Konstituentenkategorien erlaubt werden. (Warnung: Deren Behandlung ist nicht vollständig implementiert.)

In jeder Grammatik wird durch Fakten der Form

```
start_symbol(A).
```

eine oder mehrere *Hauptkategorien* festgelegt. Ein Parser soll später nur Ausdrücke dieser Kategorien erkennen. Außerdem muß durch

```
auxiliary_predicates[start_symbol/1]
```

mindestens das Prädikat `start_symbol/1`, aber normalerweise weitere Prädikate wie die Definition von Merkmalbereichen, Kongruenzprädikaten usw., als *Hilfsprädikat* der Grammatik erklärt werden. Diese Hilfsprädikate werden später bei der Anpassung der Grammatik mdiesen (vgl. Abschnitt 5.3.3) an den jeweiligen Parser übernommen.

Wir schreiben Grammatiken in Dateien mit dem Suffix `.grm`, damit die Bedienung des Systems durch den Editor `Emacs` unterstützt werden kann.

5.1.1 Grammatiken mit atomaren Kategorien

Im einfachsten Fall sind die grammatischen Kategorien Prolog-*Atome*.

Beispiel 5.1 Die Grammatik enthält zuerst eine Deklaration, die `-->` zu einem Infix-Operator von Prolog macht:

⟨*Grammatiken/cfg.1.grm*⟩≡

```
:- op(1100,xfx,'-->').
```

Dann folgt die Festlegung der Hauptkategorien und Regeln:

⟨*Grammatiken/cfg.1.grm*⟩+≡

```
auxiliary_predicates([start_symbol/1]).

start_symbol(s).
start_symbol(np).

% Rules:

s --> np, vp.

np --> det, ap, n.
np --> np, conj, np.

vp --> v.
vp --> v, np.
vp --> v, np, av.

av --> adv.
av --> pp.
av --> [].

pp --> p, np.

ap --> a.
ap --> grad, ap.
ap --> ap, conj, ap.

a --> [].
grad --> [].
```

Die grammatischen Kategorien sind `s,np,vp,av,pp,ap`, die lexikalischen sind `det,n,a,v,adv,p,conj,grad`. Die Grammatik hat Löschregeln und Kettenregeln.

Ein zu dieser Grammatik passendes, d.h. nur diese lexikalischen Kategorien verwendendes, Lexikon wurde in Beispiel 4.1 angegeben.

5.1.2 Grammatiken mit komplexen Kategorien

Normalerweise sind die grammatischen Kategorien Prolog-*Terme*, deren Argumente zur Unterklassifizierung durch morphologische Merkmale (Eigenschaften von Wortvorkommen) oder syntaktische Merkmale (Eigenschaften wie Stelligkeit von Verben, Positionsbeschränkungen etc.) dienen.

In diesem Fall lassen sich auch *schematische* oder bedingte Grammatikregeln formulieren.

Beispiel 5.2 Als Kategorien kann man zweistellige Prolog-Terme der Form `Kategorie(Unterklasse,Form)` verwenden, wobei sowohl die Unterklasse als auch die Form durch eine Liste von Merkmal(wert)en dargestellt werden.

In der folgenden Grammatik wird bei Verben und Verbalphrasen die Unterklasse durch nur *ein* Merkmal definiert, die Liste seiner Komplemente (inklusive Subjekt, dem ersten Komplement), und die Form nur durch Numerus und Person. Transitive Verbformen in der dritten Person im Singular haben daher die Kategorie `v([[nom,akk]],[sg,3])`:

⟨*Grammatiken/cfg.2.grm*⟩≡

```
:- op(1100,xfx,'-->').
auxiliary_predicates([start_symbol/1,genus/1,numerus/1,kasus/1,person/1]).

start_symbol(s).
start_symbol(np([Gen],[Num,Kas])) :-
        genus(Gen), numerus(Num), kasus(Kas).

genus(mask).
genus(fem).
genus(neut).

numerus(sg).
numerus(pl).

kasus(nom).
kasus(gen).
kasus(dat).
kasus(akk).

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

s --> np([Gen],[Num,nom]), vp([[nom]],[Num,3])
        :- genus(Gen), numerus(Num).

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

```
np([mask],[pl,nom]) -->
        np([mask],[Num,nom]), conj, np([mask],[Num,nom])
        :- numerus(Num).

vp([[nom]],[Num,Pers]) --> v([[nom]],[Num,Pers])
        :- numerus(Num), person(Pers).

vp([[nom]],[Num,Pers]) -->
        v([[nom,akk]],[Num,Pers]), np([Gen],[Num2,akk])
        :- numerus(Num), person(Pers), genus(Gen), numerus(Num2).

vp([[nom]],[Num,Pers]) -->
        v([[nom,akk]],[Num,Pers]), np([Gen],[Num2,akk]), av(lok)
        :- numerus(Num), person(Pers), genus(Gen), numerus(Num2).

av(lok) --> adv(lok).
av(lok) --> pp(lok).
av(lok) --> [].

pp(lok) --> p([[dat],lok]), np([Gen],[Num,dat])
        :- genus(Gen), numerus(Num).

ap([Gen,Num,Kas]) --> a([Gen,Num,Kas])
        :- genus(Gen),numerus(Num),kasus(Kas).
ap([mask,sg,nom]) --> grad, ap([mask,sg,nom]).
ap([mask,sg,nom]) --> ap([mask,sg,nom]), conj, ap([mask,sg,nom]).

a([Gen,Num,Kas]) --> []
        :- genus(Gen),numerus(Num),kasus(Kas).
grad --> [].
```

Ein dazu passendes Lexikon ist die entsprechende Änderung des vorherigen Lexikons:

⟨*Lexika/lexikon.2.wrd*⟩≡

```
word(der,det([mask,sg,nom])).
word(der,det([fem,sg,dat])).
word(der,det([mask,pl,gen])).
word(ein,det([mask,sg,nom])).
word(ein,det([neut,sg,akk])).

word('Lied',n([neut],[sg,akk])).
word('Hamster',n([mask],[sg,nom])).
word('Hamster',n([mask],[sg,akk])).
word('Hamster',n([mask],[pl,gen])).
word('Grube',n([fem],[sg,dat])).

word(pfeift,v([[nom,akk]],[sg,3])).
word(pfeift,v([[nom]],[sg,3])).

word(arme,a([mask,sg,nom])).

word(vor,p([[dat],lok])).
word(in,p([[dat],lok])).
```

```
word(sehr,grad).

word(und,conj).
word(oder,conj).
```

Bemerkung: Wenn in allen Regeln die Bereichsbedingungen angegeben sind, erhält man de facto eine große kontextfreie Grammatik; die Parser werden dann viel zu langsam, da ständig unnötige Tests durchgeführt werden. Daher sind in den Grammatiken später entsprechende Zeilen auskommentiert.

Vorläufige Faustregel: die Bereichsprädikate sollten *nur dann* angegeben werden, wenn

(i) sie nicht schon daraus folgen, daß sie in Konstituenten der rechten Regelseite auftreten und für jede Phrase der entsprechenden Kategorie erfüllt sind, und

(ii) die Merkmalterme der linken Regelseite von Prolog nicht berechnet werden können, ohne daß gewisse Variablen der rechten Regelseite belegt sind.

5.1.3 Grammatiken mit Metaregeln

Metaregeln sind Ausdrücke, aus denen Syntaxregeln erzeugt werden. Beispielsweise sind Satzregeln, die die Stelligkeit des Verbs offen lassen, solche Metaregeln, etwa Regeln der „X-bar-Theorie" wie `X' --> ... X ...` die besagen sollen, daß Ausdrücke der Kategorie `X'` aus einem Kopfausdruck der Kategorie `X` und (an den Stellen ...) vollständigen Phrasen bestehen. Hier soll eine besondere Beziehung zwischen `X'` und `X` bestehen, z.B. die zwischen Verbalphrase `X'` und dem finiten Verb `X`.

Um solche Regeln benutzen zu können, muß ein Parser sie in eine Menge normaler Syntaxregeln übersetzen.

(Eine entsprechende Beispielgrammatik folgt noch.)

5.2 Aufruf der Syntaxanalyse

Einige Aufgaben sind bei allen Analyseverfahren gleich. Wir lagern sie in eine eigene Datei aus:

⟨*Parser/main.pl*⟩≡

```
:- ensure_loaded('Parser/PrintTree/showtree.pl').
⟨parse⟩
⟨erkenne⟩
```

5.2.1 Einlesen von der Konsole

Wie bei der Tokenisierung und der lexikalischen Analyse braucht man auch für die syntaktische Analyse eine Möglichkeit, das Analyseverfahren von der Konsole zu starten.

Dazu dient `parse/0`, das sich auf ein in `auswahl.pl` gewähltes Analyseverfahren `parse/2` stützt.

(i) Nach dem Einlesen der Eingabe erfolgt zuerst die Tokenisierung und dann die lexikalische Analyse.

Dabei werden die Satzzeichen entfernt, außer wenn die Option `satzzeichen` eingeschaltet ist.[6]

(ii) Nach der lexikalischen Analyse wird überprüft, ob die Wortarten auch Kategorien der Grammatik sind.

(iii) Falls die lexikalische Analyse gescheitert, d.h. auf unbekannte Eingabewörter gestoßen ist, oder die gefundenen Wortarten der Grammatik nicht bekannt sind, wird am Bildschirm gemeldet, daß keine Syntaxanalyse durchgeführt wurde, und dann abgebrochen.

(iv) Sonst wird die Eingabe syntaktisch analysiert: durch `baeume(Wortfolge+,Baeume-)` wird eine Liste `Baeume` aller syntaktischen Strukturen der Wortfolge ermittelt.

(v) Falls es mindestens eine Analyse gibt und die Option `showTrees` eingeschaltet ist, werden alle Bäume am Bildschirm ausgegeben. Die Baumdarstellungen werden aus `'Parser/PrintTree/showtree.pl'` hinzugeladen.

⟨*parse*⟩≡

```
parse :-
        write(user,'Beende den Eingabesatz mit <Satzzeichen><Return>'),nl,
        read_sentence(user,String,[]),
        stringToAtoms(String,Atomlist),
        (ein(satzzeichen)
        -> Wortfolge = Atomlist
        ; entferneSatzzeichen(Atomlist,Wortfolge)),
        lexAnalyse(Wortfolge),
        (ein(pruefeLexWord) -> erkenneWortarten ; true),
        (gescheitert(lexAnalyse)
        -> (nl, write(user,'Keine syntaktische Analyse durchgeführt.'))
        ; (baeume(Wortfolge,Trees)
          -> (ein(showTrees)
              -> (ein(asciiTrees) -> showASCIItrees(Trees) ; showTeXtrees(Trees))
              ; (nl, writeq(Trees)))
          ; (nl, write(user,'Keine syntaktische Analyse gefunden.'))
          )
        ).
```

[6] Leider kann das doch nicht in der lexikalischen Analyse gemacht werden, denn dann würde `baeume` mit einer längeren Liste von Token arbeiten.

Nach der lexikalschen Analyse der Eingabe wird mit einem aus der Grammatik vorberechneten Prädikat `lexCat/1` (vgl. Abschnitt 5.3.4) geprüft, ob die gefundenen Wortkategorien der Grammatik bekannt sind. Alle Einträge mit unbekannter Kategorie werden entfernt; wenn ein Wort *nur* unbekannte Kategorien hat, wird mit `add(gescheitert(lexAnalyse))` ein Analyseversuch verhindert, ansonsten wird mit den verbliebenen Einträgen eine Analyse versucht[7]:

⟨*parse*⟩+≡
```
erkenneWortarten :-
        lexword(I,W,Cat),
        ((not(lexCat(Cat)),not(lexword(I,Cat,Cat)))
        -> nl,write('Unbekannte Kategorie '),
           writeq(Cat),
            write(' des Worts \`'),write(W),write('\' gelöscht.'),
            retract(lexword(I,W,Cat)),
            ((lexword(I,W,CatW),lexCat(CatW)) -> true ; add(gescheitert(lexAnalyse))),
            (ein(verbose) -> nl ; true)
        ; true),
        fail.
erkenneWortarten :- nl.
```

Wir nehmen an, daß es mindestens eine Analyse gibt und ermitteln die Menge aller von `parse(Wortfolge+,Baum-)` gefundenen Bäume. Der benötigte Aufwand wird am Bildschirm angezeigt, falls die Option `showProfile` eingeschaltet ist:

⟨*parse*⟩+≡
```
baeume(Wortfolge, Trees) :-
        ein(showProfile) ->
         profile((setof(Baum, parse(Wortfolge, Baum), Trees),nl),plain,5)
        ; setof(Baum, parse(Wortfolge, Baum), Trees).
```

Falls `setof/3` von Prolog keine Lösungen findet, scheitert es.

Ist man nicht an den Analysen der Eingabe interessiert, sondern nur an ihrer syntaktischen Kategorie, so benutzt man das einfachere Prädikat

⟨*erkenne*⟩≡
```
erkenne :-
        write(user,'Beende den Eingabesatz mit <Satzzeichen><Return>'),nl,
        read_sentence(user,String,[]),
        stringToAtoms(String,Atomlist),
        (ein(satzzeichen)
        -> Wortfolge = Atomlist
        ; entferneSatzzeichen(Atomlist,Wortfolge)),
        lexAnalyse(Wortfolge),
        (ein(pruefeLexWord) -> erkenneWortarten ; true),
        (gescheitert(lexAnalyse)
        -> (nl,write(user,'Keine syntaktische Analyse durchgeführt.'))
        ; (kategorien(Wortfolge,Kategorien)
          -> (nl,writeq(Kategorien))
          ; (nl,write('Keine syntaktischen Analysen gefunden.')))
        ).
```

[7] Man kann daher im Lexikon auch Einträge haben, mit denen erst eine Erweiterung der Grammatik umgehen kann.

Durch `kategorien(Wortfolge+,Kategorien-)` ermitteln wir alle von `erkenne(Wortfolge+,Kategorie-)` gefundenen Kategorien. Der benötigte Aufwand wird am Bildschirm angezeigt, falls die Option `showProfile` eingeschaltet ist:

⟨*erkenne*⟩+≡
```
kategorien(Wortfolge, Kategorien) :-
        ein(showProfile) ->
         profile((setof(Kat, erkenne(Wortfolge, Kat), Kategorien),nl),plain,5)
        ; setof(Kat, erkenne(Wortfolge, Kat), Kategorien).
```

Das Prädikat `erkenne/2` ist eine vereinfachte Version des jeweiligen Parsers, bei der (normalerweise) keine syntaktischen Strukturen aufgebaut werden.

Alternativ kann man, um die syntaktischen Kategorien zu finden, natürlich auch eine vollständige Syntaxanalyse der Eingabe durchführen und von den gefundenen Syntaxbäumen nur die Markierungen an den Wurzeln ausgeben:

⟨*erkenne (alternativ; unbenutzt)*⟩≡
```
kategorien(Satz, Kategorien) :-
        baeume(Satz, Trees),
        wurzeln(Trees, Kategorien).

wurzeln([[Kat|_Struktur]|Trees],[Kat|Kats]) :-
        wurzeln(Trees,Kats).
wurzeln([],[]).
```

Hier wird vorausgesetzt, daß Syntaxbäume als *Listen* `[Cat|Trees]` aus einer grammatischen Kategorie `Cat` und den Teilbäumen `Trees` dargestellt werden.

5.2.2 Einlesen von einer Datei: Erkennen von Phrasen

Man braucht ein Programm, mit dem man eine Liste vorgegebener Beispielphrasen auf grammatische Korrektheit überprüfen und die Ergebnisse in einer Datei festhalten kann. Die Beispielphrasen müssen in der Datei durch einen Punkt abgeschlossen werden, sodaß das Programm zur Erkennung von Satzenden aus Abschnitt 3.1 auch zur Erkennung von Phrasenenden benutzt werden kann. Das folgende Programm testet, ob die Beispielphrasen erkannt werden, und gibt die gefundenen Kategorien (nicht die Analysen) aus:

⟨*Parser/phrasenerkenner.pl*⟩≡
```
:- ensure_loaded('Tokenizer/tokenizer.pl').

erkennePhrasen :-
        write(user,'Eingabedatei: '),
        read(Eingabedatei),
        grammar_parser_datum_suffix(Suffix),
        concat_atom([Eingabedatei,'.erk',Suffix],Ausgabedatei),
        write(user,'Ausgabedatei: '),write(Ausgabedatei),nl,
        erkennePhrasen(Eingabedatei,Ausgabedatei).

erkennePhrasen(Eingabedatei,Ausgabedatei) :-
        open(Eingabedatei,read,InStream),
        open(Ausgabedatei,write,OutStream),
        erkennePhrasen(InStream,OutStream,0),
```

```
        close(InStream),
        close(OutStream),
        nl,write(user,'Done.').

erkennePhrasen(InStream,OutStream,N) :-
        get0(InStream,Char),
        (dateiende(Char)
        -> true
        ; read_sentence(InStream,Sentence,[Char]),
            stringToAtoms(Sentence,Atomlist),
            (ein(satzzeichen)
            -> Wortfolge = Atomlist
            ; entferneSatzzeichen(Atomlist,Wortfolge)),
            lexAnalyse(Wortfolge),
            write(OutStream,N),write(OutStream,': '),
            name(Phrase,Sentence),write(OutStream,Phrase),
            (gescheitert(lexAnalyse)
            -> (write(OutStream,' - Lexikalische Analyse ist gescheitert.\n'))
            ; (kategorien(Wortfolge,Kategorien) ->
                  write(OutStream,' '),writeq(OutStream,Kategorien),write(OutStream,'\n')
              ; write(OutStream,' - Ungrammatisch.\n'))
            ),
            write(user,'.'),flush_output(user),
            M is N + 1,
            erkennePhrasen(InStream,OutStream,M)).
```

Es wird vorausgesetzt, daß passende Lexika und eine für den geladenen Parser vorbereitete Grammatik geladen sind. (Die Suffixe werden in z.Zt. in der Auswahldatei `auswahl.pl` definiert.)

Man braucht natürlich auch Programme, um *Teil*phrasen von Sätzen zu erkennen. Dazu kann man spezielle Grammatikregeln schreiben, mit denen alle Kategorien von Wörtern im Links- und Rechtskontext der gesuchten Phrasen ignoriert werden können.

Der Nachteil ist, daß dies nicht davor schützt, daß sich die lexikalische Ambiguität der Wörter im Links- und Rechtskontext der gesuchten Phrasen in einer hohen Mehrdeutigkeit der (Pseudo-)Sätze niederschlägt. (Natürlich kann man aus den erzeugten Parsetabellen die Teilphrasen der gewünschten Kategorien extrahieren, aber der Analyseaufwand wird nicht geringer.)

Wahlweise wollen wir auch die gefundenen Analysen auf die Ausgabedatei schreiben:

⟨*Parser/phrasenerkenner.pl*⟩+≡

```
analysierePhrasen :-
        write(user,'Eingabedatei: '),
        read(Eingabedatei),
        grammar_parser_datum_suffix(Suffix),
        concat_atom([Eingabedatei,'.ana',Suffix],Ausgabedatei),
        write(user,'Ausgabedatei: '),write(Ausgabedatei),nl,
        analysierePhrasen(Eingabedatei,Ausgabedatei).

analysierePhrasen(Eingabedatei,Ausgabedatei) :-
        open(Eingabedatei,read,InStream),
        tell(Ausgabedatei),
        einschalten(asciiTrees),
        analysiere(InStream,0),
        close(InStream),
```

```
        told,
        nl,write(user,'Done.').

analysiere(InStream,N) :-
        get0(InStream,Char),
        (dateiende(Char)
        -> true
        ; read_sentence(InStream,Sentence,[Char]),
            stringToAtoms(Sentence,Atomlist),
            (ein(satzzeichen)
            -> Wortfolge = Atomlist
            ; entferneSatzzeichen(Atomlist,Wortfolge)),
            lexAnalyse(Wortfolge),
            write(N),write(': '),
            name(Phrase,Sentence),write(Phrase),
            (gescheitert(lexAnalyse)
            -> (write(' - Lexikalische Analyse ist gescheitert.\n'))
            ; (baeume(Wortfolge,Baeume) ->
                  showAsciiTrees(Baeume),write('\n')
              ; write(' - Ungrammatisch.\n'))
            ),
            write(user,'.'),flush_output(user),
            M is N + 1,
            analysiere(InStream,M)).
```

5.3 Tabellenverfahren zur Syntaxanalyse

Ein *Tabellenverfahren*, englisch: *Chart-Parser*, baut für die Eingabe eine Tabelle auf, in der für jedes Teilwort alle seine grammatischen Kategorien eingetragen werden.

Dem Teilstück $w(i,j)$ von Position i bis j in der Eingabe w entspricht ein Feld $F(i,j)$ der Tabelle, in dem die Kategorien von $w(i,j)$ gesammelt werden. Rekursiv trägt man zuerst die Kategorien der Teilstücke von $w(i,j)$ ein, d.h. man füllt alle Felder $F(i',j')$ mit $i \leq i' < j' \leq j$. Aus der lexikalischen Analyse weiß man, was in die Felder $w(i,i+1)$ gehört.

Wenn die Eingabe w die Länge $n+1$ hat, stehen schließlich alle grammatischen Kategorien von $w = w(0,n)$ im Feld $F(0,n)$. Daran kann man also erkennen, ob w ein korrekter Satz ist.

Tabellenverfahren sind meist nur *Erkennungsverfahren*. Aus der Tabelle zu w kann man aber alle Analysen rekonstruieren (bis auf unwichtige Unterschiede) - man rekonstruiert, *warum* eine Kategorie in ein Feld eingetragen wird.

Wir machen Tabellenverfahren zu Analyseverfahren, indem wir in die Felder der Tabelle alle *Analysen* der Teilausdrücke schreiben. Dazu muß man gewisse Unterschiede zwischen Analysen ignorieren – sonst würden es unendlich viele.

5.3.1 Earley-Parser ohne Vorberechnungen

Ein für alle, auch nicht-deterministische, kontextfreie Grammatiken taugliches Tabellenverfahren ist das von Jay Earley (1970). Eine Implementierung dieses Erkennungsverfahrens in Prolog wird im Buch „Parsing“, S. 143ff, von Naumann/Langer beschrieben.

Wir erweitern diese Implementierung zu einem Analyseverfahren, indem wir Analysen, also Syntaxbäume, in die Felder der Tabelle schreiben. Die Felder $F(i,j)$ zu Positionen $i \leq j$ in der Eingabe werden in Unterfelder oder *Kanten*

$$F(i, j, A \to \alpha \cdot \beta)$$

aufgespalten. Dabei ist $A \to \alpha\beta$ eine Grammatikregel, deren rechte Regelseite in einen *geschlossen* Anfang α und einen noch *offenen* Rest β zerlegt ist. Eine (nicht leere) Kante

$$i \stackrel{A \to \alpha \cdot \beta}{\longrightarrow} j$$

enthält Baumfolgen $\vec{t}$, deren Wurzeln mit den Kategorien α markiert sind und deren Blätter die Wörter des Eingabestücks $w(i, j)$ bilden. Falls β leer ist, ist der Baum mit Wurzel A und Teilbaumfolge $\vec{t}$ eine *vollständige* Analyse von $w(i, j)$, andernfalls eine *partielle* Analyse von $w(i, j)$. (Man spricht auch von *passiven* bzw. *aktiven* Kanten.)

Der Aufbau der Kanten zu $w = a_0 \cdots a_n$ erfolgt durch

(i) Voraussagen brauchbarer Regeln:

$$\frac{}{0 \stackrel{S \to \cdot \alpha}{\longrightarrow} 0} \quad \text{(Start)} \qquad \frac{i \stackrel{C \to \gamma \cdot A\delta}{\longrightarrow} j}{j \stackrel{A \to \cdot \alpha}{\longrightarrow} j} \quad \text{(Predict)}$$

(ii) Lesen der Eingabewörter:

$$\frac{\mathit{lexword}(a_i, A)}{i \stackrel{A \to a_i \cdot}{\longrightarrow} i+1} \quad \text{(Scan)}$$

(iii) Abschluß unter Verkettung von Kanten:

$$\frac{i \stackrel{A \to \alpha \cdot B\gamma}{\longrightarrow} j \qquad j \stackrel{B \to \beta \cdot}{\longrightarrow} k}{i \stackrel{A \to \alpha B \cdot \gamma}{\longrightarrow} k} \quad \text{(Complete)}$$

Jedesmal, wenn neue Einträge zu $F(i, j, B \to \beta \cdot)$ kommen, wird die Vervollständigungsregel angewendet, bis sie keine neuen Einträge mehr erzeugt werden.

Die Feldeinträge $\vec{t} \in F(i, j, A \to \alpha \cdot \beta)$ werden in Prolog durch Fakten der Form

`kante(I, J, A, Alpha, Beta, Baumliste).`

dargestellt. Sie stehen für i.a. partielle Analysen:

```
              A
            /   \
        Alpha   Beta
       /     \
      / Baum- \
     /  liste  \
   ai ....... aj
```

Der Parser berechnet die Tabelle durch Anwenden der obigen Regeln (mit entsprechender Verkettung der Baumfolgen). Man durchläuft die Eingabe w von links nach rechts und berechnet alle Analysen der kurzen Präfixe von w vor denen der längeren:

⟨*Parser/earley.doerre.pl*⟩≡
```
:- op(1100,xfx,'-->').
:- ensure_loaded('Parser/main.pl').
```
⟨*Earley-Dörre, Parser*⟩
⟨*Earley-Dörre, Erkenner*⟩

Durch `parse/2` wird zuerst die Tabelle einer möglichen früheren Analyse (oder Erkennung) gelöscht und der Abschluß aller Kanten $0 \overset{S \to \cdot \alpha}{\longrightarrow} 0$ mit Startsymbol S und leerer Analyse berechnet:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩≡
```
:- dynamic kante/6.

parse(_,_) :-
        retractall(kante(_I,_J,_A,_alpha,_beta)),
        retractall(kante(_I,_J,_A,_alpha,_beta,_trees)),
        start_symbol(S),
        (S --> Alpha),
        closure(0,0,S,[],Alpha,[]),
        fail.
```

Das bedeutet, man „erwartet“ an der Position 0 einen Ausdruck der Kategorie S. Die Abschlußbildung ignoriert Lesarten, die nicht zu dieser Erwartung passen.

Dann wird die Analyse der Eingabe, einer Wortliste, gestartet,

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
parse([],_) :-
        !, nl,write('Fehler: Leere Eingabe.'),fail.
parse(Wortliste,Baum) :-
        parse(0,Wortliste,Baum).
```

und die Wörter nacheinander verarbeitet (s.u.):

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
parse(I,[Wort|Rest],Baum) :-
        J is I + 1,
        shift(I,J,Wort),
        parse(J,Rest,Baum).
```

Wenn das letzte Wort verarbeitet ist, ist die Tabelle berechnet; man prüft, ob das Feld der gesamten Eingabe $w(0,n)$ einen Baum enthält, dessen Wurzel ein Startsymbol ist:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
parse(I,[],Baum) :-
        start_symbol(S),
        kante(0,I,S,_Alpha,[],Baeume),
        Baum = [S|Baeume].
```

Um das Wort a and der Stelle i der Eingabe zu verarbeiten, legt man für jeden seiner Lexikoneinträge A die Folge $[a]$ mit dem (einknotigen) Baum a in das Feld $F(i, i+1, A \to a\cdot)$ der Tabelle und schließt sie ab:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
shift(I,J,Wort) :-
        lexword(I,Wort,Kategorie),
        closure(I,J,Kategorie,[Wort],[],[[Wort]]),
        fail.
shift(_,_,_).
```

Beim Abschließen der Kanten unter Verkettung, `closure/6`, werden die Baumfolgen der entsprechenden Felder verkettet. Man ruft `closure(I+,J+,A+,Alpha+,Beta+,Baumliste+)` auf, wenn eine vermutlich neue `Baumliste` in das Feld $F(i, j, A \to \alpha \cdot \beta)$ eingetragen werden soll. Dann sind verschiedene Dinge zu tun:

(i) Falls es keine offenen Kategorien mehr gibt und `[Kopf|Baumliste]` eine *neue* Analyse von $w(i, j)$ ist, muß sie mit Analysen von Teilen $w(k, i)$ kombiniert werden:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
closure(I,J,Kopf,Geschlossen,[],Baumliste) :-
        neue_kante(I,J,Kopf,Geschlossen,[],Baumliste),
        complete(I,J,Kopf,Baumliste).
```

(ii) Falls die erste offene Kategorie, `B`, getilgt werden kann, ist der Baum `[B,[]]` mit Wurzel `B` und leerer Teilbaumfolge `[]` eine mögliche Analyse von `B`, mit der die partielle Analyse `Baumliste` fortgesetzt werden kann. Für die fortgesetzte Analyse ist wieder `closure` aufzurufen:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
closure(I,J,Kopf,Geschlossen,[B|Bs],Baumliste) :-
        deletable(B),
        append(Geschlossen,[B],Geschlossen1),
        append(Baumliste,[[B,[]]],Baumliste1),
        closure(I,J,Kopf,Geschlossen1,Bs,Baumliste1).
```

Hier ergänzen wir zwei Fälle, mit denen disjunktive und optionale Kategorien auf der rechten Regelseite behandelt werden.

⟨*Earley-Dörre (mit Löschbäumen)*⟩≡
```
closure(I,J,Kopf,Geschlossen,[{B}|Bs],Baumliste) :-
        !,closure(I,J,Kopf,Geschlossen,[B|Bs],Baumliste).

closure(I,J,Kopf,Geschlossen,[(B;C)|Bs],Baumliste) :-
        !,closure(I,J,Kopf,Geschlossen,[B|Bs],Baumliste),
        closure(I,J,Kopf,Geschlossen,[C|Bs],Baumliste).
```

(iii) Falls die partielle Analyse `Baumliste` neu zu $F(i, j, A \to \alpha \cdot B\beta)$ getan wird, ist die erste offene Kategorie, `B`, eine (evtl. neue) Erwartung für die Eingabe ab j:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
closure(I,J,Kopf,Geschlossen,[B|Bs],Baumliste) :-
        neue_kante(I,J,Kopf,Geschlossen,[B|Bs],Baumliste),
        expand(J,B).
```

In Fall (i) und (iii) *testet* man mit `neue_kante/6` (und dem `!,fail` darin), ob der Eintrag schon vorhanden ist oder durch Belegung von freien Variablen aus einem vorhandenen Eintrag hervorgeht (Vgl. `add/2` später. Zu `known_instance/2` siehe Anhang 9). Wenn ja, ist die Kante nicht neu:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
neue_kante(I,J,Kopf,Geschlossen,Offen,Baumliste) :-
        known_instance(kante(I,J,Kopf,Geschlossen,Offen,Baumliste)),
        !,fail.
```

Wenn nicht, macht man den Eintrag und fährt fort:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡
```
neue_kante(I,J,Kopf,Geschlossen,Offen,Baumliste) :-
        (ein(verbose)
        -> nl,nl,write_kante(I,J,Kopf,Geschlossen,Offen,Baumliste)
        ; true),
        assertz(kante(I,J,Kopf,Geschlossen,Offen,Baumliste)).
```

Bem.: Die einfachere Regel

⟨*Earley-Dörre, Parser (mit Löschbäumen) - Fehler*⟩≡

```
neue_kante(I,J,Kopf,Geschlossen,Offen,Baumliste) :-
        kante(I,J,Kopf,Geschlossen,Offen,Baumliste),
        !,fail.
```

funktioniert nur, wenn keine freien Variablen in den Kanten auftreten – sonst können diese beim Vergleich belegt werden, sodaß neue Regeln nicht als neu erkannt werden. Wenn man stattdessen alle vorhandenen Einträge durchsucht und testet, ob die neue Kante eine Instanz ist, wird der Paser sehr langsam:

⟨*Earley-Dörre, Parser (mit Löschbäumen) - Schlecht*⟩≡

```
neue_kante(I,J,Kopf,Geschlossen,Offen,Baumliste) :-
        kante(I,J,KopfX,GeschlossenX,OffenX,BaumlisteX),
        matches((Kopf,Geschlossen,Offen,Baumliste),
                (KopfX,GeschlossenX,OffenX,BaumlisteX)),
        !,fail.
```

Im Fall (i) wurde eine neue Analyse s = `[B|Teilbaeume]` von $w(i,j)$ gefunden. Alle Analysen $\vec{t} \in F(k,i,A \rightarrow \alpha \cdot B\beta)$ von Teilen $w(k,i)$, die „auf einen B-Ausdruck warten", müssen um die neue Analyse s fortgesetzt werden. Dazu wird s an $\vec{t}$ angehängt und in das entsprechende Feld $F(k,j,A \rightarrow \alpha B \cdot \beta)$ eingetragen. Anschließend wird erneut closure aufgerufen:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡

```
complete(I,J,B,Teilbaeume) :-
        kante(K,I,A,Alpha,[B|Beta],Baumliste),
        append(Alpha,[B],AlphaB),
        append(Baumliste,[[B|Teilbaeume]],BaumlisteB),
        closure(K,J,A,AlphaB,Beta,BaumlisteB),
        fail.
complete(_,_,_,_).
```

Wenn ab j ein Ausdruck der Kategorie A erwartet wird, muß der Abschluß der neuen Kanten $j \xrightarrow{A \rightarrow \cdot\alpha} j$ gebildet werden:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡

```
expand(J,A) :-
        (A --> Alpha),
        closure(J,J,A,[],Alpha,[]),
        fail.
expand(_,_).
```

Zur Information kann man sich während der Analyse die gefundenen Einträge anzeigen lassen:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩+≡

```
write_kante(I,J,Kopf,Offen,Geschlossen,Baumliste) :-
        tab(3),write(I),write(','),
        write(J),write(': ('),
        write(Kopf),write('-->'),
        write(Offen),write('.'),
        write(Geschlossen),write(') '),
        nl,tab(8),write(Baumliste).
```

Dieser Parser erwartet allerdings die Grammatikregeln in einer leicht vom DCG-Format abweichenden Form:

(i) rechte Regelseiten sollten *Listen* von Kategorien sein, keine Tupel,

(ii) löschbare Kategorien sollten durch ein Prädikat `del/1` angegeben werden,

(iii) Startkategorien sollen durch ein Prädikat `start_symbol/1` angegeben werden.

Aufgabe 5.1 Man schreibe ein Prolog-Programm, das Grammatiken im Pseudo-DCG-Format in das von diesem Earley-Parser erwartete Format umrechnet. Dazu muß man

(i) Prolog-Listen `[A,B,C,D,E]` usw. in Prolog-Tupel `(A,(B,(C,(D,E))))` umrechnen,

(ii) ein Prädikat `del(Kategorie?)` schreiben, das die löschbaren Kategorien der Grammatik bestimmt, d.h. diejenigen Kategorien A, zu denen es eine Regel $A \to \alpha$ gibt, wo α leer ist oder nur aus löschbaren Kategorien besteht.

Die Startsymbole sollen schon in der DCG festgelegt werden.

Beispiel 5.3 Hier ist die entsprechende Grammatik zur DCG aus Beispiel 5.1:

$\langle$*Grammatiken/cfg.1.grm.ed*$\rangle\equiv$

```
:- op(1100,xfx,'-->').

start_symbol(s).
start_symbol(np).

s --> [np, vp].

np --> [det, ap, n].
np --> [np, conj, np].


vp --> [v].
vp --> [v, np].
vp --> [v, np, av].

av --> [adv].
av --> [pp].
% av --> []

pp --> [p, np].

ap --> [a].
ap --> [grad, ap].
ap --> [ap, conj, ap].

% a --> [].
% grad --> [].

deletable(av).
deletable(a).
deletable(grad).
deletable(ap).
```

Bei dieser Grammatik erhält man für den Satz `Der Hamster pfeift.` drei Analysen, was von der Löschbarkeit von Adjektiven, Gradpartikeln und Adjektivphrasen verursacht wird:

```
18 ?- parse.
Beende den Eingabesatz mit <Satzzeichen><Return>
|: Der Hamster pfeift.

[ [s, [np, [det, ['Der']], [ap, []], [n, ['Hamster']]], [vp, [v, [pfeift]]]],
  [s, [np, [det, ['Der']], [ap, [a, []]], [n, ['Hamster']]], [vp, [v, [pfeift]]]],
  [s, [np, [det, ['Der']], [ap, [grad, []], [ap, []]], [n, ['Hamster']]], [vp, [v, [pfeift]]]] ]

Yes
19 ?-
```

Aufgabe 5.2 Man schreibe ein Prädikat `showASCIItrees(Trees+)`, das eine Liste von Bäumen in der hier verwendeten Listendarstellung so ausgibt, daß

(i) die Analysen mit einer Nummer versehen werden, und

(ii) die direkten Teilbäume untereinander mit 3 Leerzeichen eingerückt geschrieben werden.

Der erste Baum im obigen Beispiel soll also etwa in der folgenden Form erscheinen:

```
1. s
      np
         det
            Der
         ap
            []
         n
            Hamster
      vp
         v
            pfeift
```

Dieser Earley-Parser ist nur für Grammatiken ohne Lösch- und Kettenregeln geeignet: er kann sonst für manche Eingaben in eine Endlosschleife geraten. Das liegt an den löschbaren Kategorien und den Kettenregeln, d.h. Regeln der Form $A \rightarrow B$ mit nur einer Kategorie auf der rechten Seite. Diese führen dazu, daß es unendlich viele Analysen einer Eingabe geben kann.

Um das auszuschließen, werden wir die Grammatik umformen und die Lösch- und Kettenregeln gesondert behandeln müssen.

Aufgabe 5.3 Man ändere durch Einbauen einer Option `omitEmptyTrees` die Ausgabe des Parsers so, daß Löschungen in der Analyse gar nicht mehr dargestellt werden. Dazu muß man beim Aufbau der Bäume die Zweige, die gelöscht werden, weglassen.

Wenn man dann vorher die Option `einschalten(omitEmptyTrees).` setzt, fallen die drei Analysen aus Beispiel 5.3 zu einer einzigen zusammen:

```
[ [s, [np, [det, ['Der']], [n, ['Hamster']]], [vp, [v, [pfeift]]]] ]
```

Die Version mit dieser Option wird als ⟨*Earley-Dörre, Parser*⟩ in die Datei `Parser/earley.doerre.pl` aufgenommen.

5.3.2 Earley-Erkenner ohne Vorberechnungen

Diese Probleme der Terminierung treten nicht auf, wenn man nur einen Erkenner hat: dann sind die Einträge in den Feldern keine Analysen, sondern Kategoriensymbole, (von denen es nur endlich viele gibt). Wir lassen einfach das Argument der `Baumliste` im obigen Parser weg:

⟨*Earley-Dörre, Erkenner*⟩≡

```
:- dynamic kante/5.

erkenne(_,_) :-
        retractall(kante(_I,_J,_A,_alpha,_beta)),
        retractall(kante(_I,_J,_A,_alpha,_beta,_trees)),
        start_symbol(S),
        (S --> Alpha),
        closure(0,0,S,[],Alpha),
        fail.
erkenne(Wortfolge,Kategorie) :-
        erkenne(0,Wortfolge,Kategorie).

erkenne(I,[Wort|Rest],Kategorie) :-
        J is I + 1,
        shift_e(I,J,Wort),
        erkenne(J,Rest,Kategorie).
erkenne(I,[],Kategorie) :-
        start_symbol(S),
        kante(0,I,S,_Alpha,[]),
        Kategorie = S.

shift_e(I,J,Wort) :-
        lexword(I,Wort,Kategorie),
        closure(I,J,Kategorie,[Wort],[]),
        fail.
shift_e(_,_,_).

closure(I,J,Kopf,Geschlossen,[]) :-
        neue_kante(I,J,Kopf,Geschlossen,[]),
        complete(I,J,Kopf).
closure(I,J,Kopf,Geschlossen,[B|Bs]) :-
        deletable(B),
        append(Geschlossen,[B],Geschlossen1),
        closure(I,J,Kopf,Geschlossen1,Bs).
closure(I,J,Kopf,Geschlossen,[{B}|Bs]) :-
        !,closure(I,J,Kopf,Geschlossen,[B|Bs]).
closure(I,J,Kopf,Geschlossen,[(B;C)|Bs]) :-
        !,closure(I,J,Kopf,Geschlossen,[B|Bs]),
        closure(I,J,Kopf,Geschlossen,[C|Bs]).
closure(I,J,Kopf,Geschlossen,[B|Bs]) :-
        neue_kante(I,J,Kopf,Geschlossen,[B|Bs]),
        expand_e(J,B).

neue_kante(I,J,Kopf,Geschlossen,Offen) :-
        known_instance(kante(I,J,Kopf,Geschlossen,Offen)),
        !,fail.
```

```
neue_kante(I,J,Kopf,Geschlossen,Offen) :-
        assertz(kante(I,J,Kopf,Geschlossen,Offen)).

complete(I,J,B) :-
        kante(K,I,A,Alpha,[B|Beta]),
        append(Alpha,[B],AlphaB),
        closure(K,J,A,AlphaB,Beta),
        fail.
complete(_,_,_).

expand_e(J,A) :-
        (A --> Alpha),
        closure(J,J,A,[],Alpha),
        fail.
expand_e(_,_).
```

Die Prädikate `shift/3` und `expand/2` wurden umbenannt; die anderen haben eine geringere Stelligkeit als die entsprechenden Prädikate des Parsers.

Beispiel 5.4 (Forts.) Der Erkenner findet die Kategorien der eingegebenen Wortfolge:

```
10 ?- erkenne.
Beende den Eingabesatz mit <Satzzeichen><Return>
|: Der Hamster pfeift.

[s]

Yes
```

Die aufgebaute Tabelle für diesen Beispielsatz ist:

```
11 ?- listing(kante/5).

kante(0, 0, s, [], [np, vp]).
kante(0, 0, np, [], [det, ap, n]).
kante(0, 0, np, [], [np, conj, np]).
kante(0, 1, det, ['Der'], []).
kante(0, 1, np, [det, ap], [n]).
kante(0, 1, np, [det], [ap, n]).
kante(1, 1, ap, [a], []).
kante(1, 1, ap, [], [a]).
kante(1, 1, ap, [grad, ap], []).
kante(1, 1, ap, [grad], [ap]).
kante(1, 1, ap, [], [grad, ap]).
kante(1, 1, ap, [ap], [conj, ap]).
kante(1, 1, ap, [], [ap, conj, ap]).
kante(1, 2, n, ['Hamster'], []).
kante(0, 2, np, [det, ap, n], []).
kante(0, 2, s, [np], [vp]).
kante(2, 2, vp, [], [v]).
kante(2, 2, vp, [], [v, np]).
kante(2, 2, vp, [], [v, np, av]).
```

```
kante(0, 2, np, [np], [conj, np]).
kante(2, 3, v, [pfeift], []).
kante(2, 3, vp, [v], []).
kante(0, 3, s, [np, vp], []).
kante(2, 3, vp, [v], [np]).
kante(3, 3, np, [], [det, ap, n]).
kante(3, 3, np, [], [np, conj, np]).
kante(2, 3, vp, [v], [np, av]).

Yes
```

Auch mit den Lösch- und Kettenregeln bei Adjektivphrasen hat der Erkenner keine Probleme:

```
12 ?- erkenne .
Beende den Eingabesatz mit <Satzzeichen><Return>
|: der arme Hamster.

[np]

Yes
```

Dagegen terminiert der Parser für diese Eingabe nicht: da `grad` löschbar ist, erhält man über die Regel `ap --> [grad, ap]` Syntaxbäume mit beliebiger Tiefe, die der Parser alle in die Tabelle eintragen muß!

5.3.3 Kanonische Analysebäume und Vorberechnungen

Die Analyse einer Eingabe w kann nur terminieren, wenn die Anzahl der syntaktischen Strukturen von w bezüglich der Grammatik G endlich ist. Das ist aber nicht bei allen kontextfreien Grammatiken der Fall.

Wir fassen daher die Analysen in endlich viele Äquivalenzklassen zusammen und berechnen beim Parsen von w nur die Äquivalenzklassen der Analysen von w. Die *Äquivalenz von G-Analysen* definieren wir wie folgt, wobei A eine Kategorie, B ein Wort oder eine Kategorie und w, v, u Wörter sind:[8]

(i) Alle Analysen zu $A \to^* \epsilon$ sind zueinander äquivalent; die kürzeste darunter ist die *kanonische* Analyse zu $A \to^* \epsilon$.

(ii) Alle Analysen zu $A \to^* B$ sind zueinander äquivalent; die kürzeste darunter ist die *kanonische* Analyse zu $A \to^* B$.

(iii) Zwei Analysen zu $A \to^* w$ mit $|w| > 1$ sind äquivalent, wenn es eine Grammatikregel $B \to B_1 \cdots B_k$ von G und eine Zerlegung $w = w_1 \cdots w_k$ gibt, so daß

  (a) beide Analysen mit $A \to^* B \to B_1 \cdots B_k$ beginnen und Teilanalysen zu $B_i \to^* w_i$, $i = 1, \ldots, k$ haben,

  (b) für jedes i die Teilanalysen zu $B_i \to^* w_i$ zueinander äquivalent sind,

  (c) alle $w_1, \ldots, w_k$ von w verschieden sind;

  die *kanonischen* unter diesen Analysen sind diejenigen, bei denen die Teilanalyse zu $A \to^* B$ und die zu den $B_i \to^* w_i$ kanonische Analysen sind.

[8] Da wir das Lexikon abtrennen, sind hier allerdings mit Wörtern die *lexikalischen Kategorien* gemeint!

Intuitiv gesagt sind die kanonischen Analysen diejenigen, bei denen der Aufwand für Löschungen und Kettenableitungen minimal ist. Beachte, daß durch Zusammensetzen kanonischer Analysen im allgemeinen eine nicht-kanonische Analyse entsteht.

Um ein Analyseverfahren zu erhalten, das nur kanonische Bäume aufbaut, werden wir

(i) Löschregeln $A \to \epsilon$ und Kettenregeln $A \to B$ (mit Kategorie B) aus der Grammatik entfernen,

(ii) spezielle Prädikate einführen, die den Bedingungen $A \to^* \epsilon$ und $A \to^* B$ entsprechen.

Außerdem werden wir weitere Beziehungen zwischen Kategorien vorberechnen und als *Hilfstabellen* abspeichen. Diese Hilfstabellen sind meistens *extensional* gegebene Prolog-Prädikate: sie bestehen nur aus Fakten. (Für größere Grammatiken wird allerdings auch mit einer Darstellung durch Prolog-Regeln experimentiert.) Dazu muß die transitive Hülle zweistelliger Relationen berechnet werden, was im Anhang 9.1 ausgeführt wird.

⟨*Parser/cfg-vorberechnungen.pl*⟩≡
```
:- op(1100,xfx,'-->').
:- ensure_loaded('Parser/transclos.pl').

add(H :- B) :-
        clause(H,B),!.
add(H :- B) :-
        !,(ein(verbose) -> write('adding rule: '),write(H :- B),nl ; true),
        assertz(H :- B).
```

Terme, die Instanzen schon vorhandener Einträge sind, sollen nicht gespeichert werden. (Das ist natürlich nur korrekt, solange keine Einträge aus der Datenbank entfernt werden!). Ein Term ist eine Instanz eines vorhandenen Eintrags, wenn der Grundterm, der durch Einsetzen neuer Konstanten für die Variablen des Terms entsteht, beweisbar ist:

⟨*Parser/cfg-vorberechnungen.pl*⟩+≡
```
add(Fact) :-
        \+ \+ (numbervars(Fact,const,0,NumVars), Fact), !.
add(Fact) :-
        (ein(verbose) -> write('adding fact: '),write(Fact),nl ; true),
        assertz(Fact).
```

⟨*Lexikalische Kategorien*⟩
⟨*Löschbare Kategorien*⟩
⟨*Kettenbeziehung*⟩
⟨*Vorberechnung der First-Relation*⟩
⟨*Vorberechnung brauchbarer Regeln*⟩

5.3.4 Vorberechnung der lexikalischen Kategorien

Es ist sinnvoll, bei den Vorberechnungen auch eine Tabelle `lexCat/1` der lexikalischen Kategorien der Grammatik anzulegen. Damit kann der Parser, wenn man die Option `pruefeLexWord` einschaltet, unabhängig vom (vielleicht transformierten) Regelformat überprüfen, ob alle von der lexikalischen Analyse gefundenen Wortarten auch der Grammatik bekannt sind:

⟨*Lexikalische Kategorien*⟩≡
```
% ------- Berechnung der lexikalischen Kategorien -----
:- dynamic lexCat/1.
```

```
compute_lexCat :-
        retractall(lexCat(_)),
        setof(C,A^B^Bs^Cs^X^((A --> B),
                         tupleToList(B,Bs),
                         collect_lexCats(Bs,Cs),
                         member(C,Cs)),
              LexCatList),
        add_LexCats(LexCatList).

collect_lexCats([{C}|Cs],Cats) :-
        !,collect_lexCats([C|Cs],Cats).
collect_lexCats([(C;D)|Cs],Cats) :-
        !,collect_lexCats([C,D|Cs],Cats).
collect_lexCats([(C,D)|Cs],Cats) :-
        !,collect_lexCats([C,D|Cs],Cats).
collect_lexCats([C|Cs],Cats) :-
        !,collect_lexCats(Cs,CsCats),
        (has_no_syntaxrules(C) -> Cats = [C|CsCats] ; Cats = CsCats).
collect_lexCats([],[]).

has_no_syntaxrules(C) :-
        ((C --> X) -> X = [] ; true).

add_LexCats(B;C) :- !,add_LexCats(B),add_LexCats(C).
add_LexCats({C}) :- !,add_LexCats(C).
add_LexCats([]) :- !.
add_LexCats([C|Cs]) :- !,add_LexCats(C), add_LexCats(Cs).
add_LexCats(C) :- add(lexCat(C)).
```

Bemerkung 5.5 Das Prädikat `has_no_syntaxrule(C)` soll die Kategorien herausfiltern, die 'eigentlich' keine Syntaxregeln haben. Darunter werden hier die Kategorien verstanden, deren einzige Regeln Löschregeln sind, also Regeln `C --> X` mit `X = []` . Dazukommen sollten vielleicht die Kategorien `C`, zu denen es nur Kettenregeln `C --> L` gibt, wo `L` eine lexikalische Kategorie ist. Mit ihnen könnte man Wortarten aufspalten (z.Zt. `a(Art,Form) --> ordinalzahl` und `a(Art,Form) --> kardinalzahl`). Man will hier raten, welche Kategorien in Lexikoneinträgen auftreten können; das kann im Prinzip nicht gehen, aber eine vernünftige Konvention wäre schon, daß man durch Grammatikregeln Wortarten aufspalten darf; dann sollte sowohl die Wortart wie ihre Unterarten in den Lexikoneinträgen auftreten dürfen.

5.3.5 Vorberechnung der löschbaren Kategorien

Aus den Pseudo-DCG-Regeln wird eine Tabelle `del/1` gebildet, die die löschbaren Kategorien enthält.

⟨*Löschbare Kategorien*⟩≡

```
% ------- Berechnung der löschbaren Kategorien A -->* [] -----
:- dynamic del/1, aNewDeletable/0.

compute_deletable_categories :-
        retractall(del(_)),
        retractall(aNewDeletable),
        compute_del_basecase,
        compute_del_iteration.
```

Zuerst werden die Kategorien gesammelt, für die es eine Löschregel gibt. Damit die Tabelle übersichtlich bleibt, werten wir die Bedingungen in den Löschregeln nicht aus, sondern schreiben *bedingte* Löschkategorien in die Tabelle:

⟨*Löschbare Kategorien*⟩+≡
```
compute_del_basecase :-
    clause(A --> [],Cond),
    add_del(A,Cond),
    fail.
compute_del_basecase.

add_del(A,Cond) :-
    clause(del(A),Cond),!.
add_del(A,Cond) :-
    assertz((del(A) :- Cond)),
    (aNewDeletable -> true ; assert(aNewDeletable)).
```

Hinzu kommen alle Kategorien `A`, zu denen es eine Regel `(A --> Alpha)` gibt, deren rechte Seite löschbar ist:

⟨*Löschbare Kategorien*⟩+≡
```
compute_del_iteration :-
    repeat,
    comp_del_iteration,
    not(aNewDeletable),!.
compute_del_iteration.

comp_del_iteration :-
    retract(aNewDeletable),
    clause(A --> Body,Cond),
    deletable(Body),
    simplify(Cond,NewCond),
    add_del(A,NewCond),
    fail.
comp_del_iteration.
```

Da mit `deletable(Body)` normalerweise die Variablen belegt werden, erhalten wir hier meist unbedingte Löschkategorien; nur wenn `Body` ohne Belegung der Variablen als löschbar nachweisbar ist, erhält man bedingte Löschkategorien.

Der Löschbarkeitstest folgt dem Aufbau der Kategorienausdrücke einer Pseudo-DCG-Regel:

⟨*Löschbare Kategorien*⟩+≡
```
deletable((A,B)) :-
    !,deletable(A),deletable(B).
deletable((A;B)) :-
    !,(deletable(A);deletable(B)).
deletable({_C}) :- !, true.
deletable([_A|_B]) :- !, fail.
deletable([]) :- !, true.
deletable(A) :- !, del(A).
```

Da in `(A,B)` (bzw. `(A;B)`) zumindest `B` wieder eine Konjunktion (bzw. Disjunktion) sein kann, da `A,B,C` dasselbe ist wie `(A,(B,C))`, muß man rekursiv die Teilausdrücke bearbeiten.

5.3.6 Vorberechnung der Kettenbeziehung

Hier wird einfachheitshalber vorausgesetzt, daß die rechten Regelseiten einfache Konjunktionen atomarer Prolog-Klauseln sind. Neben der Grammatik muß auch die Tabelle `del/1` der löschbaren Kategorien geladen sein.

⟨*Kettenbeziehung*⟩≡

```
% ------- Berechnung der Kettenbeziehung A -->* B --------
:- dynamic chain/2.

compute_chain_relation :-
        retractall(chain(_,_)),
        ensure_loaded('Parser/transclos.pl'),
        comp_chain,
        compute_chain.
```

Zuerst werden die Paare (A, B) gesammelt, für die es eine Regel $A \to \alpha B \beta$ mit löschbaren α und β gibt:

⟨*Kettenbeziehung*⟩+≡

```
comp_chain :-
    clause(A --> Body,Cond),
    tupleToList(Body,BodyL),
    append(Beta,[B|Gamma],BodyL),    % ineffizient
    deletables(Beta),
    deletables(Gamma),
    simplify(Cond,NewConds),
    addChains(A,B,NewConds),
    fail.
comp_chain.

deletables([C|Cs]) :-
        deletable(C), deletables(Cs).
deletables([]).
```

Falls die Grammatik direkte Kettenregeln hat (d.h. nicht durch Löschungen von Konstituenten entstehende), so werden daraus *bedingte* Tupel von `chain/2`. Sonst werden die Löschbarkeitstests normalerweise die Variablen belegen und durch `simplify` wird die Bedingung ausgewertet, sodaß man meist eine Reihe von unbedingten Tupeln zu `chain/2` hinzufügt.

In `addChains(A,B,Cond)` wird `B` nach seiner Struktur untersucht und die erreichbaren Teile in die Kettenbeziehung übernommen:

⟨*Kettenbeziehung*⟩+≡

```
addChains(A,(C,D),Cond) :-
        !, (deletable(D) -> addChains(A,C,Cond) ; true),
        (deletable(C) -> addChains(A,D,Cond) ; true).
addChains(A,(C;D),Cond) :-
        !, addChains(A,C,Cond), addChains(A,D,Cond).
addChains(A,{B},Cond) :-
        !, addChains(A,B,Cond).
addChains(A,[W|Ws],Cond) :-
        !,add(chain(A,[W|Ws],Cond)).
addChains(A,[],Cond) :-
        !. % add(chain(A,[])).
```

```
addChains(A,B,Cond) :-
        (A == B -> true ; add(chain(A,B) :- Cond)).
```

Diese Grundrelation `chain/2` enthält nur Paare `(A,B)`, bei denen beide Komponenten verschiedene Prolog-Terme sind, die nicht von der Form `(C;D)`,`(C,D)`,`[C|Cs]`,`[]`,`{C}` sind. Da auch die linken Regelseiten dieser Einschränkung genügen, verliert man in der folgenden Bildung der irreflexiven transitiven Hülle keine Tupel.

Nun wird mit `itc/3` die irreflexive transitive Hülle der gerade erzeugten Tabelle `chain/2` gebildet. Aus der Grundrelation `chain/2` werden die Elemente der Form `chain(X,X)` entfernt und dann die neu gefundenen hinzugefügt:

⟨*Kettenbeziehung*⟩+≡

```
compute_chain :-
    findall((X,Y),itc(chain,X,Y),Chains),
    % retractall(chain(_,_)),
    retractall(chain(X,X)),
    add_chain(Chains).

add_chain([(X,Y)|T]) :-
    add(chain(X,Y)),
    add_chain(T).
add_chain([]).
```

Bem: Die vorhandenen Klauseln der Grundrelation werden nicht entfernt, damit die durch Regeln kompakter beschriebenen Teile nicht durch große Listen von Tupeln ersetzt werden.

Die Relation `chain/2` enthält keine Paare mit gleichen Komponenten oder optionalen oder disjunktiven Kategorien; diese werden später durch `chain(A,B);instance(B,A)` berücksichtigt.

Die (irreflexive) transitiven Hülle einer Relation wird mit dem Algorithmus von Warshall berechnet. Die Version aus Abschnitt 9.1.1 nach R.O'Keefe erzeugt *nur für eine Grundrelation aus variablenfreien Termen die korrekte transitive Hülle.*

Für Merkmalgrammatiken, in deren Grammatikregeln nicht alle Variablen durch Bereichsprädikate beschränkt sind, sind für die transitive Hülle zwei weitere Möglichkeiten implementiert (vgl. Anhang Hilfsprogramme):

(i) eine Variante des Warshall-Algorithmus, die Terme mit Variablen so behandelt, als seien die Variablen ∀-quantifiziert,

(ii) das Programm `tc/4`, das die transitive Hülle anders berechnet.

⟨*Kettenbeziehung2*⟩≡

```
compute_chain :-
    findall((X,Y),chain(X,Y),BasicChain),
    tc(BasicChain,link,matches,Chains),
    retractall(chain(_,_)),
    add_chain(Chains).
```

Beide Möglichkeiten benutzen Hilfsprädikate `link/2` und `matches/2`, mit denen die Unifikation von freien Variablen umgangen wird. Diese Programme sind nicht gründlich getestet und erfordern bei größeren Grammatiken eine zu große Rekursionstiefe. Man *muß* dann die Grundrelation durch die Auswahl `basefirst(restrict)` vereinfachen und mit `predictor(uniform)` den uniformen statischen Predictor benutzen (s.u.).

5.3.7 Vorberechnung der First-Relation

Eine für verschiedene Parser nützliche Relation zwischen Kategorien ist

$$\mathit{first} := \{(B, C) \mid \exists\beta\ C \to^* B\beta\}.$$

Wir berechnen sie aus der Grammatik und können dann im Parser in der Tabelle `first/2` nachsehen, ob ein B-Ausdruck am Anfang von C-Ausdrücken stehen kann.

Zuerst wird das Ergebnis einer vorherigen Berechnung von `first/2` gelöscht, dann bestimmt man die aus einer einzigen Regel ablesbaren Tupel der Relation, und schließlich bildet man davon die transitive Hülle:

⟨*Vorberechnung der First-Relation*⟩≡

```
% -------Berechnung der Relation  Es gibt beta: A -->* B beta --
:- dynamic first/2.

compute_first_relation :-
        retractall(first(_,_)),
        comp_first_basecase,
        compute_first.
```

Die Grundrelation, deren transitive Hülle gebildet werden muß, ist im wesentlichen

$$\{(B, A) \mid A \to \alpha B\beta,\ \alpha \to^* \epsilon\}.$$

Da wir Pseudo-DCG-Regeln behandeln wollen, müssen die rechten Regelseiten zerlegt werden, bis man auf eine `[Wortfolge]` oder einen einfachen Prolog-Term (eine grammatische Kategorie) stößt:

⟨*Vorberechnung der First-Relation*⟩+≡

```
comp_first_basecase :-
        (ein(verbose) -> write('Base relation:'),nl ; true),
        (basefirst(rules)
        -> clause(A --> Alpha, Cond), collect_basefirst(A,Alpha,Cond)
        ; (A --> Alpha), collect_basefirst(A,Alpha)),
        fail.
comp_first_basecase :-
        pruefe_restrictor.
```

(Die Auswahl `basefirst(rules)` scheint nicht besonders nützlich zu sein.) Mit `collect_basefirst(A,Alpha)` werden die Tupel gesammelt, die im ersten Schritt in die First-Relation aufzunehmen sind. Da keine Kategorien der Form `(B;C)` und `{B}` als Kategorien erkannter Teilphrasen auftreten[9], kann man sie aus der Relation herauslassen:

⟨*Vorberechnung der First-Relation*⟩+≡

```
collect_basefirst(A,(B,C)) :-
        !,collect_basefirst(A,B),
        (deletable(B) -> collect_basefirst(A,C) ; true).
collect_basefirst(A,(B;C)) :-
        !,
        collect_basefirst(A,B),
        collect_basefirst(A,C).
collect_basefirst(A,{B}) :-
        !,
        collect_basefirst(A,B).
collect_basefirst(_A,[]) :-
        !.
```

[9] Man könnte allerdings disjunktive Kategorien in Lexikoneinträgen brauchen!

Da im allgemeinen die nachfolgende Berechnung der transitiven Hülle der hier gebildeten Grundrelation `first/2` aufwendig ist, lassen wir eine Vereinfachung der Grundrelation zu: statt Tupel *vollständiger* Kategorienausdrücke werden *vereinfachte* oder *eingeschränkte* Kategorienausdrücke gesammelt:

⟨*Vorberechnung der First-Relation*⟩+≡
```
collect_basefirst(A,[B|Rest]) :-
        !,(basefirst(restrict)
          -> restrict_all([A,B|Rest],[AR,BR|RestR]), add(first([BR|RestR],AR))
          ; add(first([B|Rest],A))).
collect_basefirst(A,B) :-
        !,(A == B -> true
          ; (basefirst(restrict)
            -> restrict_all([A,B],[AR,BR]), add(first(BR,AR))
            ; add(first(B,A)))).

restrict_all([T|Ts],[TR|TsR]) :-
        restrict(T,TR),
        restrict_all(Ts,TsR).
restrict_all([],[]).
```

Im Fall `A == B` lassen wir Tupel der Form `(A,A)` weg. Wir setzen –ohne es zu überprüfen– voraus, daß bei der Bildung der transitiven Hülle die Merkmalausdrücke nicht beliebig groß werden. Regeln wie `cat(f(X)) --> cat(X)` werden durch diese Voraussetzung ausgeschlossen. Andernfalls braucht die Vorberechnung nicht zu terminieren.

Die Einschränkung `restrict(Kategorie+,Vereinfachung-)` besteht darin, gewisse Teilausdrücke zu ignorieren. Das wird für jede Kategorie unterschiedlich gemacht, in den Dateien der Grammatik, die die Kategorien beschreiben, z.B. `np.cat.pl`. Um zu prüfen, ob alle in der Grammatik benutzten Kategorien eine Einschränkung durch `restrict/2` haben, definieren wir:

⟨*Vorberechnung der First-Relation*⟩+≡
```
pruefe_restrictor :-
        write(user,'***** Prüfe, ob alle lexCats eine Restriktion haben.\n'),
        lexCat(Cat),
        (restrict(Cat,_) -> true
        ; write(user,'Warnung: Kategorie '), write(user,Cat),
            write(user,' hat keinen Restriktor.\n')),
        fail.
pruefe_restrictor.
```

Bem.: Ebenso für Phrasenkategorien zu machen!

Außerdem sollte dieses Restriktionsprädikat benutzt werden, um die Berechnung der transitiven Hülle der Grundrelation `first/2` zu beschleunigen, was bisher nicht geschieht.

Bemerkung 5.6 Bei kleineren Grammatiken hatte die folgende grobere Einschränkung einen ebensoguten Effekt wie eine genaue Berechnung mit allen Merkmalen. Wir ersetzen einfach alle echten Teilterme durch neue Variable:

⟨*Vorberechnung der First-Relation (unbenutzt)*⟩≡
```
restrict(Term,Vereinfacht) :-
        compound(Term),
        !, functor(Term,F,Arity),
        length(Liste,Arity),
        Vereinfacht =.. [F|Liste].
restrict(Term,Term).
```

Bei größeren Grammatiken ist diese Restriktion aber nicht gut genug, da noch zu viele Regeln vorausgesagt werden. Daher wird z.Zt. mit verschiedenen `restrict`-Fakten für die einzelnen Kategorien experimentiert, wodurch man je nach der Kategorie auf bestimmte Merkmale achten und andere wegprojizieren kann.

Für den Fall `basefirst(rules)` werden zur Darstellung der Grundrelation `first/2` Regeln in die Datenbank geschrieben:

⟨*Vorberechnung der First-Relation*⟩+≡
```
collect_basefirst(A,(B,C),Cond) :-
        !,collect_basefirst(A,B,Cond),
        (deletable(B) -> collect_basefirst(A,C,Cond) ; true).
collect_basefirst(A,(B;C),Cond) :-
        !,
        collect_basefirst(A,B,Cond),
        collect_basefirst(A,C,Cond).
collect_basefirst(A,{B},Cond) :-
        !,
        collect_basefirst(A,B,Cond).
collect_basefirst(_A,[],_Cond) :-
        !.
collect_basefirst(A,[B|Rest],Cond) :-
        !, simplify(Cond,NewCond),
        add((first([B|Rest],A) :- NewCond)).
collect_basefirst(A,B,Cond) :-
        !,(A == B -> true ; simplify(Cond,NewCond), add((first(B,A) :- NewCond))).

simplify((Cond,Conds),Cs) :-
        !,simplify(Conds,NewConds),
        (ground(Cond)
        -> (call(Cond) -> Cs = NewConds ; Cs = fail)
        ; Cs = (Cond,NewConds)).

simplify(Cond,NewCond) :-
        !,(ground(Cond) -> call(Cond),NewCond = true ; NewCond = Cond).
```

Die bisher gesammelten Tupel bilden eine Grundrelation, die noch erweitert werden muß. Die irreflexive transitive Hülle von `first/2`, genannt `itc(first,X,Y)`, wird mit dem Algorithmus von Warshall (in der Implementierung aus dem Anhang) berechnet.[10]

Die Ausgangsrelation `first/2` wird nicht beseitigt, sondern es werden nur die neu gefundenen Tupel ergänzt. (Das hat den Vorteil, daß man die Grundrelation auch durch Regeln angeben kann.) Nur die Tupel mit gleichen Komponenten werden entfernt.

⟨*Vorberechnung der First-Relation*⟩+≡
```
compute_first :-
    (ein(verbose) -> nl,write('Transitive closure of the base relation:') ; true),
    (ein(showProfile)
        -> profile(findall((X,Y),itc(first,X,Y),Firsts),plain,5)
        ; findall((X,Y),itc(first,X,Y),Firsts)),
    add_first(Firsts).
```

[10] Bei `findall((X,Y),itc(first,X,Y),Firsts)` wird in `itc(first,X,Y)` der Algorithmus von Warshall nur einmal aufgerufen, nicht etwa für jedes `(X,Y)`.

Die Elemente der irreflexiven transitive Hülle speichern wir als Tabelle `first/2` in der Prolog-Datenbank ab. (Diese Relation wird in der Literatur auch `leftcorner/2` genannt; dann ist `first/2` der Teil, bei dem die erste Komponente eine lexikalische Kategorie ist.)

⟨*Vorberechnung der First-Relation*⟩+≡
```
add_first([(X,Y)|T]) :-
    add(first(X,Y)), add_first(T).
add_first([]).
```

Elemente der Form (X, X) werden nicht gespeichert, da wir bei der reflexiven transitiven Hülle *alle* solche Elemente brauchen. Das wird später durch `first(X,Y);instance(X,Y)` erreicht, wobei

⟨*Vorberechnung der First-Relation*⟩+≡
```
instance(D,(A,B)) :-
        !,instance(D,A).
instance(D,(A;B)) :-
        !,(instance(D,A);instance(D,B)).
instance(D,{A}) :-
        !, instance(D,A).
instance(D,D).
```

dazu dient, disjunktive und optionale Kategorien wie in impliziten Regeln `(A;B) --> A.`, `(A;B) --> B.`, und `{A} --> A.` zu behandeln.

5.3.8 Vorberechnung brauchbarer Regeln

Die bei einer Erwartungskategorie C und einer Wortkategorie L *brauchbaren Regeln* sind

$$predict(C, L) := \bigcup_{\beta \neq \epsilon, \gamma, \delta} \{A \to \alpha B \beta \mid C \to^* A\gamma,\ \alpha \to^* \epsilon, B \to^* L\delta\}.$$

Sie können für alle (C, L) durch `predict_static(C,L)` vorberechnet werden, wobei man sich auf lexikalische Kategorien L beschränken kann. Zur Parsezeit treten bei der Stelle j der Eingabe nur die (C, L) auf, bei denen C eine Erwartungskategorie für $w(j, n)$ und L die lexikalische Kategorie von $w(j, j + 1)$ ist.

Da bei größeren Grammatiken sehr viele Paare `(C,L)` auftreten, wird diese Tabelle der brauchbaren Regeln ein Vielfaches der Grammatikregeln und zu groß. Mit der Auswahl `auswahl(predictor(uniform))` beschränkt man die Vorberechnung der brauchbaren Regeln darauf, die durch Beseitigen löschbarer Präfixe von rechten Regelseiten entstehenden zusätzlichen Regeln berechnet (s.u. `predict_static/0`).

⟨*Vorberechnung brauchbarer Regeln*⟩≡
```
% --- Bei Erwartung C und Wortkategorie LexCat brauchbare Regeln ---
:- dynamic predicted/6.
:- dynamic predicted/4.

compute_predictable_rules :-
        nl,write('***** Computing statically predictable rules ... (wait) '),nl,nl,
        retractall(predicted(_,_,_,_)),
        retractall(predicted(_,_,_,_,_,_)),
        (ein(showProfile)
        -> profile(comp_static_predictor,plain,5) ; comp_static_predictor).

comp_static_predictor :-
```

```
        (predictor(uniform) -> predict_static
        ; (first(LexCat,C),lexCat(LexCat),predict_static(C,LexCat),fail)).
comp_static_predictor.
```

Die bei einer Erwartung C als Kategorie der Resteingabe und einer lexikalischen Kategorie L des nächsten Eingabewortes als *brauchbar* vorhergesagten Regeln sind solche, deren Kopf A am Anfang einer C-Phrase stehen kann und deren rechte Regelseite nach einem löschbaren Präfix eine Kategorie B hat, für die L-Ausdrücke am Anfang von B-Phrasen stehen können. Bei ihrer Berechnung werden die vorher vorberechneten `del/1` und `first/2` verwendet:

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
predict_static(C,LexCat) :-
        (ein(verbose) -> nl,write('static predictor for '),write((C,LexCat)),nl ; true),
        setof((A,Alpha,[B,B2|Beta],AlphaTrees),
               C^Alphas^As^((A=C ; first(A,C)), % first(A,C) kann man nicht sparen!
                            (A --> Alphas),
                            tupleToList(Alphas,As),
                            append(Alpha,[B,B2|Beta],As), % erzwingt Verzweigung, braucht chain/2.
                            deletables(Alpha),
                            instance(BI,B),
                            (instance(LexCat,BI) ; first(LexCat,BI)),
                            canonicalTrees(Alpha,AlphaTrees)),

              Rules),
        (ein(verbose)
        -> write('Vorausgesagte Regeln: '),length(Rules,Lg),write(Lg),nl
        ; true),
        add_predicted(C,LexCat,Rules).

add_predicted(C,LexCat,[(A,Alpha,[B,B2|Beta],AlphaTrees)|Rules]) :-
        add(predicted(C,LexCat,A,Alpha,[B,B2|Beta],AlphaTrees)),
        add_predicted(C,LexCat,Rules).
add_predicted(_C,_LexCat,[]).
```

Das Prädikat `canonicalTrees/2` ist dasselbe wie in <*Earley-Pratt-Leiß*>:

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
canonicalTrees([B|Bs],[[B,[]]|BsTrees]) :-
    canonicalTrees(Bs,BsTrees).
canonicalTrees([],[]).
```

Bei der Auswahl `predictor(uniform)` werden statt der für jedes `(C,L)` bestimmten brauchbaren Regeln nur all die Regeln mit löschbarem Präfix der rechten Seite gesammelt und dann zur Parsezeit die für die jeweilige Erwartung `C` und lexikalische Kategorie `LexCat` des nächsten Worts passenden ausgewählt:

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
predict_static :-
        (ein(verbose) -> nl,write('static predictor (uniform)'),nl ; true),
        setof((A,Alpha,[B,B2|Beta],AlphaTrees,Cond),
              Alphas^As^(clause(A --> Alphas,Cond),
                         tupleToList(Alphas,As),
                         append(Alpha,[B,B2|Beta],As),
```

```
                    deletables(Alpha),
                    canonicalTrees(Alpha,AlphaTrees)),
               Rules),
        (ein(verbose)
        -> write('Vorausgesagte Regeln: '),length(Rules,Lg),write(Lg),nl
        ; true),
        add_predicted(Rules).
```

Mit der Option `ein(predictUniform)` werden –nur bei der Auswahl `predictor(uniform)`– die als brauchbar vorhergesagten Grammatikregeln nicht durch (viele) einzelne Prolog-*Fakten* zu `predicted/6` gespeichert, sondern durch (wenige) Prolog-*Regeln*:[11]

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
  add_predicted([(A,Alpha,[B,B2|Beta],AlphaTrees,Cond)|Rules]) :-
        (ein(predictUniform)
        -> add(predicted(C,LexCat,A,Alpha,[B,B2|Beta],AlphaTrees)
                   :- ((A = C ; first(A,C)), instance(BI,B),
                          (LexCat = BI ; first(LexCat,BI)), lexCat(LexCat), Cond))
        ; add_predicted_instances((A,Alpha,[B,B2|Beta],AlphaTrees,Cond))),
        add_predicted(Rules).
  add_predicted([]).
```

Ohne die Option werden für die Grammatikregeln, deren löschbare Präfixe der rechten Regelseiten beseitigt sind, und die möglichen Kategorienpaare `(C,L)` die jeweils relevanten Instanzen gebildet und in `predicted/6` abgespeichert:

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
  add_predicted_instances((A,Alpha,[B,B2|Beta],AlphaTrees,Cond)) :-
        (A = C ; first(A,C)),
        instance(BI,B),
        (LexCat = BI ; first(LexCat,BI)),
        lexCat(LexCat),
        add(predicted(C,LexCat,A,Alpha,[B,B2|Beta],AlphaTrees) :- Cond),
        fail.
  add_predicted_instances((A,Alpha,[B,B2|Beta],AlphaTrees,Cond)).
```

5.3.9 Transformation der Grammatik

Welche Eigenschaften der Grammatik man zur Beschleunigung der Syntaxanalyse vorberechnen kann oder sollte, hängt vom jeweiligen Analyseverfahren ab. Ist ein Analyseverfahren und eine Grammatik ausgewählt (vgl. Abschnitt 1) so kann im Menü (oder in Prolog mit `vorberechnen/0`) das folgende Prädikat `precomputations/1` aufgerufen werden, das die Vorberechnung der ausgewählten Grammatik für den ausgewählten Parser durchführt.

⟨*Parser/vorberechnen.pl*⟩≡

```
  :- op(1100,xfx,'-->').
  :- dynamic (-->)/2.
  :- dynamic clause/1.
  :- ['Parser/cfg-vorberechnungen.pl'].  % Neu laden, damit :- dynamic del/1,... gilt
```

⟨*Vorberechnungen*⟩
⟨*Regeltransformation*⟩

[11] Dadurch wird die Analyse langsamer, aber der statische Predictor ist übersichtlicher, also besser zur Fehlersuche geeignet.

Die Vorberechnung der Hilfsprädikate zu einer Grammatik in einer Datei `<Datei>.grm` geht wie folgt:

(i) Alle `-->`-Regeln einer vorhandenen Grammatik und die Ergebnisse einer vorherigen Vorberechnung werden gelöscht; ebenso die Restriction der Kategorien. Dann wird die Grammatik aus der angegebenen Datei `G` geladen:

⟨*Vorberechnungen*⟩≡

```
precomputations(G) :-
        retractall(clause(_)),
        retractall('-->'(_,_)),
        abolish(restrict/2),
        consult(G),
        ((parser(earley_pratt_leiss), ein(pruefeWortkategorien), wortkategorie(_))
        -> write(user, '\n*****  Prüfe, ob nur Wortkategorien mit Restriktion deklariert sind.\n\n'),
            unbeschraenkte_wortkategorien(user) ; true),
```

(ii) Die löschbaren Kategorien, `del/1`, und die lexikalischen Kategorien, `lexCat/1`, werden berechnet

⟨*Vorberechnungen*⟩+≡

```
        ensure_loaded('Parser/cfg-vorberechnungen.pl'),
        compute_deletable_categories,
        (ein(verbose) -> nl,listing(del/1) ; true),
        compute_lexCat,
        (ein(verbose) -> nl,listing(lexCat/1) ; true),
```

(iii) Für die verbesserte Form des Earley-Parsers wird die Ketten- und die `first`-Relation zwischen Kategorien berechnet:

⟨*Vorberechnungen*⟩+≡

```
        (parser(earley_pratt_leiss) ->
                compute_chain_relation,
                (ein(verbose) -> nl,listing(chain/2) ; true),
                nl,write('*****  Computing the first-relation ... (wait) '),nl,nl,
                compute_first_relation,
                (ein(verbose) -> nl,listing(first/2) ; true)
        ; true),
```

(iv) Für die verbesserte Form des Earley-Parsers werden die vorhersagbaren Grammatikregeln als `predicted/6` vorberechnet, für jedes Paar von möglicher Erwartungskategorie und lexikalischer Kategorie; hieraus wird zur Analysezeit eine Teilmenge ausgesucht:

⟨*Vorberechnungen*⟩+≡

```
        (parser(earley_pratt_leiss)
        -> compute_predictable_rules,
           (ein(verbose) -> nl,listing(predicted/6) ; true)
        ; true),
```

(v) Nun werden die Regeln von der DCG-Notation in die Darstellung mit Listen als rechter Regelseite umgeformt:

⟨*Vorberechnungen*⟩+≡

```
        regeln_transformieren,
        (ein(verbose) -> nl,listing('-->'/2) ; true),
```

(vi) Unter neuem Dateinamen `<Datei>.grm.ed.pl` bzw. `<Datei>.grm.epl.pl` werden die berechneten Prädikate und die umgeformten Regeln abgespeichert:

⟨*Vorberechnungen*⟩+≡

```
atom_concat(BaseG,'.grm',G),
(parser(earley_doerre) -> atom_concat(BaseG,'.grm.ed.pl',PrepGrm) ; true),
(parser(earley_pratt_leiss) -> atom_concat(BaseG,'.grm.epl.pl',PrepGrm) ; true),
expand_file_name(PrepGrm,[Grammar]), % expand the path to the home-directory
nl,write('*****  Exporting the transformed grammar to file '),
nl,write('*****  '),write(Grammar),nl,
(parser(earley_doerre)
->
    Kopfzeile = ' Precomputed for parser: Parser/Earley/earley.doerre.pl ',
    Regelzeile = '/***** Non-deleting rules  A --> B,... (transformed) ******/'
;
    Kopfzeile = ' Precomputed for parser: Parser/Earley/earley.pratt.leiss.pl ',
    Regelzeile = '/***** Branching rules  A --> B,C,... (transformed) ******/'),
tell(Grammar),
write('/*********************************************************'),nl,
write(' File: '),write(PrepGrm),nl,
write(' Source grammar: '),write(G),nl,
write(Kopfzeile),nl,
(parser(earley_pratt_leiss)
-> write(' Parameters: First-Relation: '),
    basefirst(BF),tc(TC),write(BF),write(','),write(TC),nl,
    write('             Predictor:      '),
    predictor(PR),write(PR),nl
; true),
writeDate,nl,
write(' *********************************************************/'),nl,
nl,
(parser(earley_pratt_leiss)
-> portray_clause(parser_version((epl,first(BF,TC),predictor(PR)))),nl ; true),
(parser(earley_doerre)
-> portray_clause(parser_version(ed)), nl ; true),
nl,portray_clause((:- op(1100,xfx,-->))),
nl,write('/*****            Auxiliary predicates               ******/'),nl,
list_auxiliary_predicates,
nl,write(Regelzeile),nl,
listing(-->),
nl,write('/*****             Deletable categories             ******/'),nl,
listing(del/1),
listing(deletables/1),
listing(deletable/1),
nl,write('/*****              Lexical categories              ******/'),nl,
listing(lexCat/1),
(parser(earley_pratt_leiss) ->
    nl,write('/*****     Chain rules: chain(A,B) iff A -->* B     ******/'),nl,
    listing(chain),
    nl,write('/***** First-relation: first(B,A) iff A -->* B beta ******/'),nl,
    listing(first),
    nl,write('/***** Static predictor: predicted(C,LexC,A,Alpha,[B|Beta],Tree) ******/'),nl,
    listing(predicted)
; true),
```

```
        nl,write('/*****                   End                         ******/'),
        told,
        % nl,write('*****  Precomputation of the grammar is finished.'),
        nl,write('*****  Transformed grammar will now be loaded.'),
        abolish('-->'/2),
        [Grammar].
```

Das Erzeugungsdatum wird in der Datei festgehalten, damit man nach Änderungen der Originalgrammatik erkennen kann, ob die transformierte Grammatik noch aktuell ist:

⟨*Vorberechnungen*⟩+≡

```
writeDate :-
        get_time(X),convert_time(X,Year,Month,Day,Hour,Min,Sec,_),
        write(' Date: '),
        write(Day),write('.'),
        write(Month),write('.'),
        write(Year), nl,
        write(' Time: '),
        write(Hour),write(':'),
        write(Min), write(':'),
        write(Sec), write(' h'),
        write('.').
```

Man muß noch die Hilfsprädikate, die neben den `-->`-Klauseln zur Definition der Grammatikregeln in `<Datei>.grm` verwendet wurden, in die transformierte Grammtik übertragen:

⟨*Vorberechnungen*⟩+≡

```
list_auxiliary_predicates :-
    setof(P,Ps^(auxiliary_predicates(Ps),member(P,Ps)),Predicates),
    listing(Predicates).
```

Solche Hilfsprädikate können zum Beispiel die Festlegungen von Merkmalbereichen, z.B. `kasus/1`, sein. Das Prädikat `auxiliary_predicates/1` *muß* in der Grammatikdatei `<Datei>.grm` erklärt sein. (Damit man eine Grammatik auf verschiedene Dateien verteilen kann, werden mehrere Klauseln zu `auxiliary_predicates/1` durchsucht und Doppeleinträge entfernt.)

Bem. Zum Schluß werden die alten `-->`-Regeln mit `abolish/1` beseitigt, da man zur Regeltransformation am Anfang `dynamic '-->'/2` erklärt hatte. Damit beim Laden einer Grammatik die vorhandenen `-->`-Regeln in Zukunft überschrieben werden, muß man `-->` wieder zu einem statischen Prädikat machen!

Grammatikregeln umformen

Um die Grammatikregeln in eine Form zu bringen, bei der auf den rechten Seiten Listen statt Tupel von Kategorien stehen, verpacken wir die `-->`-Klauseln zuerst durch `clause/1` und ersetzen dann jede Regel durch eine neue `-->`-Klausel.

⟨*Regeltransformation*⟩≡

```
regeln_transformieren :-
        toClause,
        transform_rhs.
```

Die `-->`-Regeln der DCG werden ersetzt durch „geschützte“ Regeln `clause(A --> B :- C)`.

⟨*Regeltransformation*⟩+≡
```
toClause :-
        clause(A --> B,C),
        retract((A --> B) :- C),
        assertz(clause((A --> B) :- C)),
        fail.
toClause.
```

Die durch `clause/1` geschützten Regeln werden beseitigt und als `-->`-Regeln mit Listen als rechter Regelseite in die Datenbank geschrieben.

⟨*Regeltransformation*⟩+≡
```
transform_rhs :-
        clause((A --> B) :- C),
        tupleToList(B,L),
        retract(clause((A --> B) :- C)),
        assertz((A --> L) :- C),
        fail.
```

Da Löschregel durch `del/1` gesondert erfaßt sind, werden sie dabei beseitigt; für den verbesserten Parser werden auch die Kettenregeln beseitigt, da sie durch `chain/2` behandelt werden:

⟨*Regeltransformation*⟩+≡
```
transform_rhs :-
        retractall(_ --> [[]]),
        (parser(earley_pratt_leiss) -> retractall(_ --> [_]) ; true).
```

Es ist einfach, (nicht-leere) Tupel in Listen umzuwandeln. Für Regeln mit variabler Konstituentenfolge (d.h. bei Metaregeln) muß eine Variable als Sonderfall zuerst behandelt werden.

⟨*Regeltransformation*⟩+≡
```
tupleToList(A,[A]) :-
        var(A),!.
tupleToList((A,B),[A | C]) :-
        !,tupleToList(B,C).
tupleToList(B,[B]).
```

Falls man DCG-Regeln erlaubt, müssen hier natürlich weitere Fälle eingebaut werden.

5.3.10 Earley-Parser mit Vorberechnungen

Die folgende Variante des Earley-Parsers erzeugt alle *kanonischen* Analysebäume der Eingabe. Dabei werden für Löschungen $A \to^* \epsilon$ und Ketten $A \to^* B$ nur jeweils *eine* Analyse zugelassen; daher gibt es für jede Eingabe nur noch endlich viele (kanonische) Analysen. Jeder mit den Regeln der Grammatik analysierbare Eingabe hat auch eine kanonische Analyse.

Es wird vorausgesetzt, daß die Grammatik in der nach Abschnitt 5.3.3 gegebenen Form vorliegt, also vorberechnete Prädikate `del/1`, `first/2`, `chain/2`, `predicted/6` hat. Bäume werden durch Listen `[Wurzel|Teilbäume]` dargestellt.

Das Feld $F(i,j,A \to \alpha \cdot \beta)$ der Tabelle enthält die *partiellen* Analysen $\vec{t}$ von $w(i,j)$, bei denen α die Folge der Wurzeln von $\vec{t}$ und $A \to \alpha\beta$ eine Syntaxregel ist. Wir speichern die *echt partiellen* Analysen, bei denen $\beta \neq \epsilon$ ist, unter

$$\texttt{ptrees}(\texttt{I},\texttt{J},\texttt{Kategorie},\texttt{Alpha},\texttt{Beta},\texttt{Baumliste}).$$

In $F(i,j,A) := \bigcup\{F(i,j,A \to \alpha\cdot) \mid A \to \alpha\}$ fassen wir die *vollständigen* Analysen von $w(i,j)$ mit Wurzel A zusammen. Diese speichern wir unter

ctrees(I, J, Kategorie, Baum),

wobei nicht mehr nach der an der Wurzel verwendeten Regel unterschieden wird.

Zuerst werden die Prädikate `parse/0`, `erkenne/0` wie bei dem Earley-Parser aus Abschnitt 5.3.1 geladen und die sich während der Analyse ändernden Prädikate deklariert:

⟨*Parser/earley.pratt.leiss.pl*⟩≡
```
:- ensure_loaded('Parser/main.pl').
:- dynamic inputlaenge/1, expect/2, ctrees/4, ptrees/6.
```
⟨*Earley-Pratt-Leiß*⟩

Der Hauptaufruf ist wieder `parse(Wortliste+,Baum?)`:

⟨*Earley-Pratt-Leiß*⟩≡
```
parse([],_) :-
    !,write('Fehler: Leere Eingabe.\n'),fail.
parse([W|Ws],Baum) :-
    retractall(ctrees(_,_,_,_)),
    retractall(ptrees(_,_,_,_,_,_)),
    retractall(expect(_,_)),
    retractall(inputlaenge(_)),
    length([W|Ws],N),
    assert(inputlaenge(N)), % s. complete2
    earley([W|Ws],0),
    !,extract(0,N,Baum).
```

Hier und im Folgenden sind Fakten bestimmter Form zur Datenbank hinzuzunehmen, wenn sie nicht schon darin vorkommen oder Instanzen vorhandener Fakten sind. (Ob ein Term eine Instanz vorhandener Einträge ist, wird wie bei den Vorberechnungen durch Erzeugen einer neuen Grundinstanz und Testen ihrer Beweisbarkeit festgestellt.)

⟨*Earley-Pratt-Leiß*⟩+≡
```
add(Fact) :-
        known_instance(Fact),!.
add(Fact) :-
        (ein(verbose) -> write('adding fact: '),write(Fact),nl ; true),
        assertz(Fact).
```

Durch `earley/2` wird die Tabelle aufgebaut; die abgeschlossenen Analysen sind darin leicht zu finden, da sie in `ctrees/4` abgetrennt wurden. Da die Tabelle nur kanonische Bäume sammelt, muß eine Kette von einem Startsymbol zur Kategorie des gefundenen Baums ergänzt werden:

⟨*Earley-Pratt-Leiß*⟩+≡
```
extract(I,J,Baum) :-
        start_symbol(S),
        ctrees(I,J,Cat,Tree),
        (chain(S,Cat) -> Baum = [S,Tree] ; S = Cat, Baum = Tree).
```

Als Kategorien der gesamten Wortfolge erwartet man nur die Hauptkategorien, die mit einer Kategorie `Cat` des ersten Wortes verträglich sind:

⟨*Earley-Pratt-Leiß*⟩+≡
```
  expect_initial(Cat) :-
          start_symbol(C),
          first(Cat,C),           % ; C = Cat
          addExpect(0,C),
          fail.
  expect_initial(_).
```

Welche Kategorien man für $w(j,n)$ mit $0 < j$ erwartet, bestimmt man aus partiellen Analysen der $w(i,j)$ während der Analyse, in `complete2/3` unten.

Da keine disjunktiven und optionalen Regelköpfe erlaubt sind, braucht man auch keine disjunktiven oder optionalen Erwartungen:

⟨*Earley-Pratt-Leiß*⟩+≡
```
  addExpect(J,(B;C)) :-
          !,addExpect(J,B),addExpect(J,C).
  addExpect(J,{B}) :-
          !,addExpect(J,B).
  addExpect(J,C) :-
          (C = [] -> true ; add(expect(J,C))).
```

Die Hauptschleife, `earley/2`, liest die Wortfolge der Eingabe von links nach rechts. Eine Verbesserung des Parsers besteht darin, *zuerst* das Eingabewort $w(j, j+1)$ zu lesen: seine Kategorien kann man dann bei der Vervollständigung der Analysen aller $w(i,j)$ und bei der Vorhersage „brauchbarer“ Regeln für die Reste $w(j,n)$ ausnutzen, um überflüssige Einträge zu vermeiden:

⟨*Earley-Pratt-Leiß*⟩+≡
```
  earley([W|Ws],J) :-
          M is J + 1,
          scan(J,M,W),
          complete(J),
          predict(J),
          earley(Ws,M).
  earley([],J) :-
          !,complete(J).
```

Um ein Eingabewort zu verarbeiten, werden wieder alle lexikalischen Lesarten des Worts, die an dieser Position sinnvoll scheinen, zu Wurzeln entsprechender Analysebäume:

⟨*Earley-Pratt-Leiß*⟩+≡
```
  scan(I,J,W) :-
          (ein(verbose) -> write('scan'),writeTriple(I,J,W),nl ; true),
          lexword(I,W,Cat),
          add(ctrees(I,J,Cat,[Cat,W])),
          fail.
  scan(_,_,_).
```

Um bei j „brauchbare“ Regeln vorauszusagen, sucht man aus den vorberechneten brauchbaren Regeln unter `predicted/6` die aus, die zu einer für $w(j,n)$ erwarteten Kategorie `C` und einer Kategorie `Cat` von $w(j, j+1)$ passen.

Von diesen als bei der Vorberechung als möglicherweise brauchbar eingestuften Regeln können wir (mit der Option `omitRulesWithUselessLexCat`) noch diejenigen weglassen, deren rechte Seite eine lexikalische Kategorie enthält, die keine Kategorie eines der Restwörter der Eingabe ist. (Versuch)

⟨*Earley-Pratt-Leiß*⟩+≡
```
predict(J) :-
        (ein(verbose) -> write('predict('),write(J),write(')'),nl ; true),
        M is J+1,
        (ein(omitRulesWithUselessLexCat)
        -> inputLexCats(J,InputLexCats) ; true),
        ctrees(J,M,Cat,[Cat|_]),
        (J = 0 -> expect_initial(Cat) ; true),
        expect(J,C),
        predicted(C,Cat,A,Alpha,[B|Beta],AlphaTrees),
        (ein(omitRulesWithUselessLexCat)
        -> memberLexCats([B|Beta],InputLexCats) ; true),
        (ein(omitEmptyTrees)
        -> add(ptrees(J,J,A,[],[B|Beta],[]))
        ; add(ptrees(J,J,A,Alpha,[B|Beta],AlphaTrees))),
        addExpect(J,B),
        fail.
predict(_).
```

(Die Vorberechnung kann nicht wissen, ob man später die gelöschten Bäume sehen will oder nicht; deshalb stehen sie in `predicted/6` drin, müssen dann bei der Option `omitEmptyTrees` hier aber gelöscht werden.)

⟨*Earley-Pratt-Leiß*⟩+≡
```
inputLexCats(J,LexCats) :-
        inputLexCats(J,[],LexCats).
inputLexCats(J,Acc,Cats) :-
        inputlaenge(N),
        setof(C,Word^lexword(J,Word,C),Cs),
        append(Cs,Acc,CsAcc), % muss man billiger machen!
        K is J+1,
        (K < N -> inputLexCats(K,CsAcc,Cats) ; Cats = CsAcc).
```

Mit `memberchk/2` wird (noch:fehlerhaft!) getestet, ob eine Kategorie `A`, falls sie eine lexikalische Kategorie ist, in der Liste `LexCats` vorkommt; die durch `memberchk/2` verursachten Bindungen werden mit `\+`
`\+` wieder aufgehoben:

⟨*Earley-Pratt-Leiß*⟩+≡
```
memberLexCats([A|As],LexCats) :-
        (lexCat(A) -> \+ \+ memberchk(A,LexCats) ; true),
        memberLexCats(As,LexCats).
memberLexCats([],_).
```

Bem.1: Beachte, daß `lexCat/1` nur *einfache* Kategorien enthält. Daher werden optionale oder disjunktiv verbundene „Kategorien“ aus der Eingabeliste ignoriert. Das ist korrekt, da dies keine notwendigen Konstituenten einer mit einer entsprechenden Grammatikregel aufgebauten Phrase sind.

Bem.2: Es scheint, als habe dieser Filter im Zusammenhang mit der Auswahl `predictor(uniform)` keinen großen Effekt. Die Parsegeschwindigkeit wurde in Beispielen nur um ca. 1/15 gesenkt. Mit der Auswahl `predictor(explicit)` wurde die Parsegeschwindigkeit allerdings um 1/2 gesenkt!

Nachdem das Wort $w(j, j+1)$ gelesen wurde, geht man zurück zur vorherigen Positionen $i \leq j$ und vervollständigt die Analysen von $w(i, j)$, indem man Analysen von $w(i, k)$ mit denen von $w(k, j)$ kombiniert, für jedes $k = j, j-1, \ldots, i$.

$\langle$*Earley-Pratt-Leiß*$\rangle+\equiv$
```
complete(J) :-
        (0 < J -> I is J -1, complete(I,J) ; true).
complete(I,J) :-
        K is J - 1,
        complete(I,K,J),
        (0 < I
        -> M is I - 1,
            complete(M,J)
        ; true).
complete(I,K,J) :-  % K < J garantiert
        (ein(verbose) -> write('complete'),writeTriple(I,K,J),nl ; true),
        ((0 =< I,  I < K) -> complete1(I,K,J) ; true),
        ((0 =< I, I =< K) -> complete2(I,K,J) ; true),
        (I < K -> M is K - 1, complete(I,M,J) ; true).
```

Bei der Vervollständigung der Analysen aller $w(i, j)$ verbindet man eine partielle Analyse $\vec{t} \in F(i, k, A \to \alpha \cdot B\beta)$ von $w(i, k)$ mit einer vollständige Analyse $s \in F(k, j, D)$ von $w(k, j)$, wenn jeder Ausdruck der Kategorie D auch die Kategorie B hat. Man unterscheidet zwei Fälle:[12]

(i) β enthält nur löschbare Kategorien: Dann ergänzt man s „kanonisch" zum Baum s' einer B-Phrase, bildet „kanonische" Löschungen $\vec{e}$ der β und hat eine *vollständige* Analyse von $w(i, j)$ mit Wurzel A und Teilbäumen $\vec{t}s'\vec{e}$:

$\langle$*Earley-Pratt-Leiß*$\rangle+\equiv$
```
complete1(I,K,J) :-
    ptrees(I,K,A,_Alpha,[B|Beta],AlphaTrees),
    instance(BI,B),
    (chain(BI,D) ; BI = D),
    ctrees(K,J,D,DTree),
    deletables(Beta),
    (instance(D,B) -> BTree = DTree ; BTree = [BI,DTree]),
    (ein(omitEmptyTrees)
    -> BetaTrees = []
    ; canonicalTrees(Beta,BetaTrees)),
    append(AlphaTrees,[BTree|BetaTrees],Subtrees),
    add(ctrees(I,J,A,[A|Subtrees])),
    fail.
complete1(_,_,_).

instance(D,(A,B)) :-
        !,instance(D,A).
instance(D,(A;B)) :-
        !, (instance(D,A); instance(D,B)).
instance(D,{A}) :-
        !, instance(D,A).
instance(D,D).
```

[12] Mit der Option `omitEmptyTrees` kann man einstellen, daß keine leeren Teilbäume in die Analyse eingebaut werden sollen. Das kann nützlich sein, um in der Grammatik optionale Konstituenten durch löschbare Konstituenten einzuführen, die in der Analyse ggf. gar nicht erscheinen.

Da dies nur für $i < k < j$ gemacht wird, enthält die neue Analyse von $w(i,j)$ an der Wurzel einen Verzweigungsknoten.

(ii) β enthält mindestens eine Kategorie C, die mit dem nächsten Eingabewort $w(j,j+1)$ verträglich ist: Dann sucht man solche C in $\beta = \vec{B}C\gamma$, bei denen alle $\vec{B}$ löschbar sind, ergänzt wieder s „kanonisch" zum Baum s' einer B-Phrase, bildet „kanonische" Löschungen $\vec{\epsilon}$ der $\vec{B}$ und hat eine *partielle* Analyse $\vec{ts}'\vec{\epsilon} \in F(i,j,A \to \alpha B\vec{B} \cdot C\gamma)$ von $w(i,j)$:

$\langle$*Earley-Pratt-Leiß*$\rangle{+}{\equiv}$
```
complete2(I,K,J) :-
    inputlaenge(N),          % C must derive at least one input word
    J < N,
    ptrees(I,K,A,Alpha,[B|Beta],AlphaTrees),
    append(Bs,[C|Gamma],Beta),
    deletables(Bs),
    ctrees(K,J,D,DTree),
    instance(BI,B),
    (chain(BI,D) ; BI = D),
    J1 is J + 1,
    ctrees(J,J1,LexCat,_),   % LexCat = category of next word, already scanned!
    instance(CI,C),
    (first(LexCat,CI) ; LexCat = CI),
    (D = BI -> BTree = DTree ; BTree = [BI,DTree]),
    (ein(omitEmptyTrees)
    ->  append(Alpha,[B],NewAlpha),
        BsTrees = []
    ; append(Alpha,[B|Bs],NewAlpha),
        canonicalTrees(Bs,BsTrees)),
    append(AlphaTrees,[BTree|BsTrees],NewAlphaTrees),
    add(ptrees(I,J,A,NewAlpha,[C|Gamma],NewAlphaTrees)),
    addExpect(J,C),
    fail.
complete2(_,_,_).
```

Man beachte, daß später keine Löschmöglichkeiten für dieses C berücksichtigt werden müssen. Da C mit einer Kategorien `LexCat` des Worts $w(j,j+1)$ verträglich ist, ist es eine für $w(j,n)$ erwartete Kategorie und wird zu `expect/2` getan.

Wird ein leeres Stück der Eingabe als Ausdruck der Kategorie `B` verwendet, so erlauben wir nur eine *kanonische* Analyse, `[B,[]]`, als Darstellung aller Ableitungen $B \to^* \epsilon$.

$\langle$*Earley-Pratt-Leiß*$\rangle{+}{\equiv}$
```
canonicalTrees([B|Bs],[[B,[]]|BsTrees]) :-
    canonicalTrees(Bs,BsTrees).
canonicalTrees([],[]).
```

Nur zum Testen dient folgendes Prädikat:

$\langle$*Earley-Pratt-Leiß*$\rangle{+}{\equiv}$
```
writeTriple(I,K,J) :-
       write('('),write(I),write(','),write(K),write(','),write(J),write(')').
```

Bem.: Man könnte in `complete1/3` in manchen Fällen darauf verzichten, die Analyse `ctrees(I,J,A,[A|Subtrees])` in die Tabelle einzutragen: sie ist höchstens dann brauchbar, wenn die lexikalische Kategorie des nächsten Eingabesymbol $w(j, j+1)$ überhaupt auf eine A-Phrase folgen kann. Dazu kann man nach `deletables(Beta)` diesen Test einbauen:

⟨*Earley-Pratt-Leiß (Zusatzfilter, unbenutzt)*⟩≡

```
       (J < N
       -> J1 is J + 1, ctrees(J,J1,LexCat,_), follow(A,LexCat)
        ; true),
```

Das erfordert aber eine weitere Vorberechnung, `follow/2`, welche lexikalische Kategorien auf welchen Phrasenkategorien folgen können (in Ausdrücken der Erwartungskategorien).

5.3.11 Earley-Erkenner mit Vorberechnungen

Für die Fälle, wo nur die grammatische Kategorie von Ausdrücken gesucht wird, genügt statt des Parsers ein Erkenner. Wir erhalten ihn wieder aus dem Parser, indem wir den Aufbau der syntaktischen Strukturen weglassen.

Dazu müssen in den Prädikaten des Parsers die Argumente und Teile weggelassen werden, die die Baumlisten betreffen. Wir bennenen die geänderten Prädikate durch Anhängen von `_e` so um, daß es keine Kollision mit den entsprechenden Prädikaten des Parsers gibt.

Die dynamischen Prädikate für die Tabelle sind `ctrees/3` und `ptrees/5`, d.h. sie haben dieselben Prädikatnamen wie im Parser, aber eine um 1 verringerte Stelligkeit.

⟨*Earley-Pratt-Leiß*⟩+≡

```
  % Earley-Erkenner (ohne Hilfspraedikate)
  :- dynamic ctrees/3, ptrees/5.

  erkenne([],_) :-
      write('Keine leere Eingabe erlaubt.').
  erkenne([W|Ws],Kategorie) :-
      retractall(ctrees(_,_,_)),
      retractall(ptrees(_,_,_,_,_)),
      retractall(expect(_,_)),
      retractall(inputlaenge(_)),
      length([W|Ws],N),
      assert(inputlaenge(N)),
      earley_e([W|Ws],0),
      !,extract_cat(0,N,Kategorie),
      (ein(verbose)
      -> listing(ptrees(_,_,_,_,_)),listing(ctrees(_,_,_))
      ; true).

  extract_cat(I,J,S) :-
          start_symbol(S),
          ctrees(I,J,Cat),
          (chain(S,Cat) -> true ; S = Cat).

  earley_e([W|Ws],J) :-
          M is J + 1,
          scan_e(J,M,W),
          complete_e(J),
          predict_e(J),
          earley_e(Ws,M).
```

```
earley_e([],J) :-
        !,complete_e(J).
scan_e(I,J,W) :-
        (ein(verbose) -> write('scan'),writeTriple(I,J,W),nl ; true),
        lexword(I,W,Cat),
        add(ctrees(I,J,Cat)),
        fail.
scan_e(_,_,_).

predict_e(J) :-
        (ein(verbose) -> write('predict('),write(J),write(')'),nl ; true),
        M is J+1,
        ctrees(J,M,Cat),
        (J = 0 -> expect_initial(Cat) ; true),
        expect(J,C),
        predicted(C,Cat,A,Alpha,[B|Beta],_),
        add(ptrees(J,J,A,Alpha,[B|Beta])),
        addExpect(J,B),
        fail.
predict_e(_).

complete_e(J) :-
        (0 < J -> I is J -1, complete_e(I,J) ; true).
complete_e(I,J) :-
        K is J - 1,
        complete_e(I,K,J),
        (0 < I
        -> M is I - 1,
            complete_e(M,J)
        ; true).
complete_e(I,K,J) :-
        (ein(verbose) -> write('complete'),writeTriple(I,K,J),nl ; true),
        ((0 =< I,  I < K) -> complete1_e(I,K,J) ; true),
        ((0 =< I, I =< K) -> complete2_e(I,K,J) ; true),
        (I < K -> M is K - 1, complete_e(I,M,J) ; true).

complete1_e(I,K,J) :-
    ptrees(I,K,A,_Alpha,[B|Beta]),
    instance(BI,B),      % via BI: Nachtrag 23.2.02
    (chain(BI,D) ; BI = D),
    ctrees(K,J,D),
    deletables(Beta),
    add(ctrees(I,J,A)),
    fail.
complete1_e(_,_,_).

complete2_e(I,K,J) :-
    inputlaenge(N),
    J < N,
    ptrees(I,K,A,Alpha,[B|Beta]),
    append(Bs,[C|Gamma],Beta),
    deletables(Bs),
    ctrees(K,J,D),
```

```
      instance(BI,B),
      (chain(BI,D) ; BI=D),
      J1 is J + 1,
      ctrees(J,J1,LexCat),
      instance(CI,C),
      (first(LexCat,CI) ; LexCat=CI),
      addExpect(J,C),
      append(Alpha,[B|Bs],NewAlpha),
      add(ptrees(I,J,A,NewAlpha,[C|Gamma])),
      fail.
  complete2_e(_,_,_).
```

Beispiel 5.7 Für dieselbe Eingabe wie in Beispiel 5.4, den Satz `Der Hamster pfeift.`, werden mit diesem verbesserten Erkenner folgende partiellen und folgende vollständigen Analysen (oder richtiger: Kanten) in die Tabelle eingetragen:

```
ptrees(0, 0, np, [], [det, ap, n]).
ptrees(0, 0, np, [], [np, conj, np]).
ptrees(0, 0, s, [], [np, vp]).
ptrees(0, 1, np, [det, ap], [n]).
ptrees(0, 2, s, [np], [vp]).
ptrees(2, 2, vp, [], [v, np]).
ptrees(2, 2, vp, [], [v, np, av]).

ctrees(0, 1, det).
ctrees(1, 2, n).
ctrees(2, 3, v).
ctrees(0, 2, np).
ctrees(0, 3, s).
```

Das sind 12 Einträge, während der Erkenner aus Abschnitt 5.3.2 dafür 27 Einträge gebraucht hat. Man beachte aber, daß die Tabellen des verbesserten Erkenners schwerer zu verstehen sind: obwohl z.B. Einträge wie `ctrees(2, 3, vp)` nicht gemacht werden, wird bei der Erkennung neben `ctrees(2, 3, v)` implizit die Kettenregel `vp --> v` benutzt wird. (Der Eintrag mit `vp` fehlt, weil der Baum `[vp, [v, pfeift]]` nicht kanonisch ist!).

Durch Einschalten von `ein(verbose)` erhält man ein Protokoll, in welchem Schritt der Erkennung die Einträge in die Tabellen gemacht werden:

```
55 ?- erkenne.
Beende den Eingabesatz mit <Satzzeichen><Return>
|: der Hamster pfeift.
adding fact: inputlaenge(3)
scan(0,1,der)
adding fact: ctrees(0, 1, det)
predict(0)
adding fact: expect(0, s)
adding fact: expect(0, np)
adding fact: ptrees(0, 0, np, [], [det, ap, n])
adding fact: expect(0, det)
adding fact: ptrees(0, 0, np, [], [np, conj, np])
adding fact: ptrees(0, 0, s, [], [np, vp])
```

```
scan(1,2,Hamster)
adding fact: ctrees(1, 2, n)
complete(0,0,1)
adding fact: expect(1, n)
adding fact: ptrees(0, 1, np, [det, ap], [n])
predict(1)
scan(2,3,pfeift)
adding fact: ctrees(2, 3, v)
complete(1,1,2)
complete(0,1,2)
adding fact: ctrees(0, 2, np)
complete(0,0,2)
adding fact: expect(2, vp)
adding fact: ptrees(0, 2, s, [np], [vp])
predict(2)
adding fact: ptrees(2, 2, vp, [], [v, np])
adding fact: expect(2, v)
adding fact: ptrees(2, 2, vp, [], [v, np, av])
complete(2,2,3)
complete(1,2,3)
complete(1,1,3)
complete(0,2,3)
adding fact: ctrees(0, 3, s)
complete(0,1,3)
complete(0,0,3)
```

Man kann zwar die Erwartungen aus der Tabelle `ptrees/5` ablesen, aber wenn man sie –wie oben geschehen– abspeichert, muß man auch die 6 Einträge von `expect/2` zur Größe der Hilfstabellen zählen.

Aufgabe 5.4 In Aufgabe 5.3 wurde der Earley-Dörre-Parser um die Option `omitEmptyTrees` erweitert. Mit dieser Option kann verhindert werden, daß die erzeugten Bäume *breiter* werden als die analysierte Wortfolge. Man baue eine weitere Option, `boundedChains`, in den Earley-Dörre-Parser ein, die verhindert, daß die *Höhe* der erzeugten Bäume die Länge der Wortfolge unbeschränkt übersteigen kann. Man erlaube nur die Bäume, deren Wurzelmarkierung A auf dem *Stamm* des Baums, d.h. dem Pfad von der Wurzel bis zur ersten echten Verzweigung, nur einmal vorkommt.

Hinweis: Man muß in den Fällen, wo eine vollständige Analyse $s \in F(j,k,B)$ mit einer partiellen Analyse $\vec{t} \in F(i,j,A \to \alpha \cdot B)$ verbunden werden soll, folgende Fälle ausnehmen:

(i) $i = j$ und s ist eine Ableitung der Form $B \to C \to \cdots \to A \to^* w(j,k)$,

(ii) $j = k$ und $\vec{t}$ ist eine Ableitung der Form $A \to C \to \cdots \to A \to^* w(i,j)$.

Vorausgesetzt ist, daß die Option `omitEmptyTrees` schon eingeschaltet ist. Dann hat man zur Parsezeit (im Wesentlichen) verhindert, daß nicht-kanonische Bäume erzeugt werden, ohne die Vorberechnungen durchführen zu müssen.

6 Grammatik des Deutschen

6.1 Grammatische Kategorien und grammatische Relationen

Eine sorgfältige Klassifizierung der sprachlichen Ausdrücke führt zu einer großen Anzahl von Klassen. Diese Klassen lassen sich im Wesentlichen als eine Hierarchie darstellen, wobei die Unterklassen durch *Klassenparameter* angegeben werden. Durch die Wahl gleicher Parameter lassen sich Beziehungen zwischen Unterklassen verschiedener Klassen andeuten.

Klassenparameter beschreiben Eigenschaften, die bei der Zusammensetzung von Ausdrücken restringierend wirken, aber nicht variabel sind. Zum Beispiel ist das Genus bei Nominalphrasen ein Klassenparameter.

Jeder Ausdruck einer Unterklasse der Kategorie kann in verschiedenen *Formen* auftreten. Was die möglichen Formen sind, wird –abhängig von der jeweiligen Ausdrucksklasse– durch *Formparameter* beschrieben.

In der Implementierung werden die Wort- und Ausdruckskategorien als zweistellige Prädikate

```
Kategorie([Klassenparameter,...],[Formparameter,...])
```

angegeben.

Zum Beispiel ist der Kasus ein Formparameter von Nominalphrasen, der Modus ein Formparameter von Verben und Aussagesätzen, die Verbstellung ein Formparameter von Sätzen.

6.1.1 Kongruenz

Die *Kongruenz* zweier oder mehrer Teilausdrücke eines Ausdrucks ist eine Beziehung zwischen den Formen der Teilausdrücke, die von der Konstruktion des Ausdrucks abhängt. Sie wird als Beziehung zwischen den Formparametern der Kategorien einer Regel formuliert:

```
A ---> A_1(Klasse_1,Form_1), ... , A_k(Klasse_k,Form_k)
   :- kongruenz(Form_1,...,Form_k).
```

Die Kongruenzbedingung hängt von der Kategorie `A` und der jeweiligen Regel ab. Sie legt die Formen normalerweise nicht eindeutig fest, sondern beschränkt die Unabhängigkeit der Formen der Teilausdrücke.

Die Kongruenz von Teilausdrücken (in einer Formdimension) ist normalerweise an der Gleichheit der Formwerte (dieser Formdimension) zu erkennen, z.B. demselben Numerus oder Kasus. Dann kann man die Kongruenzbedingung durch gleiche Variable ausdrücken:

```
np(Klasse,[N,K]) ---> det(KlasseD,[N,K]), n(KlasseN,[N,K])
   :- numerus(N),kasus(K), bedingungen(Klasse,KlasseD,KlasseN).
```

Zur Kongruenz des finiten Verbs mit dem Satzsubjekt gehört aber z.B. auch die Tatsache, daß Subjektsätze nur mit finitem Verb in der dritten Person im Singular einen Satz bilden; trotzdem gibt man den Subjektsätzen keine Merkmale Numerus und Person.

6.1.2 Rektion

Ein Teilausdruck *regiert* die Form eines anderen Teilausdrucks in einem Ausdruck, wenn die Form des zweiten von der Unterklasse des ersten Teilausdrucks abhängt. Die Rektion wird in einer Grammatikregel als (funktionale) Beziehung zwischen der Unterklasse der Kategorie des regierenden und der Form der Kategorie des regierten Teilausdrucks formuliert:

```
A ---> A1(Klasse_1,Form_1), ... , A_k(Klasse_k,Form_k)
:- rektion(j,i,Klasse_j,Form_i).
```

Oft ist die Rektion als Gleichheit des Wertes eines Formparameters des abhängigen mit dem Wert eines Klassenparameters des unabhängigen Ausdrucks einer Konstruktion ausgedrückt:

```
np ---> det(KlasseD,[G,..]), a(KlasseA,[G,...]), n([G],Form)
:- genus(G).
```

In einer Nominalphrase haben sich die Form von Artikel und Adjektiv dem Genus des Nomens anzupassen, das (Genus des) Nomen(s) regiert also die Form (-dimension Genus) von Artikel und Adjektiv.

6.1.3 Merkmalvererbung

Merkmale einer zusammengesetzten Phrase hängen oft von den Merkmalen ihrer Teilphrasen ab; umgekehrt hängen auch die Merkmale der Teilphrasen von denen der gesamten Phrase ab. Diese Abhängigkeiten wird in einer Grammatikregel durch Bedingungen an die Merkmale formuliert:

```
A(Klasse,Form) ---> A_1(Klasse_1,Form_1), ..., A_k(Klasse_k,Form_k)
:- vererbung(Klasse_1,Form_1,...,Klasse_k,Form_k,Klasse,Form).
```

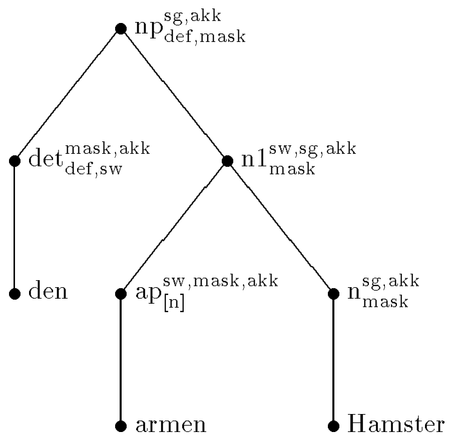
Die Klassenparameter sollten hier nicht von den Formparametern abhängen, und oft hängen auch die Formparameter nur von Formparametern ab. In vielen Fällen ist die Abhängigkeit einfach eine Gleichheit in einem Merkmal der Kopf- und Konstituentenkategeorien, z.B. beim Klassenmerkmal Genus und dem Formmerkmal Kasus der Nominalphrase:

```
np([G,...],[K,...]) ---> det([G,...],[K,...]), n([G],[K,...])
:- genus(G), kasus(K), ... .
```

Die Klasse der zusammengesetzten Phrase hängt im allgemeinen auch von der jeweiligen Grammatikregel ab. Merkmale, die nur zur Formulierung von Kongruenz und Rektion zwischen den Konstituenten dienen, sollten nicht an die gesamte Phrase vererbt werden, wie z.B. die Adjektivdeklination innerhalb der Nominalphrase.

Graphische Darstellung von Komplementrahmen, Rektion und Kongruenz

In der Baumdarstellung einer Analyse stehen dieFormparameter als obere, die Klassenparameter als untere Indizes an den Kategoriennamen. Komplementrahmen sind als Listen angegeben:



Wenn man wie hier beim Adjektiv überhaupt ein Subjekt im Komplementrahmen vorsieht, sollte es für die attributive Verwendung ein einfaches Nomen vorsehen, `[n]`, für die prädikative Verwendung eine Nominalphrase in einem bestimmten Kasus, normalerweise `[nom]`.

6.1.4 Aufbau der Grammatik aus Teilgrammatiken

Obwohl die verschiedenen Ausdruckskategorien einer Sprache simultan rekursiv von einander abhängen, muß man die Grammatik aus Teilgrammatiken für die einzelnen Kategorien zusammensetzen, um eine übersichtliche Organisation und Entwicklung zu erreichen.

Wegen der gegenseitigen Abhängigkeiten kann man nicht einfach Teilgrammatiken in einzelnen Dateien angeben, die andere Teilgrammatiken laden; sonst würde das Laden der Dateien nicht abbrechen. Wir wollen die Einzelgrammatiken so schreiben, daß

(i) Phrasenkategorien der anderen Teilgrammatiken wie lexikalische Kategorien behandelt werden,

(ii) Die Vorberechnungen der Teilgrammatiken unabhängig von einander möglich sind und in die Vorberechnung der kombinierten Grammatik eingehen.

Dazu muß jede Teilgrammatiken so aufgeteilt werden, daß

(i) ihre Prädikate zur Definition der Merkmalbereiche und ihre anderen Hilfsprädikate ohne ihre Regeln auch zur Vorberechung und Benutzung anderer Grammatiken geladen werden können,

(ii) die Vorberechnung für die Teilgrammatik separat vorgenommen werden kann,

(iii) Kombination beliebiger Teilgrammatiken durch Kombinationen von Hilfsprädikaten, Vorberechnungen und Regeln möglich ist.

Die Gesamtgrammatik lädt die Definitionen der Kategorien und ihrer Merkmale sowie die Teilgrammatiken. Bei den Vorberechnungen der Hilfsprädikate ist eine modulare Berechnung wünschenswert, hier aber nicht verwirklicht.

Bemerkung 6.1 Die Vorberechnung muß die Kongruenz- und andere Prädikate einer Teilgrammatik in die vorberechnete (Gesamt-) Grammatik übertragen. In der Teilgrammatik wird durch `auxiliary_predicates/1` die Liste dieser Prädikate angegeben. Die Gesamtgrammatik muß am Anfang eine Deklaration `:- multifile auxiliary_predicates/1.` enthalten, damit nicht die in der letzten Teilgrammatik eingeführten Prädikate die der vorigen überschreiben.

Beim erneuten Laden der Grammatik oder einer Teilgrammatik werden solche in mehreren Dateien erklärten Prädikate nur teilweise ersetzt: die Anteile, die aus der erneut geladenen Grammatik stammen, werden gelöscht und ihre Neudefinition am Ende angehängt. (Kann diese Änderung der Reihenfolge der Klauseln hier einen unerwünschsten Effekt haben?)

Ein wesentlicher Aspekt der Grammatik ist die Wahl der Klassen- und Formparameter und ihrer Wertebereiche für die verschiedenen lexikalischen und phrasalen Kategorien.

6.2 Lexikalische Kategorien und ihre Merkmale

Merkmale, die die Zugehörigkeit zu Flexionsklassen, die Passivierbarkeit von Verben, das Hilfsverb im Perfekt u.ä. beschreiben, müssen in einem Stammformenlexikon enthalten sein; in einem Vollformenlexikon, von dem hier ausgegangen wird, braucht man sie nicht. Nur wenn man Ausdrücke in verschiedene Formen setzen will, braucht man diese Informationen; dafür genügt es, wenn die Vollform einen Verweis auf die Stammform enthält.

Da wir hier nur an der Analyse interessiert sind, können wir auch den Verweis auf die Stammform bei den Vollformen weglassen. Wir legen uns (solange es geht) auf folgende lexikalischen Kategorien und ihre Merkmale fest.

Warnung: Von den Festlegungen aus der Datei `Lexika/wortarten.cat.pl` *werden bisher nur die Teile <Lexikalische Merkmale> benutzt, und das auch nicht in den Grammatiken aus* `Seminarbeitraege`. *Das Prädikat* `wortkategorie/1` *wird bisher nur zu Dokumentations- und Planungszwecken benutzt, es soll aber mittelfristig die tatsächlich verwendeten lexikalischen Kategorien beschreiben und zu Korrektheitsprüfungen benutzt werden, z.B. für eine Schnittstelle zwischen Lexikon und Grammatik.*

⟨*Lexika/wortarten.cat.pl*⟩≡
```
% :- module(wortarten,
%           [wortkategorie/1,restrict/2,
%            verbklasse/1,finiteVerbform/1,finiteVerbformPraef/1,subjekt/1,objekt/1,
%            verbtempus/1,modus/1,numerus/1,person/1,verbstellung/1,infiniteVerbform/1,
%            genus/1,numerus/1,kasus/1,definitheit_pron/1,
%            adjektivdeklination/1,komparation/1,adverbialeFunktion/1]).
:- multifile restrict/2.
```

⟨*Wortkategorien*⟩
⟨*Lexikalische Merkmale*⟩

6.2.1 Verben

(i) finite Verben: wir unterscheiden zwei Fälle, normale Verben und solche, die bei Verberst- und Verbzweitstellung ein Präfix abspalten. Für die zweite Sorte ist die Verbstellung im Satz daher ein Formparameter, und das abgespaltene Präfix wird wie ein Komplement als Klassenparameter im Lexikoneintrag notiert:

⟨*Wortkategorien*⟩≡
```
wortkategorie(v(VerbKlasse,FiniteForm)) :-
        verbklasse(VerbKlasse),
        finiteVerbform(FiniteForm).
wortkategorie(v(VerbKlasse,FiniteForm)) :-
        verbklassePraef(VerbKlasse),
        finiteVerbformPraef(FiniteForm).
```

⟨*Lexikalische Merkmale*⟩≡
```
verbklasse([Rahmen]) :-
        komplemente(Rahmen).
finiteVerbform([Pers,Num,Temp,Mod]) :-
        person(Pers),
        numerus(Num),
        verbtempus(Temp),
        modus(Mod).

verbklassePraef([Rahmen,Praef]) :-
        verbklasse([Rahmen]),
        praefix(Praef).
finiteVerbformPraef([Pers,Num,Temp,Mod,Vst]) :-
        finiteVerbform([Pers,Num,Temp,Mod]),
        verbstellung(Vst).
```
Die abtrennbaren Verbpräfixe, `vpr(Praef)`, muß man `Lexika/praepositionen.wrd` entnehmen.

⟨*Lexikalische Merkmale*⟩+≡
```
praefix(Praef) :-
        ensure_loaded('Lexika/praepositionen.wrd'),
        word(_, vpr(Praef)).
```
Jeder Komplementrahmen ist eine Liste, die das Subjekt und die Objekte beschreibt:

⟨*Lexikalische Merkmale*⟩+≡
```
komplemente([C1]) :-
        subjekt(C1).
komplemente([C|Cs]) :-
        subjekt(C),
        objekte(Cs).
```
Dabei werden die Komplemente nur in der Reihenfolge angegeben, in der sie in Sätzen mit Verbletztstellung auftreten. (Die übrigen Stellungen werden in der Grammatik durch Regeln formuliert. Optionale Argumente werden vorläufig durch eigene Komplementrahmen, aber mittelfristig wie in `word(lese,v([ [nom,{akk}] ],Form))` angegeben.)

Jedes Komplement wird durch eine Abkürzung seiner Ausdrucksform beschrieben, aus der seine Ausdrucksklasse hervorgeht:

⟨*Lexikalische Merkmale*⟩+≡
```
subjekt(nom).
subjekt(gen).
subjekt(dat).
subjekt(es).
subjekt(daß).
subjekt(inf).
subjekt(infzu).

objekte([]).
objekte([C]) :- objekt(C).
objekte([C,D]) :- objekt(C), objekt(D).

objekt(gen).
objekt(dat).
objekt(dat(refl)).
```

```
objekt(akk).
objekt(akk(refl)).
objekt(praep(P)).
objekt(daß).
objekt(qu).
objekt(inf).
objekt(infzu).
objekt(infzu(nom)).
objekt(infzu(dat)).
objekt(infzu(akk)).
objekt(infzu(praep(P))).
```

Bem.: Durch `objekt(P) :- praeposition(P).` werden die Komplementrahmen vom Lexikon abhängig und vor allem werden dann zu viele Grammatikregeln entstehen.

Die Unterklassen, die für die Syntaxanalyse unterschieden werden müssen, kann man an den Verbrahmen ablesen: intransitive Verben haben kein Objekt, Modalverben sind diejenigen mit einem `inf`-Objekt, reflexive Verben diejenigen mit einem `dat(refl)`- oder `akk(refl)`-Objekt. Durch `qu` werden interrogative Objekte bei Verben wie `fragen`, `nicht wissen` usw. angegeben.

Kontrollverben haben `infzu`-Objekte, die durch `infzu(Kasus)` dargestellt werden: der Kasus gibt an, welches Nominalobjekt des Kontrollverbs das implizite Subjekt des Infinitivs ist; falls im Infinitivobjekt Reflexivpronomina vorkommen, müssen diese mit dem impliziten Subjekt in Person und Numerus übereinstimmen: z.B. hat `raten` den Komplementrahmen `[nom,dat,infzu(dat)]` und `versprechen` den Rahmen `[nom,dat,infzu(nom)]` . Analog ist `infzu(praep(C))` für Kontrollverben mit Präpositionalobjektkontrolle gedacht, z.B. hat `appellieren` den Rahmen `[nom,praep(an),infzu(praep(an))]` .

Die Formmerkmale der finiten Verbformen haben folgende Wertebereiche:

⟨*Lexikalische Merkmale*⟩+≡

```
verbtempus(praes).
verbtempus(praet).

modus(ind).
modus(konj).

numerus(sg).
numerus(pl).

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

verbstellung(erst).
verbstellung(zweit).
verbstellung(letzt).
```

Die Verbstellung ist eigentlich ein Formmerkmal von einfachen Sätzen, das aber eben auch die Abspaltung von Verbpräfixen regiert und deshalb hier auftritt.

(ii) infinite Verbformen: Bei der Partizipform eines Verbs, die zur Bildung von Säzten im Perfekt gebraucht wird, notieren wir auch das Hilfsverb, mit dem das Perfekt gebildet wird.

⟨*Wortkategorien*⟩+≡

```
wortkategorie(v(VerbKlasse,InfiniteForm)) :-
```

```
          infiniteVerbklasse(VerbKlasse),
          infiniteVerbform(InfiniteForm).
```

⟨*Lexikalische Merkmale*⟩+≡

```
infiniteVerbform([inf]).
infiniteVerbform([infzu]).
infiniteVerbform([part]).
infiniteVerbform([part2]).

infiniteVerbklasse([Rahmen]) :-
        verbklasse([_Subj|Rahmen]).
infiniteVerbklasse([Rahmen,PerfAux]) :-
        verbklasse([_Subj|Rahmen]),
        hilfsverb(PerfAux).

hilfsverb(sein).
hilfsverb(haben).
```

Bei den infiniten Formen `inf`, `infzu` und `part` entfernen wir das Subjekt aus dem Komplementrahmen. Diese Entscheidung muß man überprüfen, wenn man Satzregeln für Infinitivsätze mit Anhebungs- und Kontrollverben und Tempusformen mit Hilfsverben schreibt.

Die Hilfsverben sollten dem Lexikon entnommen werden.

6.2.2 Nomen

Vorläufig wird folgende Parametrisierung vorgeschlagen:

(i) Eigennamen:

⟨*Wortkategorien*⟩+≡

```
wortkategorie(en([Gen],[Num,Kas])) :-
        genusErw(Gen), numerus(Num), kasus(Kas).
wortkategorie(vn([Gen],[Num,Kas])) :-
        genus(Gen), numerus(Num), kasus(Kas).
wortkategorie(nn([],[Num,Kas])) :-
        numerus(Num), kasus(Kas).
```

Bei den Eigennamen `en` wird das Genus in der Regel `unsp` sein und nur bei den Vornamen `vn` ein echtes Genus sein; bei Nachnamen `nn` ist die Unterscheidung nach dem Genus unnötig. In beiden Fällen wird gelegentlich eine Pluralform verwendet, z.B. `alle Kläuse, die Meiers`, aber das wird hier ignoriert.

(ii) Personalpronomen:[13]

⟨*Wortkategorien*⟩+≡

```
% wortkategorie(pron([Def,Pers,Gen],[Num,Kas])) :-
wortkategorie(pron([Def|Pos],[Gen,Num,Kas,Pers])) :-
        definitheit_pron(Def),
        \+ (Def = rel; Def = refl),
        numerus(Num),
        person(Pers),
        (((Pers = 1 ; Pers = 2), Genus = unsp)
        ; (Pers = 3, genus(Gen), Num = sg)),
```

[13] Das wird unten leider nicht eingehalten! Dort wird `pron([Def|Pos],[Gen,Num,Kas,Pers]))` benutzt.

```
        kasus(Kas).
wortkategorie(pron([refl],[unsp,Num,Kas,Pers])) :-
        numerus(Num),
        person(Pers),
        kasus(Kas).
wortkategorie(pron([rel],[Gen,Num,Kas])) :-
        ((genus(Gen), Num = sg) ; (Gen = unsp, Num = pl)),
        kasus(Kas).
```

Die Definitheit wird zur Unterscheidung zwischen bestimmten (definiten) und unbestimmten, quantifizierenden, relativierenden, interrogativen Pronomina gebraucht.

⟨*Lexikalische Merkmale*⟩+≡

```
definitheit_pron(def).
definitheit_pron(indef).
definitheit_pron(rel).
definitheit_pron(qu).
```

(a) Definite Personalpronomen:

⟨*Lexika/pronomen.wrd*⟩≡

```
% Personalpronomen:
word(ich, pron([def,pers], [unsp, sg, nom, 1]), ppron).
word(meiner, pron([def,pers], [unsp, sg, gen, 1]), ppron).
word(mir, pron([def,pers], [unsp, sg, dat, 1]), ppron).
word(mich, pron([def,pers], [unsp, sg, akk, 1]), ppron).

word(du, pron([def,pers], [unsp, sg, nom, 2]), ppron).
word(deiner, pron([def,pers], [unsp, sg, gen, 2]), ppron).
word(dir, pron([def,pers], [unsp, sg, dat, 2]), ppron).
word(dich, pron([def,pers], [unsp, sg, akk, 2]), ppron).

word(er, pron([def,pers], [mask, sg, nom, 3]), ppron).
word(seiner, pron([def,pers], [mask, sg, gen, 3]), ppron).
word(ihm, pron([def,pers], [mask, sg, dat, 3]), ppron).
word(ihn, pron([def,pers], [mask, sg, akk, 3]), ppron).

word(sie, pron([def,pers], [fem, sg, nom, 3]), ppron).
word(ihrer, pron([def,pers], [fem, sg, gen, 3]), ppron).
word(ihr, pron([def,pers], [fem, sg, dat, 3]), ppron).
word(sie, pron([def,pers], [fem, sg, akk, 3]), ppron).

word(es, pron([def,pers], [neut, sg, nom, 3]), ppron).
word(seiner, pron([def,pers], [neut, sg, gen, 3]), ppron).
word(ihm, pron([def,pers], [neut, sg, dat, 3]), ppron).
word(es, pron([def,pers], [neut, sg, akk, 3]), ppron).

word(wir, pron([def,pers], [unsp, pl, nom, 1]), ppron).
word(unser, pron([def,pers], [unsp, pl, gen, 1]), ppron).
word(uns, pron([def,pers], [unsp, pl, dat, 1]), ppron).
word(uns, pron([def,pers], [unsp, pl, akk, 1]), ppron).

word(ihr, pron([def,pers], [unsp, pl, nom, 2]), ppron).
word(euer, pron([def,pers], [unsp, pl, gen, 2]), ppron). % ??
word(eurer, pron([def,pers], [unsp, pl, gen, 2]), ppron). % ??
```

```
word(euch, pron([def,pers], [unsp, pl, dat, 2]), ppron).
word(euch, pron([def,pers], [unsp, pl, akk, 2]), ppron).

word(sie, pron([def,pers], [unsp, pl, nom, 3]), ppron).
word(ihrer, pron([def,pers], [unsp, pl, gen, 3]), ppron).
word(ihnen, pron([def,pers], [unsp, pl, dat, 3]), ppron).
word(sie, pron([def,pers], [unsp, pl, akk, 3]), ppron).
```

(b) Indefinite Personalpronomen: (Da man zwischen positiver und negativer Version wählen kann, sollte das ein *Form*parameter sein.)

⟨*Lexika/pronomen.wrd*⟩+≡

```
% Indefinite Pronomen:
word(man, pron([indef,pers], [mask, sg, nom, 3])).  % mask!

word(einer, pron([indef,pos], [mask, sg, nom, 3]), ein).
word(eines, pron([indef,pos], [mask, sg, gen, 3]), ein). % kommt das vor?
word(einem, pron([indef,pos], [mask, sg, dat, 3]), ein).
word(einen, pron([indef,pos], [mask, sg, akk, 3]), ein).

word(eine,  pron([indef,pos], [fem, sg, nom, 3]), ein).
word(einer, pron([indef,pos], [fem, sg, gen, 3]), ein).
word(einer, pron([indef,pos], [fem, sg, dat, 3]), ein).
word(eine,  pron([indef,pos], [fem, sg, akk, 3]), ein).

word(eines, pron([indef,pos], [neut, sg, nom, 3]), ein).
word(eines, pron([indef,pos], [neut, sg, nom, 3]), ein).
word(einem, pron([indef,pos], [neut, sg, dat, 3]), ein).
word(eines, pron([indef,pos], [neut, sg, akk, 3]), ein).

word(keiner, pron([indef,neg], [mask, sg, nom, 3]), ein).
word(keines, pron([indef,neg], [mask, sg, gen, 3]), ein). % kommt das vor?
word(keinem, pron([indef,neg], [mask, sg, dat, 3]), ein).
word(keinen, pron([indef,neg], [mask, sg, akk, 3]), ein).

word(keine,  pron([indef,neg], [fem, sg, nom, 3]), ein).
word(keiner, pron([indef,neg], [fem, sg, gen, 3]), ein).
word(keiner, pron([indef,neg], [fem, sg, dat, 3]), ein).
word(keine,  pron([indef,neg], [fem, sg, akk, 3]), ein).

word(keines, pron([indef,neg], [neut, sg, nom, 3]), ein).
word(keines, pron([indef,neg], [neut, sg, nom, 3]), ein).
word(keinem, pron([indef,neg], [neut, sg, dat, 3]), ein).
word(keines, pron([indef,neg], [neut, sg, akk, 3]), ein).


word(etwas, pron([indef,pos], [unsp, sg, nom, 3]), etw). % neut
word(etwas, pron([indef,pos], [unsp, sg, akk, 3]), etw).
word(etwas, pron([indef,pos], [unsp, sg, dat, 3]), etw).
word(etwas, pron([indef,pos], [unsp, sg, gen, 3]), etw).

word(nichts, pron([indef,neg], [unsp, sg, nom, 3]), etw). % neut
word(nichts, pron([indef,neg], [unsp, sg, akk, 3]), etw).
word(nichts, pron([indef,neg], [unsp, sg, dat, 3]), etw).
```

```
word(nichts, pron([indef,neg], [unsp, sg, gen, 3]), etw).

word(jemand, pron([indef,pos], [mask, sg, akk, 3]), jmd).
word(jemand, pron([indef,pos], [mask, sg, nom, 3]), jmd).
word(jemand, pron([indef,pos], [mask, sg, dat, 3]), jmd).
word(jemandem, pron([indef,pos], [mask, sg, dat, 3]), jmd).
word(jemanden, pron([indef,pos], [mask, sg, akk, 3]), jmd).
word(jemandes, pron([indef,pos], [mask, sg, gen, 3]), jmd). % poss?

word(niemand, pron([indef,neg], [mask, sg, akk, 3]), jmd).
word(niemand, pron([indef,neg], [mask, sg, nom, 3]), jmd).
word(niemand, pron([indef,neg], [mask, sg, dat, 3]), jmd).
word(niemandem, pron([indef,neg], [mask, sg, dat, 3]), jmd).
word(niemanden, pron([indef,neg], [mask, sg, akk, 3]), jmd).
word(niemandes, pron([indef,neg], [mask, sg, gen, 3]), jmd). % poss?
```

(c) Reflexivpronomen:

⟨*Lexika/pronomen.wrd*⟩+≡

```
% Reflexivpronomen:
word(mir,  pron([refl],[unsp, sg, dat, 1]), rpron).
word(mich, pron([refl],[unsp, sg, akk, 1]), rpron).

word(dir,  pron([refl],[unsp, sg, dat, 2]), rpron).
word(dich, pron([refl],[unsp, sg, akk, 2]), rpron).

word(sich, pron([refl],[unsp, sg, dat, 3]), rpron).
word(sich, pron([refl],[unsp, sg, akk, 3]), rpron).

word(uns,  pron([refl],[unsp, pl, dat, 1]), rpron).
word(uns,  pron([refl],[unsp, pl, akk, 1]), rpron).

word(euch, pron([refl],[unsp, pl, dat, 2]), rpron).
word(euch, pron([refl],[unsp, pl, akk, 2]), rpron).

word(sich, pron([refl],[unsp, pl, dat, 3]), rpron).
word(sich, pron([refl],[unsp, pl, akk, 3]), rpron).

word(einander, pron([refl], [unsp, pl, nom, 3]), rpron).
word(einander, pron([refl], [unsp, pl, akk, 3]), rpron).
word(einander, pron([refl], [unsp, pl, dat, 3]), rpron).
word(einander, pron([refl], [unsp, pl, gen, 3]), rpron).
```

Man sollte vielleicht `einander` nicht als `refl`, sondern als `rezi` für *reziprok* klassifizieren; es sollte nur bei Verben verwendbar sein, deren Subjekt notwendigerweise eine Nominalphrase im Plural ist, die als kollektives Subjekt zu verstehen ist (keine distributive Bedeutung).

(d) Interrogativpronomen: (vs. interrogative Determinatoren)

⟨*Lexika/pronomen.wrd*⟩+≡

```
word(wer, pron([qu], [unsp, sg, nom, 3])).   % statt qu auch rel!
word(wessen, pron([qu], [unsp, sg, gen, 3])).
word(wem, pron([qu], [unsp, sg, dat, 3])).
word(wen, pron([qu], [unsp, sg, dat, 3])).

word(was, pron([qu], [neut, sg, nom, 3])).
word(was, pron([qu], [neut, sg, akk, 3])).
```

(e) Relativpronomen: Hier ist die Person kein Formmerkmal, da dieselbe Form für alle Personen verwendet wird (wobei in erster und zweiter Person Singular das grammatische Genus dem natürlichen angepaßt wird).

⟨*Lexika/pronomen.wrd*⟩+≡

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

word(die, pron([rel], [fem, sg, nom]), drel).
word(deren, pron([rel], [fem, sg, gen]), drel).
word(der, pron([rel], [fem, sg, dat]), drel).
word(die, pron([rel], [fem, sg, akk]), drel).

word(das, pron([rel], [neut, sg, nom]), drel).
word(dessen, pron([rel], [neut, sg, gen]), drel).
word(dem, pron([rel], [neut, sg, dat]), drel).
word(das, pron([rel], [neut, sg, akk]), drel).

word(die, pron([rel], [unsp, pl, nom]), drel).
word(deren, pron([rel], [unsp, pl, gen]), drel).
word(denen, pron([rel], [unsp, pl, dat]), drel).
word(die, pron([rel], [unsp, pl, akk]), drel).
```

Bem. Die Possessivpronomen stehen unter den Determinatoren/Adjektiven, da sie als Determintatoren und als Prädikative (`mein Buch` vs. `das Buch ist meines`) verwendet werden.

Frage: Ist es ein Kriterium für Determinatoren, daß sie in diesen beiden Funktionen verwendet werden können?

(iii) Einfache Nomen:

⟨*Wortkategorien*⟩+≡

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

(iv) Nominalisierte Verben: (`nv` ??)

⟨*Wortkategorien*⟩+≡

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

(v) Nominalisierte Adjektive: bei diesen sind die Komparation und die Adjektivdeklination eine Formeigenschaft (`das Schwierigste`, `ein Beamter`, `der Beamte`):

⟨*Wortkategorien*⟩+≡

```
wortkategorie(na([Gen|Rahmen],[Komp,Adekl,Num,Kas])) :-
        genus(Gen), numerus(Num), kasus(Kas),
        komplemente(Rahmen),
        komparation(Komp),
        adjektivdeklination(Adekl).
```

Bem. Hier werden viele unnötige Formen erzeugt: bei vielen nominalisierten Adjektiven sind keine Pluralformen erlaubt, bei manchen keine Kompartationsformen, aber das wird von der Vollformenerstellung nicht (oder nicht ausreichend) berücksichtigt.

Die Merkmalbereiche für Numerus und Person sind schon oben bei den Verben festgelegt; die für Numerus und Kasus sind:

⟨*Lexikalische Merkmale*⟩+≡
```
genus(mask).
genus(fem).
genus(neut).
genusErw(Gen) :-
        genus(Gen) ; Gen = unsp.

kasus(nom).
kasus(gen).
kasus(dat).
kasus(akk).
```

Bem: Die indefiniten Pronomina im Singular wie `man,jemand`, haben im CISLEX das Genus `unsp`, sollten aber `mask` haben: man soll *seine* Fehler nicht wiederholen!

6.2.3 Adjektive

Bei den Klassenmerkmalen der Adjektive unterscheiden wir nach dem Komplementrahmen und den syntaktischen Funktionen, die das Adjektiv übernehmen kann: welche Adjektive sind attributiv zum Nomen, welche adverbial und welche prädikativ verwendbar, welche graduierbar?

⟨*Wortkategorien*⟩+≡
```
wortkategorie(a([Rahmen,Funktionen],[Komp,Adekl,Gen,Num,Kas])) :-
        komparation(Komp), adjektivdeklination(Adekl),
        ((Num = sg, genusErw(Gen)) ; (Num = pl, Gen = unsp)),
        kasus(Kas),
        komplemente(Rahmen),
        verwendbarkeit(Funktionen).

verwendbarkeit(praedik).    % testhalber
verwendbarkeit(attrib).     % testhalber
verwendbarkeit(adverbial).  % testhalber
```

Warnung: im Lexikon steht statt `[Rahmen,Funktionen]` *nur eine leere Merkmalliste!* Man muß also Komplementrahmen noch ergänzen. (Flexionsprogramme?) Deswegen ergänzen wir:

⟨*Wortkategorien*⟩+≡
```
wortkategorie(a([Rahmen],[Komp,Adekl,Gen,Num,Kas])) :-
        komparation(Komp), adjektivdeklination(Adekl),
        genus(Gen), numerus(Num), kasus(Kas),
        komplemente(Rahmen).
wortkategorie(a([],[Komp,Adekl,Gen,Num,Kas])) :-
        komparation(Komp), adjektivdeklination(Adekl),
        genusErw(Gen), numerus(Num), kasus(Kas).
```

Frage: Sollen Ordinal- und Kardinalzahlen als Unterklassen markiert werden?

Bei den Formmerkmalen der deklinierten Adjektive unterscheiden wir neben Genus (erweitert um den Wert `unsp` wie im CISLEX), Numerus und Kasus nach der Adjektivdeklination und der Komparation. Diese haben die Werte

⟨*Lexikalische Merkmale*⟩+≡
```
  adjektivdeklination(st).
  adjektivdeklination(sw).
  adjektivdeklination(ge).

  komparation(pos).
  komparation(komp).
  komparation(sup).
```

Unflektiert verwendete Adjektive werden durch andere Lexikoneinträge dargestellt, von denen es bisher noch nicht viele gibt:

⟨*Wortkategorien*⟩+≡
```
  wortkategorie(a([],[Komp,praedik]))  :-
          komparation(Komp).
```

(Vgl. `Seminarbeitraege/LenkLeonardi/adjektive.wrd`). Das Format kann sich ändern; es sollte z.B. zumindest noch ein Komplementrahmen erlaubt sein. Vorläufig behandeln wir nur einstellige Adjektive:

⟨*Wortkategorien*⟩+≡
```
  wortkategorie(a([Kas],[Komp,praedik]))  :-
          kasus(Kas),
          komparation(Komp).
  wortkategorie(a([Praep],[Komp,praedik]))  :-
          % praeposition(Praep), % oder a([praep(Praep)],Form) ??
          komparation(Komp).
```

Frage: Welche Partikel sind Adjektivmodifikatoren (Gradpartikel)?

6.2.4 Präpositionen

Prä- und Postpositionen werden im Stammformenlexikon `wrd/2` nach ihrem Komplement und der adverbialen Funktion, die sie anzeigen können, unterklassifiziert:

⟨*Wortkategorien (unbenutzt)*⟩≡
```
  wortkategorie(pr([[Kas],AdvF],[Praep])) :-
          kasus(Kas),
          adverbialeFunktion(AdvF).
  wortkategorie(po([[Kas],AdvF],[Praep])) :-
          kasus(Kas),
          adverbialeFunktion(AdvF).
```

Solange die adverbiale Funktion in der lexikalischen Analyse weggeblendet wird, benutzen wir nur:

⟨*Wortkategorien*⟩+≡
```
  wortkategorie(pr([[Kas]],[Praep])) :-
          kasus(Kas),
          adverbialeFunktion(AdvF).
  wortkategorie(po([[Kas]],[Praep])) :-
          kasus(Kas),
          adverbialeFunktion(AdvF).
```

⟨*Lexikalische Merkmale*⟩+≡
```
adverbialeFunktion(dir).
adverbialeFunktion(final).
adverbialeFunktion(inst).
adverbialeFunktion(temp).
adverbialeFunktion(kaus).
% u.s.w.
```

Als Komplemente sind erlaubt: durch einen Kasus `Kas` angegebene Nominalphrasen, durch eine adverbiale Funktion `AdvF` angegebene Adverbiale (darunter Adverbien, wie in `bis später`, und Präpositionalphrasen), und durch eine Präposition `Praep` angegebene Präpositionalphrasen:

```
word(bis, pr([dir(hin)],[bis])).
word(bis, pr([temp],[bis])).
word(bis, pr([zu(dat)],[bis])).
```

Die Präposition selbst wird in der Kategorie als Formmerkmal gespeichert, um sie an die Präpositionalphrase vererben zu können. Dort wird der von der Präposition regierte Kasus mit angegeben, damit bei den adverbial verwendeten Präpositionalphrasen z.B. direktionale und lokale Adverbiale `in,an,auf` unterschieden werden können.

Die semantische Unterklassifizierung der Präpositionen erhöht die (scheinbare) syntaktische Mehrdeutigkeit sehr stark. Um zunächst nicht zwischen der möglichen Verwendung einer Präpositionalphrase als Adverbial oder als Komplement unterscheiden zu müssen, projizieren wir das Merkmal `AdvF` bei der lexikalischen Analyse weg: wir leiten Einträge ins Vollformenlexikon `word/2` aus Einträgen eines Stammformenlexikons `wrd/2` ab, die semantische Information enthalten (und das Komplement nicht als Liste verpacken):

⟨*Lexika/praepositionen.wrd*⟩≡
```
word(Praep,Cat) :-
        wrd(Praep, pr([Kas,_AdvF],Form)),
        Cat = pr([[Kas]],Form).
word(Praep,Cat) :-
        wrd(Praep, po([Kas,_AdvF],Form)),
        Cat = po([[Kas]],Form).
```

Erst beim Einbau einer Präpositionalphrase in einen Satz wird überprüft, ob sie eine bestimmte adverbiale Funktion erfüllen kann. Da die Präposition (mit dem regierten Kasus) im Formmerkmal der Präpositionalphrase steht, kann die semantische Information (mit Hilfe von `wrd(für,...)`) rekonstruiert werden kann.

(i) Alphabetische Liste der Präpositionen, klassifiziert nach dem regierenden Kasus und den adverbialen Funktionen (nach Helbig/Buscha: Deutsche Grammatik), die eine mit der Präposition eingeleitete Phrase haben kann:

⟨*Lexika/praepositionen.wrd*⟩+≡
```
wrd(ab,        pr([dat,lok],[ab])).
wrd(ab,        pr([dat,temp],[ab])).
wrd(abseits,   pr([gen,lok],[abseits])).
wrd(abzüglich, pr([gen,modl],[abzüglich])).
wrd(als,       pr([Kas,modl],[als(Kas)])) :-    kasus(Kas). % du/dich als Spezialist/en
wrd(an,        pr([dat,lok],[an(dat)])).
wrd(an,        pr([dat,temp],[an(dat)])).
```

```
wrd(an,        pr([akk,dir(hin)],[an(akk)])).
wrd(angesichts,pr([gen,kaus],[angesichts])).
% anhand
% à
wrd(anläßlich, pr([gen,kaus],[anläßlich])).
wrd(anstatt,   pr([gen,modl],[anstatt])).
% anstelle
wrd(auf,       pr([dat,lok],[auf(dat)])).
wrd(auf,       pr([dat,temp],[auf(dat)])).  % auf dem Ereignis
wrd(auf,       pr([akk,dir(hin)],[auf(akk)])).
wrd(auf,       pr([akk,temp],[auf(akk)])).  % auf (die) Dauer
wrd(auf,       pr([akk,kaus],[auf(akk)])).  % auf Empfehlung (hin)
wrd(auf,       pr([akk,modl],[auf(akk)])).  % (bis) auf die Sekunde
% aufgrund
wrd(aus,       pr([dat,dir(her)],[aus])).
wrd(aus,       pr([dat,kaus],[aus])).
wrd(aus,       pr([dat,modl],[aus])).
wrd(ausschließlich, pr([gen,kop],[ausschließlich])).
wrd(außer,     pr([dat,modl],[außer])).
wrd(außer,     pr([dat,kop],[außer])).
wrd(außerhalb, pr([gen,lok],[außerhalb])).
wrd(außerhalb, pr([gen,temp],[außerhalb])).

% behufs
wrd(bei,       pr([dat,lok],[bei])).
wrd(bei,       pr([dat,temp],[bei])).
wrd(bei,       pr([dat,kond],[bei])).
% betreffs
wrd(bezüglich, pr([gen,kaus],[bezüglich])).
wrd(binnen,    pr([dat,temp],[binnen])).
% wrd(bis,     pr([akk],[bis])).    % dir: mit 2.Präp s.u. ; bis[adv(temp)]

wrd(dank,      pr([dat,kaus],[dank])).  % inst ?
wrd(diesseits, pr([gen,lok],[diesseits])).
wrd(durch,     pr([akk,lok],[durch])).
wrd(durch,     pr([akk,dir(hin)],[durch])).

wrd(einschließlich, pr([gen,kop],[einschließlich])).
wrd(entlang,   pr([gen,lok],[entlang])).
wrd(entgegen,  pr([dat,modl],[entgegen])).
wrd(exklusive, pr([gen,kop],[exklusive])).

wrd(fern,      pr([gen,lok],[fern])).
wrd(für,       pr([akk,final],[für])).
wrd(für,       pr([akk,modl],[für])).

wrd(gegen,     pr([akk,dir(hin)],[gegen])).
wrd(gegen,     pr([akk,modl],[gegen])).
wrd(gegenüber, pr([dat,lok],[gegenüber])).
wrd(gegenüber, pr([dat,modl],[gegenüber])).
wrd(gemäß,     pr([dat,modl],[gemäß])).
wrd(gemäß,     pr([dat,kaus],[gemäß])).
% gen
```

```
wrd(hinter,    pr([dat,lok],[hinter(dat)])).
wrd(hinter,    pr([akk,dir(hin)],[hinter(akk)])).

wrd(in,        pr([dat,lok],[in(dat)])).
wrd(in,        pr([dat,temp],[in(dat)])).
wrd(in,        pr([dat,modl],[in(dat)])).
wrd(in,        pr([akk,dir(hin)],[in(akk)])).
wrd(infolge,   pr([gen,kaus],[infolge])).
wrd(inklusive, pr([gen,kop],[inklusive])).
wrd(inmitten,  pr([gen,lok],[inmitten])).
wrd(innerhalb, pr([gen,lok],[innerhalb])).
wrd(innerhalb, pr([gen,temp],[innerhalb])).
wrd(je,        pr([nom,dis],[je(nom)])).         % Nullartikel!
wrd(je,        pr([akk,dis],[je(akk)])).
wrd(jenseits,  pr([gen,lok],[jenseits])).

wrd(kraft,     pr([gen,kaus],[kraft])).
wrd(längs,     pr([gen,lok],[längs])).
wrd(laut,      pr([gen,modl],[laut(gen)])).
wrd(laut,      pr([dat,modl],[laut(dat)])).

wrd(mangels,   pr([gen,kaus],[mangels])).
wrd(mit,       pr([dat,inst],[mit])).
wrd(mit,       pr([dat,kop],[mit])).
wrd(mit,       pr([dat,modl],[mit])).
wrd(mit,       pr([dat,temp],[mit])).
wrd(mit,       pr([dat,kond],[mit])).
wrd(mitsamt,   pr([dat,kop],[mitsamt])).
wrd(mittels,   pr([gen,inst],[mittels])).

wrd(nahe,      pr([dat,lok],[nahe])).
wrd(nach,      pr([dat,dir(hin)],[nach])).
wrd(nach,      pr([dat,temp],[nach])).
wrd(nach,      pr([dat,modl],[nach])).
wrd(neben,     pr([dat,lok],[neben])).
wrd(neben,     pr([dat,kop],[neben(dat)])).
wrd(neben,     pr([akk,dir(hin)],[neben(akk)])).
% nebst
% nächst
% namens
wrd(nördlich,  pr([gen,lok],[nördlich])).

% ob
wrd(oberhalb,pr([gen,lok],[oberhalb])).
wrd(ohne,      pr([akk,inst],[ohne])).
wrd(ohne,      pr([akk,kop],[ohne])).
wrd(ohne,      pr([akk,modl],[ohne])).

wrd(per,       pr([akk,dis],[per])).    % Nullartikel, nur Singular!
wrd(pro,       pr([akk,dis],[pro])).    %    "

% qua
```

```
wrd(samt,      pr([dat,kop],[samt])).
wrd(seit,      pr([dat,temp],[seit])).
wrd(seitens,   pr([gen,kaus],[seitens])).  % aber: meiner-seits
wrd(seitlich,  pr([gen,lok],[seitlich])).
% seitwärts
% sonder
wrd(statt,     pr([gen,modl],[statt])).
wrd(südlich,   pr([gen,lok],[südlich])).

wrd(trotz,     pr([gen,konz],[trotz])).  % dieselbe Praep mit 2 Kausus
wrd(trotz,     pr([dat,konz],[trotz])).

wrd(über,      pr([dat,lok],[über(dat)])).
wrd(über,      pr([akk,dir(hin)],[über(akk)])).
wrd(um,        pr([akk,lok],[um])).
wrd(um,        pr([akk,temp],[um])).
% unbeschadet
wrd(unfern,    pr([gen,lok],[unfern])). % ??
wrd(ungeachtet,pr([gen,konz],[ungeachtet])).
wrd(unter,     pr([dat,lok],[unter(dat)])).
wrd(unter,     pr([akk,dir(hin)],[unter(akk)])).
wrd(unterhalb, pr([gen,lok],[unterhalb])).
wrd(unweit,    pr([gen,lok],[unweit])).

wrd(vermittels,pr([gen,inst],[vermittels])).
wrd(vermöge,   pr([gen,inst],[vermöge])).
% vis-a-vis
wrd(von,       pr([dat,dir(her)],[von])).
wrd(von,       pr([dat,modl],[von])).
wrd(vor,       pr([dat,lok],[vor(dat)])).
wrd(vor,       pr([dat,temp],[vor(dat)])).
wrd(vor,       pr([dat,kaus],[vor(dat)])).
wrd(vor,       pr([akk,dir(hin)],[vor(akk)])).

wrd(während,   pr([gen,temp],[während])).
wrd(wegen,     pr([gen,kaus],[wegen])).  % Aber: seinetwegen - wegen ihm (!)
wrd(westlich,  pr([gen,lok],[westlich])).
wrd(wider,     pr([akk,konz],[wider])).
wrd(wie,       pr([Kas,modl],[wie(Kas)])) :- kasus(Kas). % = (so)wie dich/dir ..

wrd(zeit,      pr([gen,temp],[zeit])).
wrd(zu,        pr([dat,dir(hin)],[zu])).
% wrd(zu,        pr([dat,final],[zu])).  % zum Andenken, zur Erinnerung
wrd(zu,        pr([dat,dist],[zu])).     % zu dritt
% wrd(zu,        pr([dat,modl],[zu])).   % mir zur Freude
wrd(zugunsten, pr([gen,final],[zugunsten])).
wrd(zuzüglich, pr([dat,modl],[zuzüglich])). % gen?
wrd(zwecks,    pr([gen,final],[zwecks])).
wrd(zwischen,  pr([dat,lok],[zwischen(dat)])).
wrd(zwischen,  pr([akk,dir(hin)],[zwischen(akk)])).
```

(ii) Pronominaladverbien (da|hier|wo + Präposition):[14] Sie werden als Präpositional*phrasen* kategorisiert, (aber nicht über eine Kettenregel erzeugt und deshalb) ins Vollformenlexikon der Präpositionen geschrieben. Im Formmerkmal ist nur dann auch der Kasus der Präposition angegeben, wenn er nicht eindeutig bestimmt ist.

⟨*Lexika/praepositionen.wrd*⟩+≡

```
word(darauf,  pp([def],[auf(dat)])).
word(darauf,  pp([def],[auf(akk)])).
word(hierauf, pp([def],[auf(dat)])).
word(hierauf, pp([def],[auf(akk)])).
word(worauf,  pp([qu(3, unsp,sg)],[auf(dat)])).
word(worauf,  pp([qu(3, unsp,sg)],[auf(akk)])).
word(worauf,  pp([rel(3,neut,sg)],[auf(dat)])).  % nur neut?
word(worauf,  pp([rel(3,neut,sg)],[auf(akk)])).

word(daran,   pp([def],[an(dat)])).
word(daran,   pp([def],[an(akk)])).
word(hieran,  pp([def],[an(dat)])).
word(hieran,  pp([def],[an(akk)])).
word(woran,   pp([qu(3,unsp,sg)],[an(dat)])).    % nur 3?
word(woran,   pp([qu(3,unsp,sg)],[an(akk)])).
word(woran,   pp([rel(3,neut,sg)],[an(dat)])).
word(woran,   pp([rel(3,neut,sg)],[an(akk)])).

word(daraus,  pp([def],[aus])).
word(hieraus, pp([def],[aus])).
word(woraus,  pp([qu(3,unsp,sg)],[aus])).
word(woraus,  pp([rel(3,neut,sg)],[aus])).

word(dabei,   pp([def],[bei])).
word(hierbei, pp([def],[bei])).
word(wobei,   pp([qu(3,unsp,sg)],[bei])).
word(wobei,   pp([rel(3,neut,sg)],[bei])).

word(dadurch, pp([def],[durch])).
word(hierdurch, pp([def],[durch])).
word(wodurch, pp([qu(3,unsp,sg)],[durch])).
word(wodurch, pp([rel(3,neut,sg)],[durch])).

word(dafür,   pp([def],[für])).
word(hierfür, pp([def],[für])).
word(wofür,   pp([qu(3,unsp,sg)],[für])).
word(wofür,   pp([rel(3,neut,sg)],[für])).

word(darin,   pp([def],[in(dat)])).
word(hierin,  pp([def],[in(dat)])).
word(worin,   pp([qu ],[in(dat)])).
word(worin,   pp([rel(neut)],[in(dat)])).

word(darein,  pp([def],[in(akk)])).
word(hierein, pp([def],[in(akk)])).
word(worein,  pp([qu ],[in(akk)])).
```

[14] Helbig/Buscha: auf, an, aus, bei, für, mit, nach, vor. Dazu: durch, in(dat), (e)in(akk), über, unter, zu

```
word(worein,  pp([rel(neut)],[in(akk)])).

word(damit,   pp([def],[mit])).
word(hiermit, pp([def],[mit])).
word(womit,   pp([qu ],[mit])).
word(womit,   pp([rel(neut)],[mit])).

word(danach,  pp([def],[nach])).
word(wonach,  pp([qu ],[nach])).
word(wonach,  pp([rel(neut)],[nach])).

word(darüber, pp([def],[über(dat)])).
word(darüber, pp([def],[über(akk)])).
word(hierüber,pp([def],[über(dat)])).
word(hierüber,pp([def],[über(akk)])).
word(worüber, pp([qu ],[über(dat)])).
word(worüber, pp([qu ],[über(akk)])).
word(worüber, pp([rel(neut)],[über(dat)])).
word(worüber, pp([rel(neut)],[über(akk)])).

word(darunter, pp([def],[unter(dat)])).
word(darunter, pp([def],[unter(akk)])).  % dir
word(hierunter,pp([def],[unter(dat)])).
word(worunter, pp([qu ],[unter(dat)])).
word(worunter, pp([rel(neut)],[unter(dat)])).

word(davor,   pp([def],[vor(dat)])).
word(davor,   pp([def],[vor(akk)])).     % stelle den Eimer davor
word(hiervor, pp([def],[vor(dat)])).
word(hiervor, pp([def],[vor(akk)])).
word(wovor,   pp([qu ],[vor(dat)])).
word(wovor,   pp([qu(3,unsp,sg)],[vor(dat)])).
word(wovor,   pp([qu ],[vor(akk)])).
word(wovor,   pp([rel(neut)],[vor(dat)])).

word(dazu,    pp([def],[zu])).
word(hierzu,  pp([def],[zu])).
word(wozu,    pp([qu ],[zu])).
word(wozu,    pp([rel(neut)],[zu])).
```

(iii) Konjunktionaladverbien: (Satzglied | satzkoordinierende Konjunktion): [15]

⟨*Lexika/praepositionen.wrd*⟩+≡

```
word(deswegen, pp([def],[wegen(gen)])).  % auch als adv([def,kaus]). Unnötig.
word(weswegen, pp([qu ],[wegen(gen)])).
word(weswegen, pp([rel(kaus)],[wegen(gen)])).  % das liefert auch die rel-Lesart

word(dorthin, pp([def],   [in(akk)])).         % oder als Direktionaladverbien ??
word(wohin,   pp([qu ],   [in(akk)])).
word(wohin,   pp([rel(dir(hin))],[in(akk)])).  % der Ort, wohin ich fahre, ...

% gehen :v([nom,dir(akk)],[inf]) auf, zu,
```

[15] Helbig/Buscha: deshalb, daher, trotzdem, folglich, nämlich, insofern, deswegen, mithin, demnach, sonst, außerdem, allerdings.

```
% kommen:v([nom,dir(dat)],[inf]) aus,von:dat, in:akk
```

(iv) Abtrennbare Verbpräfixe (CISLEX: VPART), soweit sie auch Präpositionen sind:

⟨*Lexika/praepositionen.wrd*⟩+≡

```
word(ab,      vpr(ab)).
word(an,      vpr(an)).
word(auf,     vpr(auf)).
word(aus,     vpr(aus)).
word(dar,     vpr(dar)).
word(ein,     vpr(ein)).
word(hervor,  vpr(hervor)).
word(nach,    vpr(nach)).
word(statt,   vpr(statt)).     % stattfinden
word(über,    vpr(über)).
word(um,      vpr(um)).
word(vor,     vpr(vor)).
word(weg,     vpr(weg)).
word(zu,      vpr(zu)).
```

(v) Präpositionen, die auch satzkomplementierende Elemente sind: *Kategorie??*

⟨*Lexika/praepositionen.wrd*⟩+≡

```
word(ob,      ob).
word(um,      um).
word(zu,      zu).
```

(vi) Postpositionen:

⟨*Lexika/praepositionen.wrd*⟩+≡

```
wrd(entgegen,  po([dat,modl],[entgegen])).  % konz ?
wrd(entlang,   po([akk,dir(_)],[entlang])).
wrd(gegenüber, po([dat,lok],[gegenüber])).
wrd(gegenüber, po([dat,modl],[gegenüber])).
wrd(halber,    po([gen,kaus],[halber])).
wrd(nach,      po([dat,modl],[nach])).
wrd(über,      po([akk,temp],[über])).
% wrd(ungeachtet,po([gen,konz],[ungeachtet])).
wrd(wegen,     po([gen,kaus],[wegen])).     % meinet-wegen
wrd(zufolge,   po([dat,kaus],[zufolge])).
wrd(zufolge,   po([dat,modl],[zufolge])).
wrd(zuliebe,   po([dat,kaus],[zuliebe])).
% wrd(zuwider,   p([dat,modl],[zuwider])).  % adv, vpr
```

(vii) Präpositionen mit einem Adverbialargument: *erst testbar, wenn man Adverbialphrasen hat unfertig; funktioniert nicht!* (Beachte: Bei diesen `word/2`-Einträgen wird in den Kategorien die adverbiale Funktion als Komplement*liste* angegeben.)

⟨*Lexika/praepositionen.wrd*⟩+≡

```
% bis - an|auf|gegen|hinter|in|nach|unter|über|vor|zu

word(bis,            pr([[zu(dat)]],[bis])).
word(bis,            pr([[dir(hin)]],[bis])).
word(bis,            pr([[temp]],[bis])).
word(bis,            pr([[kop]],[bis])).  % bis auf NP
```

```
word(bis,           pr([[auf(akk)]],[bis])).

word(von,           pr([[dir(her)]],[von])).  % von dort her; von X aus?
word(seit,          pr([[temp]],[seit])).

% je-nach, zu-je, unweit-von
```

(viii) Zweiteilige Präpositionen (Circumpositionen): Aus einem 'Stamm'-Eintrag einer zweiteiligen Präposition werden für die beiden Teile getrennte Einträge ins Vollformenlexikon erzeugt:

⟨*Lexika/praepositionen.wrd*⟩+≡

```
word(Prae,          pc1([[Kas]],[Praep])) :-
        Praep = (Prae,Post),
        wrd(Praep, pc([Kas],[Praep])).
word(Post,          pc2([[Kas]],[Praep])) :-
        Praep = (Prae,Post),
        wrd(Praep, pc([Kas],[Praep])).

wrd((um,'Willen'),  pc([gen],[(um,'Willen')])).
wrd((an,'Statt'),   pc([gen],[(an,'Statt')])).

wrd((auf,hinauf),   pc([akk],[(auf,hinauf)])).
wrd((in,hinein),    pc([akk],[(in,hinein)])).
wrd((um,herum),     pc([akk],[(um,herum)])).
wrd((über,hinaus),  pc([akk],[(über,hinaus)])).

wrd((von,an),       pc([dat],[(von,an)])).
wrd((von,aus),      pc([dat],[(von,aus)])).
```

Bem. Im jeweiligen `wrd/2`-Stammeintrag fehlt noch die adverbiale Funktion, und das Komplement ist einfacheitshalber nicht als Liste notiert.

(ix) Präpositionen, die mit einem bestimmten Artikel verschmolzen sind: als Klassenparameter braucht man neben dem Kasus auch das Genus der möglichen Objekte; die adverbiale Funktion, die die Phrase realisieren kann, tragen wir nur im 'Stamm'-Eintrag ein:

⟨*Lexika/praepositionen.wrd*⟩+≡

```
word(PraepDet, pdet([[Kas],Gen],[Praep])) :-
        wrd(PraepDet, pdet([[Kas],_AdvF,Gen],[Praep])).

wrd(am,   pdet([[dat],lok,mask],[an(dat)])).
wrd(am,   pdet([[dat],lok,neut],[an(dat)])).
wrd(ans,  pdet([[akk],lok,neut],[an(akk)])).
wrd(aufs, pdet([[akk],dir,neut],[auf(akk)])).
wrd(beim, pdet([[dat],lok,mask],[bei])).
wrd(beim, pdet([[dat],lok,neut],[bei])).
wrd(fürs, pdet([[akk],modl,neut],[für])). % modl?
wrd(im,   pdet([[dat],lok,mask],[in(dat)])).
wrd(im,   pdet([[dat],lok,neut],[in(dat)])).
wrd(ins,  pdet([[akk],dir,neut],[in(akk)])).
wrd(vom,  pdet([[dat],dir,mask],[von])).
wrd(vom,  pdet([[dat],dir,neut],[von])).
wrd(zum,  pdet([[dat],dir,mask],[zu])).
wrd(zum,  pdet([[dat],dir,neut],[zu])).
wrd(zur,  pdet([[dat],dir,fem ],[zu])).
```

Bem.: Die Abkürzung des Komplements durch den Kasus ist hier ungenau: die Grammatikregeln müssen dafür sorgen, daß nicht beliebige Nominalphrasen, sondern nur artikellose Nominalausdrücke im Singular als Komplemente verwendet werden können.

Ein Hilfslexikon für eingebettete Nominalphrasen:

(i) Als Artikel nehmen wir alle Formen des bestimmten Artikels im Maskulinum:

⟨*Lexika/praepositionen.hilfslexikon.wrd*⟩≡

```
word(der,det([def,sw],[mask,sg,nom])).
word(des,det([def,sw],[mask,sg,gen])).
word(dem,det([def,sw],[mask,sg,dat])).
word(den,det([def,sw],[mask,sg,akk])).

word(die,det([def,sw],[mask,pl,nom])).
word(der,det([def,sw],[mask,pl,gen])).
word(den,det([def,sw],[mask,pl,dat])).
word(die,det([def,sw],[mask,pl,akk])).
```

Zum Testen seien noch ein paar andere Determinatoren im Lexikon:

⟨*Lexika/praepositionen.hilfslexikon.wrd*⟩+≡

```
word(ein,det([indef,ge],[mask,sg,nom])).
word(eines,det([indef,ge],[mask,sg,gen])).
word(einem,det([indef,ge],[mask,sg,dat])).
word(einen,det([indef,ge],[mask,sg,akk])).

word(welcher,det([qu],[mask,sg,nom])).
word(welchem,det([qu],[mask,sg,dat])).

word(eine,det([indef,ge],[fem,sg,nom])).
word(einer,det([indef,ge],[fem,sg,gen])).
word(einer,det([indef,ge],[fem,sg,dat])).
word(eine,det([indef,ge],[fem,sg,akk])).
```

(ii) Als Nomen nehmen wir nur ein Wort in allen Fällen:

⟨*Lexika/praepositionen.hilfslexikon.wrd*⟩+≡

```
word('Hamster',n([mask],[sg,nom])).
word('Hamsters',n([mask],[sg,gen])).
word('Hamster',n([mask],[sg,dat])).
word('Hamster',n([mask],[sg,akk])).

word('Hamster',n([mask],[pl,nom])).
word('Hamster',n([mask],[pl,gen])).
word('Hamstern',n([mask],[pl,dat])).
word('Hamster',n([mask],[pl,akk])).
```

(iii) Um relativierende und interrogative Präpositionalphrasen testen zu können, geben wir zwei Pseudo-Nominalphrasen vor:

⟨*Lexika/praepositionen.hilfslexikon.wrd*⟩+≡

```
word(dem, np([rel,3,mask],[sg,dat])).
word(wem, np([qu,3,mask],[sg,dat])).
```

(iv) Um Päpositionalphrasen, die eine Adverbialphrase enthalten, testen zu können, seien noch zwei Adverbien angegeben:

⟨*Lexika/praepositionen.hilfslexikon.wrd*⟩+≡
```
word(dann, adv([def,temp],[])).
word(wann, adv([qu,temp],[])).
```

(v) Um die Konjunktion von Päpositional- (und eingebetteten Nominal-) phrasen testen zu können, seien noch zwei Konjunktionen angegeben:

⟨*Lexika/praepositionen.hilfslexikon.wrd*⟩+≡
```
word(und, conj([[np,np]],[])).
word(oder, conj([[pp,pp]],[])).
```

Aufgabe 6.1 Berechne aus einer Grammatik mit zweistelligen Prädikaten `Kategorie(Klasse,Form)` die in der Grammatik verwendeten lexikalischen Kategorien und ihre Merkmalwerte. Mache das gleiche für die in einem Vollformenlexikon `word(Vollform,Kategorie)` (oder in einem Teillexikon `lexword/2` verwendeten Kategorien.

Erzeuge damit ein Testprogramm, das überprüft, ob eine Grammatik und ein (Teil-) Lexikon zueinander passen, d.h. ob im Lexikon nur Wortkategorien verwendet werden, die auch in der Grammatik auftreten.

Ein Parser sollte mit einem solchen Programm prüfen, ob die verwendete Grammatik alle von der lexikalischen Analyse genannten lexikalischen Kategorien kennt; wenn nicht, sollte der Parser gar keine Analyse versuchen, sondern eine Fehlermeldung ausgeben.

6.2.5 Determinatoren

Die Determinatoren bilden eigentlich (wie die Adverbiale) eine Sammelkategorie, sind also nicht aus einer Wortart durch Erweiterungen gebildete Ausdrücke. Es sind vielmehr Ausdrücke verschiedener Wortarten, die „nur“ deshalb zusammengefaßt werden, weil sie *auch* die syntaktische Funktion eines Determinators einer Nominalphrase übernehmen können.

Aus diesem Grund ist es nicht verwunderlich, daß nicht alle Wörter darin in genau denselben syntaktischen Funktionen auftreten. (In manchen Fällen erfolgt die Flexion eines Worts in Abhängigkeit von der Verwendung als Determinator oder Pronomen). Üblicherweise unterscheidet man z.B. (grob, mit definiten, interrogativen, relativen Versionen) die Wortarten:

(i) Demonstrativum: `der, dieser, jener, welcher, ...`

(ii) Possessivum: `mein, sein, eure, jemandes, niemandes, dessen, wessen, ..`

(iii) Quantoren: `viele, wenige, alle, wie viele, wieviel, jeder, kein,...`

Wir beschreiben die Unterklassen von Determinatoren aber wie bei den eigentlichen Wortklassen durch Klassenparameter, nicht durch eigene Kategoriennamen.

⟨*Wortkategorien*⟩+≡
```
wortkategorie(det([Def,Adekl],[Gen,Num,Kas])) :-
        definitheit_det(Def),
        adjektivdeklination(Adekl),
        genusErw(Gen),
        numerus(Num),
        kasus(Kas).
```

Als Unterklassen unterscheiden wir in definite, indefinite, possessive, quantifizierende, relativierende und interrogative Determinatoren, sowie danach, welche Adjektivdeklination der Determinator regiert:

⟨*Lexikalische Merkmale*⟩+≡

```
definitheit_det(def).
definitheit_det(indef).
definitheit_det(poss).
definitheit_det(quant).
definitheit_det(rel).
definitheit_det(qu).
```

Bei den Formmerkmalen sind Genus, Numerus und Kasus zu berücksichtigen.

Die folgende Datei von Determinatoren aus dem Beispieltext `Testbett/wissenschaftsreligion.txt` enthält Einträge der Form `word(Determiner,Kategorie,Stamm)`, worin der Stamm für die Flexion gedacht ist. (Die Datei enthält keine Lesarten als Relativpronomen.)

⟨*Lexika/determiner.wrd*⟩≡

```
word(alle, det([quant,sw], [unsp, pl, nom]), alle).
word(alle, det([quant,sw], [unsp, pl, akk]), alle).
word(alle, det([quant,sw], [fem, sg, akk]), alle).
word(alle, det([quant,sw], [fem, sg, nom]), alle).
word(allem, det([quant,sw], [mask, sg, dat]), alle).
word(allem, det([quant,sw], [neut, sg, dat]), alle).
word(allen, det([quant,sw], [unsp, pl, dat]), alle).
word(allen, det([quant,sw], [mask, sg, gen]), alle).
word(allen, det([quant,sw], [neut, sg, gen]), alle).
word(allen, det([quant,sw], [mask, sg, akk]), alle).
word(aller, det([quant,sw], [unsp, pl, gen]), alle).
word(aller, det([quant,sw], [mask, sg, nom]), alle).
word(aller, det([quant,sw], [fem, sg, gen]), alle).
word(aller, det([quant,sw], [fem, sg, dat]), alle).
word(alles, det([quant,sw], [neut, sg, nom]), alle).
word(alles, det([quant,sw], [neut, sg, akk]), alle).
word(beide, det([quant,st], [unsp, pl, akk]), beid).
word(beide, det([quant,st], [unsp, pl, nom]), beid).
word(beiden, det([quant,st], [unsp, pl, dat]), beid).
word(beider, det([quant,st], [unsp, pl, gen]), beid).
word(das, det([def,sw], [neut, sg, nom]), d).
word(das, det([def,sw], [neut, sg, akk]), d).
word(dasjenige, det([def,sw], [neut, sg, nom]), d_jenig).
word(dasjenige, det([def,sw], [neut, sg, akk]), d_jenig).
word(dasselbe, det([def,sw], [neut, sg, nom]), d_selb).
word(dasselbe, det([def,sw], [neut, sg, akk]), d_selb).
word(dem, det([def,sw], [mask, sg, dat]), d).
word(dem, det([def,sw], [neut, sg, dat]), d).
word(demjenigen, det([def,sw], [mask, sg, dat]), d_jenig).
word(demjenigen, det([def,sw], [neut, sg, dat]), d_jenig).
word(demselben, det([def,sw], [mask, sg, dat]), d_selb).
word(demselben, det([def,sw], [neut, sg, dat]), d_selb).
word(den, det([def,sw], [unsp, pl, dat]), d).
word(den, det([def,sw], [mask, sg, akk]), d).
word(denjenigen, det([def,sw], [unsp, pl, dat]), d_jenig).
word(denjenigen, det([def,sw], [mask, sg, akk]), d_jenig).
```

```
word(denselben, det([def,sw], [unsp, pl, dat]), d_selb).
word(denselben, det([def,sw], [mask, sg, akk]), d_selb).
word(der, det([def,sw], [unsp, pl, gen]), d).
word(der, det([def,sw], [mask, sg, nom]), d).
word(der, det([def,sw], [fem, sg, gen]), d).
word(der, det([def,sw], [fem, sg, dat]), d).
word(deren, det([poss,sw], [unsp, pl, gen]), d).
word(deren, det([poss,sw], [fem, sg, gen]), d).
word(derjenige, det([def,sw], [mask, sg, nom]), d_jenig).
word(derjenigen, det([def,sw], [unsp, pl, gen]), d_jenig).
word(derjenigen, det([def,sw], [fem, sg, gen]), d_jenig).
word(derjenigen, det([def,sw], [fem, sg, dat]), d_jenig).
word(derselbe, det([def,sw], [mask, sg, nom]), d_selb).
word(derselben, det([def,sw], [unsp, pl, gen]), d_selb).
word(derselben, det([def,sw], [fem, sg, gen]), d_selb).
word(derselben, det([def,sw], [fem, sg, dat]), d_selb).
word(des, det([def,sw], [mask, sg, gen]), d).
word(des, det([def,sw], [neut, sg, gen]), d).
word(desjenigen, det([def,sw], [mask, sg, gen]), d_jenig).
word(desjenigen, det([def,sw], [neut, sg, gen]), d_jenig).
word(desselben, det([def,sw], [mask, sg, gen]), d_selb).
word(desselben, det([def,sw], [neut, sg, gen]), d_selb).
word(dessen, det([rel,sw], [mask, sg, gen]), d).
word(dessen, det([rel,sw], [neut, sg, gen]), d).
word(die, det([def,sw], [unsp, pl, nom]), d).
word(die, det([def,sw], [unsp, pl, akk]), d).
word(die, det([def,sw], [fem, sg, akk]), d).
word(die, det([def,sw], [fem, sg, nom]), d).
word(diejenige, det([def,sw], [fem, sg, akk]), d_jenig).
word(diejenige, det([def,sw], [fem, sg, nom]), d_jenig).
word(diejenigen, det([def,sw], [unsp, pl, nom]), d_jenig).
word(diejenigen, det([def,sw], [unsp, pl, akk]), d_jenig).
word(diejenigen, det([def,sw], [unsp, pl, dat]), d_jenig).
word(diese, det([def,sw], [unsp, pl, nom]), dies).
word(diese, det([def,sw], [unsp, pl, akk]), dies).
word(diese, det([def,sw], [fem, sg, akk]), dies).
word(diese, det([def,sw], [fem, sg, nom]), dies).
word(dieselbe, det([def,sw], [fem, sg, akk]), d_selb).
word(dieselbe, det([def,sw], [fem, sg, nom]), d_selb).
word(dieselben, det([def,sw], [unsp, pl, nom]), d_selb).
word(dieselben, det([def,sw], [unsp, pl, akk]), d_selb).
word(dieselben, det([def,sw], [unsp, pl, dat]), d_selb).
word(diesem, det([def,sw], [mask, sg, dat]), dies).
word(diesem, det([def,sw], [neut, sg, dat]), dies).
word(diesen, det([def,sw], [unsp, pl, dat]), dies).
word(diesen, det([def,sw], [mask, sg, akk]), dies).
word(dieser, det([def,sw], [unsp, pl, gen]), dies).
word(dieser, det([def,sw], [mask, sg, nom]), dies).
word(dieser, det([def,sw], [fem, sg, gen]), dies).
word(dieser, det([def,sw], [fem, sg, dat]), dies).
word(dieses, det([def,sw], [neut, sg, nom]), dies).
word(dieses, det([def,sw], [neut, sg, akk]), dies).
word(dieses, det([def,sw], [mask, sg, gen]), dies).
```

```
word(dieses, det([def,sw], [neut, sg, gen]), dies).
word(ein, det([indef,ge], [neut, sg, nom]), ein).
word(ein, det([indef,ge], [neut, sg, akk]), ein).
word(ein, det([indef,ge], [mask, sg, nom]), ein).
word(eine, det([indef,ge], [fem, sg, akk]), ein).
word(eine, det([indef,ge], [fem, sg, nom]), ein).
word(einem, det([indef,ge], [mask, sg, dat]), ein).
word(einem, det([indef,ge], [neut, sg, dat]), ein).
word(einen, det([indef,ge], [mask, sg, akk]), ein).
word(einer, det([indef,ge], [fem, sg, gen]), ein).
word(einer, det([indef,ge], [fem, sg, dat]), ein).
word(eines, det([indef,ge], [neut, sg, gen]), ein).
word(eines, det([indef,ge], [mask, sg, gen]), ein).
word(einige, det([indef,st], [unsp, pl, nom]), einig).
word(einige, det([indef,st], [unsp, pl, akk]), einig).
word(einige, det([indef,st], [fem, sg, akk]), einig).
word(einige, det([indef,st], [fem, sg, nom]), einig).
word(einigem, det([indef,st], [mask, sg, dat]), einig).
word(einigem, det([indef,st], [neut, sg, dat]), einig).
word(einigen, det([indef,st], [unsp, pl, dat]), einig).
word(einigen, det([indef,st], [mask, sg, gen]), einig).
word(einigen, det([indef,st], [neut, sg, gen]), einig).
word(einigen, det([indef,st], [mask, sg, akk]), einig).
word(einiger, det([indef,st], [unsp, pl, gen]), einig).
word(einiger, det([indef,st], [mask, sg, nom]), einig).
word(einiger, det([indef,st], [fem, sg, gen]), einig).
word(einiger, det([indef,st], [fem, sg, dat]), einig).
word(einiges, det([indef,st], [neut, sg, nom]), einig).
word(einiges, det([indef,st], [neut, sg, akk]), einig).
word(ihr, det([poss,ge], [neut, sg, nom]), ihr).
word(ihr, det([poss,ge], [neut, sg, akk]), ihr).
word(ihr, det([poss,ge], [mask, sg, nom]), ihr).
word(ihre, det([poss,ge], [unsp, pl, akk]), ihr).
word(ihre, det([poss,ge], [unsp, pl, nom]), ihr).
word(ihre, det([poss,ge], [fem, sg, akk]), ihr).
word(ihre, det([poss,ge], [fem, sg, nom]), ihr).
word(ihrem, det([poss,ge], [mask, sg, dat]), ihr).
word(ihrem, det([poss,ge], [neut, sg, dat]), ihr).
word(ihren, det([poss,ge], [unsp, pl, dat]), ihr).
word(ihren, det([poss,ge], [mask, sg, akk]), ihr).
word(ihrer, det([poss,ge], [unsp, pl, gen]), ihr).
word(ihrer, det([poss,ge], [fem, sg, gen]), ihr).
word(ihrer, det([poss,ge], [fem, sg, dat]), ihr).
word(ihres, det([poss,ge], [neut, sg, gen]), ihr).
word(ihres, det([poss,ge], [mask, sg, gen]), ihr).
word(jede, det([quant,sw], [fem, sg, akk]), jed).
word(jede, det([quant,sw], [fem, sg, nom]), jed).
word(jedem, det([quant,sw], [mask, sg, dat]), jed).
word(jedem, det([quant,sw], [neut, sg, dat]), jed).
word(jeden, det([quant,sw], [mask, sg, akk]), jed).
word(jeder, det([quant,sw], [mask, sg, nom]), jed).
word(jeder, det([quant,sw], [fem, sg, gen]), jed).
word(jeder, det([quant,sw], [fem, sg, dat]), jed).
```

```
word(jedes, det([quant,sw], [neut, sg, nom]), jed).
word(jedes, det([quant,sw], [neut, sg, akk]), jed).
word(jedes, det([quant,sw], [mask, sg, gen]), jed).
word(jedes, det([quant,sw], [neut, sg, gen]), jed).
word(jene, det([def,sw], [unsp, pl, nom]), jen).
word(jene, det([def,sw], [unsp, pl, akk]), jen).
word(jene, det([def,sw], [fem, sg, akk]), jen).
word(jene, det([def,sw], [fem, sg, nom]), jen).
word(jenem, det([def,sw], [mask, sg, dat]), jen).
word(jenem, det([def,sw], [neut, sg, dat]), jen).
word(jenen, det([def,sw], [unsp, pl, dat]), jen).
word(jenen, det([def,sw], [mask, sg, akk]), jen).
word(jener, det([def,sw], [unsp, pl, gen]), jen).
word(jener, det([def,sw], [mask, sg, nom]), jen).
word(jener, det([def,sw], [fem, sg, gen]), jen).
word(jener, det([def,sw], [fem, sg, dat]), jen).
word(jenes, det([def,sw], [neut, sg, nom]), jen).
word(jenes, det([def,sw], [neut, sg, akk]), jen).
word(jenes, det([def,sw], [mask, sg, gen]), jen).
word(jenes, det([def,sw], [neut, sg, gen]), jen).
word(kein, det([quant,ge], [neut, sg, nom]), kein).
word(kein, det([quant,ge], [neut, sg, akk]), kein).
word(kein, det([quant,ge], [mask, sg, nom]), kein).
word(keine, det([quant,ge], [unsp, pl, akk]), kein).
word(keine, det([quant,ge], [unsp, pl, nom]), kein).
word(keine, det([quant,ge], [fem, sg, akk]), kein).
word(keine, det([quant,ge], [fem, sg, nom]), kein).
word(keinem, det([quant,ge], [mask, sg, dat]), kein).
word(keinem, det([quant,ge], [neut, sg, dat]), kein).
word(keinen, det([quant,ge], [unsp, pl, dat]), kein).
word(keinen, det([quant,ge], [mask, sg, akk]), kein).
word(keiner, det([quant,ge], [unsp, pl, gen]), kein).
word(keiner, det([quant,ge], [fem, sg, gen]), kein).
word(keiner, det([quant,ge], [fem, sg, dat]), kein).
word(keines, det([quant,ge], [neut, sg, gen]), kein).
word(keines, det([quant,ge], [mask, sg, gen]), kein).
word(sein, det([poss,ge], [neut, sg, nom]), sein).
word(sein, det([poss,ge], [neut, sg, akk]), sein).
word(sein, det([poss,ge], [mask, sg, nom]), sein).
word(seine, det([poss,ge], [unsp, pl, akk]), sein).
word(seine, det([poss,ge], [unsp, pl, nom]), sein).
word(seine, det([poss,ge], [fem, sg, akk]), sein).
word(seine, det([poss,ge], [fem, sg, nom]), sein).
word(seinem, det([poss,ge], [mask, sg, dat]), sein).
word(seinem, det([poss,ge], [neut, sg, dat]), sein).
word(seinen, det([poss,ge], [unsp, pl, dat]), sein).
word(seinen, det([poss,ge], [mask, sg, akk]), sein).
word(seiner, det([poss,ge], [unsp, pl, gen]), sein).
word(seiner, det([poss,ge], [fem, sg, gen]), sein).
word(seiner, det([poss,ge], [fem, sg, dat]), sein).
word(seines, det([poss,ge], [neut, sg, gen]), sein).
word(seines, det([poss,ge], [mask, sg, gen]), sein).
word(solche, det([quant,sw], [unsp, pl, nom]), solch).
```

```
word(solche, det([quant,sw], [unsp, pl, akk]), solch).
word(solche, det([quant,sw], [fem, sg, akk]), solch).
word(solche, det([quant,sw], [fem, sg, dat]), solch).
word(solchem, det([quant,sw], [mask, sg, dat]), solch).
word(solchem, det([quant,sw], [neut, sg, dat]), solch).
word(solchen, det([quant,sw], [unsp, pl, dat]), solch).
word(solchen, det([quant,sw], [mask, sg, gen]), solch).
word(solchen, det([quant,sw], [neut, sg, gen]), solch).
word(solchen, det([quant,sw], [mask, sg, akk]), solch).
word(solcher, det([quant,sw], [unsp, pl, gen]), solch).
word(solcher, det([quant,sw], [mask, sg, nom]), solch).
word(solcher, det([quant,sw], [fem, sg, gen]), solch).
word(solcher, det([quant,sw], [fem, sg, dat]), solch).
word(solches, det([quant,sw], [neut, sg, nom]), solch).
word(solches, det([quant,sw], [neut, sg, akk]), solch).
word(unser, det([poss,ge], [neut, sg, nom]), unser).
word(unser, det([poss,ge], [neut, sg, akk]), unser).
word(unser, det([poss,ge], [mask, sg, nom]), unser).
word(unsere, det([poss,ge], [unsp, pl, akk]), unser).
word(unsere, det([poss,ge], [unsp, pl, nom]), unser).
word(unsere, det([poss,ge], [fem, sg, akk]), unser).
word(unsere, det([poss,ge], [fem, sg, nom]), unser).
word(unserem, det([poss,ge], [mask, sg, dat]), unser).
word(unserem, det([poss,ge], [neut, sg, dat]), unser).
word(unseren, det([poss,ge], [unsp, pl, dat]), unser).
word(unseren, det([poss,ge], [mask, sg, akk]), unser).
word(unserer, det([poss,ge], [unsp, pl, gen]), unser).
word(unserer, det([poss,ge], [fem, sg, gen]), unser).
word(unserer, det([poss,ge], [fem, sg, dat]), unser).
word(unseres, det([poss,ge], [neut, sg, gen]), unser).
word(unseres, det([poss,ge], [mask, sg, gen]), unser).
word(unsern, det([poss,ge], [unsp, pl, dat]), unser).
word(unsern, det([poss,ge], [mask, sg, akk]), unser).
word(unsre, det([poss,ge], [unsp, pl, akk]), unser).
word(unsre, det([poss,ge], [unsp, pl, nom]), unser).
word(unsre, det([poss,ge], [fem, sg, akk]), unser).
word(unsre, det([poss,ge], [fem, sg, nom]), unser).
word(unsrem, det([poss,ge], [mask, sg, dat]), unser).
word(unsrem, det([poss,ge], [neut, sg, dat]), unser).
word(unsren, det([poss,ge], [unsp, pl, dat]), unser).
word(unsren, det([poss,ge], [mask, sg, akk]), unser).
word(unsrer, det([poss,ge], [unsp, pl, gen]), unser).
word(unsrer, det([poss,ge], [fem, sg, gen]), unser).
word(unsrer, det([poss,ge], [fem, sg, dat]), unser).
word(unsres, det([poss,ge], [neut, sg, gen]), unser).
word(unsres, det([poss,ge], [mask, sg, gen]), unser).
word(welche, det([qu,sw], [unsp, pl, nom]), welch).
word(welche, det([qu,sw], [unsp, pl, akk]), welch).
word(welche, det([qu,sw], [fem, sg, akk]), welch).
word(welche, det([qu,sw], [fem, sg, nom]), welch).
word(welchem, det([qu,sw], [mask, sg, dat]), welch).
word(welchem, det([qu,sw], [neut, sg, dat]), welch).
word(welchen, det([qu,sw], [unsp, pl, dat]), welch).
```

```
word(welchen, det([qu,sw], [mask, sg, akk]), welch).
word(welcher, det([qu,sw], [unsp, pl, gen]), welch).
word(welcher, det([qu,sw], [mask, sg, nom]), welch).
word(welcher, det([qu,sw], [fem, sg, gen]), welch).
word(welcher, det([qu,sw], [fem, sg, dat]), welch).
word(welches, det([qu,sw], [neut, sg, nom]), welch).
word(welches, det([qu,sw], [neut, sg, akk]), welch).
word(welches, det([qu,sw], [mask, sg, gen]), welch).
word(welches, det([qu,sw], [neut, sg, gen]), welch).
```

Bem. Die aus dem CISLEX ermittelten Einträge weichen allerdings hiervon ab, z.B.:

⟨*CISLEX-Kodierung (unbenutzt)*⟩≡

```
word(alle, det([sw], [unsp, pl, nom])).
word(einige, det([st], [fem, sg, akk])).
word(viele, a([], [pos, ge, fem, sg, akk])).
```

Diese Einträge enthalten nicht alle Informationen, die wir für die Syntaxanalyse brauchen.

6.2.6 Junktoren *Wird noch geändert/ausgearbeitet!*

(i) Einstellige Satzjunktoren, die wir nach der adverbialen Funktion des eingebetteten Satzes klassifizieren:

⟨*Wortkategorien*⟩+≡

```
wortkategorie(subj([Art])) :-
        adverbialeFunktion(Art).
```

Da diese alle die Verbletztstellung regieren, können wir dies in der Grammatikregel sichern und geben das nicht als Klassenparameter an.

Das machen wir später! Vorerst benutzen wir stattdessen: wortkategorie(conj([Vst],[Junktor])) :- verbstellung(Vst).

(ii) Zweistellige Satzjunktoren, darunter *eingliedrige*, bei denen der Junktor als Formmerkmal (eigentlich mehr wegen der Baumanzeige) und die Verbstellungen der Teilsätze als Klassenmerkmal (Rahmen) notiert werden:

⟨*Wortkategorien*⟩+≡

```
wortkategorie(conj([Vst1,Vst2],[Junktor])) :-
        verbstellung(Vst1),
        verbstellung(Vst2).
```

Daneben gibt es *zweigliedrige* Satzjunktoren, bei denen wir die beiden Teile getrennt beschreiben. Der Zusammenhang geht nur aus dem (Lexikon-)Eintrag für den ersten Teil hervor:

⟨*Wortkategorien*⟩+≡

```
wortkategorie(conj1([Vst1,Junktor2],[Junktor1])) :-
        verbstellung(Vst1).
wortkategorie(conj2([Vst2],        [Junktor2])) :-
        verbstellung(Vst2).
```

Wieder steht der jeweilige Teil als Formmerkmal in der syntaktischen Kategorie, aber der zweite Teil wird im Rahmen des ersten festgeschrieben. (vgl. die Satzregeln in Abschnitt 6.2.6.)

Bem.: Diese Beschreibungen sind bisher nur sehr rudimentär eingehalten. Die Verwendung von Junktoren zur Koordination von Nominal-, Adjektiv-, und anderen Phrasen ist noch zu beschreiben.

Die folgenden Einträge sind für die Grammatik `vp.grm` erstellt und noch kaum überprüft worden:

(i) Konjunktionen zur Verbindung von Sätzen:

⟨*Lexika/konjunktionen.wrd*⟩≡

```
word(aber, conj([zweit], [aber])).
word(ob, conj([letzt], [ob])).
word(denn, conj([zweit], [denn])).
word(oder, conj([zweit], [oder])).
word(sondern, conj([zweit], [sondern])).
word(und, conj([zweit], [und])).
word(allein, conj([zweit], [allein])).
word(als, conj([letzt], [als])).
word(außer, conj([zweit], [außer])).
word(bis, conj([letzt], [bis])).
word(da, conj([letzt], [da])).
word(damit, conj([letzt], [damit])).
word(dass, conj([letzt], [dass])).
word(doch, conj([zweit], [doch])).
word(doch, conj([erst], [doch])).
word(indes, conj([erst], [indes])).
word(indes, conj([letzt], [indes])).
word(jedoch, conj([erst], [jedoch])).
word(jedoch, conj([zweit], [jedoch])).
word(so, conj([erst], [so])).
word(weil, conj([letzt], [weil])).
word(wenn, conj([letzt], [wenn])).
word(wie, conj([letzt], [wie])).
word(sobald, conj([letzt], [sobald])).
word(nun, conj([letzt], [nun])).
word(zu, conj([infzu], [zu])). % satztyp zu in vp.grm

word(je, conj([[je],letzt], [je])).
word(je, conj([[je],erst], [desto])).
word(je, conj([[je],erst], [umso])).
word(je, conj([letzt,[desto]], [je])).
word(je, conj([letzt,[umso]], [je])).
word(je, conj([letzt,[je]], [je])).

word(weder, conj([zweit, [noch]], [weder])).
word(noch, conj([[weder],erst], [noch])).

word(um, conj([[zu]], [um,zu])).

word(wo, conj([wo], [])). %letzt

% H.L.
```

```
word(daß, conj([letzt],[daß])).
word(sodaß, conj([letzt],[sodaß])).
word(dass, conj([letzt],[dass])).
word(sodass, conj([letzt],[sodass])).

word(um,   conj([[infzu]],[um])).
word(ohne, conj([[infzu]],[ohne])).
word(ohne, conj([[daß]],[(ohne,daß)])).
word(so, conj([[daß]],[(so,daß)])).
word(so, conj([[wie]],[(so,wie)])).
word(ohne, conj([[dass]],[(ohne,dass)])).
word(so, conj([[dass]],[(so,dass)])).
word(so, conj([[wie]],[(so,wie)])).
```

(ii) Partikeln: Bei den Partikeln bleibt eine Unterklassifizierung nach syntaktischen Kriterien noch vorzunehmen. Helbig/Buscha benutzt die Einteilung der Partiken in

(a) solche, die zu Nominalphrasen, Verben, Adjektiven und Adverbien treten können, dabei teilweise vor und teilweise nach der Nominalphrase:

⟨*Partikelgruppe 1*⟩≡
```
aber, auch, beinahe, bereits, bloß, eben, erst, etwa, gar, gerade,
geradezu, ja, nahezu, nicht einmal, noch, nur, schon
```

(b) solche, die zu Nomianlphrasen, Adjektiven und Adverbien treten können,

⟨*Partikelgruppe 2*⟩≡
```
besonders, fast, ganz
```

(c) solche, die zu Verben, Adjektiven und Adverbien treten können:

⟨*Partikegruppe 3*⟩≡
```
doch, durchaus, immer, sehr, so, zu
```

(d) solche, die zu Adjektiven und Adverbien treten können (Gradpartikeln):

⟨*Partikelgruppe 4*⟩≡
```
etwas, höchst, recht, überaus, viel, weit, weitaus, ziemlich
```

(e) solche, die zu Nominalphrasen und Adverbien treten können:

⟨*Partikelgruppe 5*⟩≡
```
allein, beispielsweise, zumal
```

(f) solche, die zu Verben treten können:

⟨*Partikelgruppe 6*⟩≡
```
also, denn, nicht, nun, überhaupt
```

Wir benutzen die im Beispieltext `Testbett/wissenschaftsreligion.txt` vom CISLEX als Partikel klassifizierten Wörter. (Die Adjektivgrammatik hat eine Unterklassifikation nach der regierten Adjektivdeklination vorgenommen.)

⟨*Lexika/partikeln.wrd*⟩≡
```
word(allein, part([], [])).
word(allerdings, part([], [])).
word(also, part([], [])).
word(auch, part([], [])).
word(ausschließlich, part([], [])).
word(bereits, part([], [])).
word(bloß, part([], [])).
```

```
word(denn, part([], [])).
word(doch, part([], [])).
word(eigentlich, part([], [])).
word(einfach, part([], [])).
word(einmal, part([], [])).
word(etwas, part([], [])).
word(freilich, part([], [])).
word(ganz, part([], [])).
word(gar, part([], [])).
word(geradezu, part([], [])).
word(gleich, part([], [])).
word(immer, part([], [])).
word(ja, part([], [])).
word(man, part([], [])).
word(nicht, part([], [])).
word(noch, part([], [])).
word(nun, part([], [])).
word(nur, part([], [])).
word(nämlich, part([], [])).
word(schließlich, part([], [])).
word(schon, part([], [])).
word(sehr, part([], [])).
word(selbst, part([], [])).
word(sicher, part([], [])).
word(so, part([], [])).
word(sogar, part([], [])).
word(viel, part([], [])).
word(vielleicht, part([], [])).
word(vollständig, part([], [])).
word(zu, part([], [])).
word(zumindest, part([], [])).
```

(iii) abtrennbare Verbpräfixe:

⟨*Seminarbeitraege/HasselLiTonelli/praep.saetze.wrd*⟩≡

```
word(um, praef([um], [])).
word(ab, praef([ab], [])).
word(aus, praef([aus], [])).
word(auf, praef([auf], [])).
word(weg, praef([weg], [])).
word(vor, praef([vor], [])).
word(dar, praef([dar], [])).
word(über, praef([über], [])).
word(statt, praef([statt], [])).
word(an, praef([an], [])).
word(ein, praef([ein], [])).
word(hervor, praef([hervor], [])).
word(nach, praef([nach], [])).
```

Im CISLEX werden abtrennbare Verbpräfixe durch `VPART` angegeben,

6.2.7 Restriktion von Merkmalen bei Wortarten

Für die Vorberechnung der First-Relation zur Voraussage nützlicher Grammatikregeln während der Syntaxanalyse müssen die Merkmale bei den Wort- und Phrasenkategorien teilweise ignoriert werden, da sonst die Vorberechnung zu aufwendig wird und zu große Hilfstabellen erzeugt.

Bei den Wortarten werden die Merkmale nach folgender Tabelle ignoriert:

⟨*Lexika/wortarten.cat.pl*⟩+≡
```
restrict(v(Verbklasse,Form),v(Verbklasse,Form)) :- !. % vorlaeufig
restrict(v([hilfsverb,Verb],[Pers,Num,Temp,Mod]),
         v([hilfsverb,Verb],[Pers,Num,_,_])) :- !.
restrict(en(Art,Form),en(_,_)) :- !.
restrict(vn(Art,Form),vn(_,_)) :- !.
restrict(nn(Art,Form),nn(_,_)) :- !.
% restrict(pron([Def,Pers,Gen],[Num,Kas]),pron([Def,Pers,_],_)) :- !.
restrict(pron(DefArt,[Gen,Num,Kas,Pers]),pron(DefArt,[_,_,_,Pers])) :- !.
restrict(pron(Art,[Gen,Num,Kas]),pron(Art,[_,_,_])) :- !.
restrict(n([Gen],[Num,Kas]),n([_],[_,_])) :- !.
restrict(n([Gen,Rahmen],[Num,Kas]), n([Gen,Rahmen],[_,_])) :- !.
restrict(n([Gen,Rahmen],[Adekl,Num,Kas]), n([Gen,Rahmen],[Adekl,_,_])) :- !.
restrict(a(Art,[Komp,Adekl,Gen,Num,Kas]), a(Art,[_Komp,Adekl,_,_,_])) :- !.
restrict(a(Art,[Komp,praedik]),a(Art,[Komp,praedik])) :- !.
% restrict(pr([[Kas],AdvF],[Praep]),pr([[Kas],_],[Praep])) :- !.
% restrict(po([[Kas],AdvF],[Praep]),po([[Kas],_],[Praep])) :- !.
restrict(pr([[Kas]],[Praep]),pr([[Kas]],[Praep])) :- !.
restrict(po([[Kas]],[Praep]),po([[Kas]],[Praep])) :- !.
restrict(det([Def,Adekl],[Gen,Num,Kas]),det([Def,Adekl],[_,_,_])) :- !.
restrict(na(Art, _Form), na(_Art,_Form)) :- !.
```

Von den Merkmalen der folgenden Wortkategorien werden (vorerst) alle zur Vorberechnung berücksichtigt:

⟨*Lexika/wortarten.cat.pl*⟩+≡
```
restrict(pdet([[Kas], Gen], [Praep]),pdet([[Kas], Gen], [Praep])) :- !.
restrict(pc1(Art,Form),pc1(Art,Form)) :- !.
restrict(pc2(Art,Form),pc2(Art,Form)) :- !.
restrict(anrede([Gen],[Num]),anrede([Gen],[Num])) :- !.
restrict(titel([unsp],[Num]),titel([unsp],Num)) :- !.
restrict(ordinalzahl,ordinalzahl) :- !.
restrict(kardinalzahl,kardinalzahl) :- !.
restrict(adv(Art,Form),adv(Art,Form)) :- !.
restrict(part(Art, []), part(Art,[])) :- !.
restrict(conj([], []), conj([],[])) :- !.
restrict(conj([Vst], [Junktor]), conj([Vst],[Junktor])) :- !.
restrict(praef([Praefix], []), praef([Praefix],[])) :- !.
```

Nach dem Laden einer Grammatik sollte man prüfen können, ob zu jeder Wortkategorie eine Einschränkung in `restrict/2` definiert wurde:

⟨*Lexika/wortarten.cat.pl*⟩+≡
```
wortarten_restrictor :-
        (unbeschraenkte_wortkategorien(user) -> true ; fail).

unbeschraenkte_wortkategorien(Strom) :-
        wortkategorie(Cat),
        (restrict(Cat,CatR)
        -> true
        ; write(Strom, 'Wortart '),write(Strom,Cat),
```

```
            write(Strom,' hat keinen Restriktor.\n')),
        fail.
unbeschraenkte_wortkategorien(Strom).
```

Bei der Erzeugung eines `.epl`-Parsers muß geprüft werden, ob zu jeder Wortart (und zu jeder grammatischen Kategorie) eine Einschränkung in `restrict/2` formuliert ist. Fehlende Einschränkungen in `restrict/2` bewirken, daß die davon abhängende Relation `first/2` für den `.epl`-Parser zu klein wird und dann mögliche Analysen in `complete2/3` irrtümlich weggefiltert werden!)

6.3 Nominalphrasen

6.3.1 Klassen- und Formparameter

In der Datei `Grammatiken/np.cat.pl` geben wir wieder gesondert die Merkmale von Nominalphrasen und ihre Wertebereiche an und erklären sie zu Hilfsprädikaten, die in die für die Parser vorberechneten Grammatiken übertragen werden.

Das folgende Prädikat `kategorie/1` wird ggf. in weiteren Teilgrammatiken erweitert und ist daher entsprechend deklariert. (Bisher wird es nicht verwendet; ggf. muß beim Beginn der Vorberechnungen ein vorhandenes Prädikat `kategorie/1` gelöscht werden.)

⟨*Grammatiken/np.cat.pl*⟩≡
```
% :- use_module('Lexika/wortarten.cat.pl').
:- ensure_loaded('Lexika/wortarten.cat.pl').
auxiliary_predicates([persGenNum/3,definitheit_np/1,genusErw/1,restrict/2]).

:- multifile kategorie/1.
```

Die Kategorie der Nominalphrasen hat folgende Klassen- und Formparameter:

⟨*Grammatiken/np.cat.pl*⟩+≡
```
kategorie(np([Def,Pers,Gen],[Num,Kas])) :-
        persGenNum(Pers,Gen,Num),
        kasus(Kas),
        definitheit_np(Def).

kategorie(n1([Gen],[Komp,Adekl,Num,Kas])) :-
        genusErw(Gen),
        numerus(Num),
        kasus(Kas),
        (komparation(Komp); Komp = -),
        (adjektivdeklination(Adekl); Adekl = -).
```

Zur Vorberechnung der First-Relation der Grammatik sollen nur die Merkmale der Definitheit, des Genus, und des Numerus berücksichtigt werden:

⟨*Grammatiken/np.cat.pl*⟩+≡
```
restrict(np([Def,Pers,Gen],[Num,Kas]),np([Def,_,_Gen],[Num,_])) :- !.
restrict(n1([Gen],[Komp,Adekl,Num,Kas]),n1([_Gen],[_,_Adekl,_Num,_])) :- !.
```

Eine genauere Beschreibung erfordert mindestens noch ein Merkmal, das die Komparationsstufe eingebetteter Adjektivmodifikationen angibt: man braucht sie zur Einschränkung der Nominalphrasen in Vergleichskonstruktionen wie `ein größerer Fehler als dieser`. Allerdings wäre dieses Komparationsmerkmal, wenn es ein Formmerkmal ist, nur bei bestimmten Nominalphrasen variierbar.

⟨*Grammatiken/np.cat.pl*⟩+≡
```
persGenNum(3,Gen,pl) :- genusErw(Gen).
persGenNum(3,Gen,sg) :- genus(Gen).
persGenNum(Pers,unsp,Num) :-
        (Pers = 1; Pers = 2), numerus(Num).

definitheit_np(def).
definitheit_np(indef).
definitheit_np(rel).
definitheit_np(qu).
definitheit_np(quant).
definitheit_np(poss).
```

Die Merkmalbereiche der Art- und Formmerkmale `person/1`, `genus/1`, `numerus/1`, `kasus/1`, `adjektivdeklination/1` und `komparation/1` haben die üblichen Werte, die in `Lexika/wortarten.cat.pl` festgelegt wurden; sie werden von `np.grm`) geladen, und müssen in den erzeugten Parser exportiert werden:

⟨*Grammatiken/np.cat.pl*⟩+≡
```
auxiliary_predicates([kasus/1,numerus/1,person/1,genus/1,genusErw/1]).
auxiliary_predicates([adjektivdeklination/1,komparation/1]).
```

Die indefiniten Pronomina im Singular wie `man,jemand`, haben im CISLEX das Genus `unsp`, sollten aber `mask` haben: im Moment werden sie durch `persGenNum` ausgeschlossen! (Man soll *seine* Grammatik einfach halten!)

6.3.2 Grammatikregeln

Wir geben optionale Konstituenten in der Form `{Kategorie}` an. (Dazu sind die Parser erweitert worden.)

⟨*Grammatiken/np.grm*⟩≡
```
% Deklarartionen:

:- op(1100,xfx,'-->').
:- multifile auxiliary_predicates/1.

:- consult('Lexika/wortarten.cat.pl').
:- consult('Grammatiken/np.cat.pl').

% Hilfsprädikate:

auxiliary_predicates([start_symbol/1,kongruenzGenusNP/2,kongruenzAdeklNP/2]).

start_symbol(np([Def,Pers,Gen],[Num,Kas])).
```

(i) Nominalphrasen in der dritten Person können aus einem Determinator und einem (optional) modifizierten Nomen gebildet werden. Das Genus des Determinators wird vom Nomen regiert, die Adjektivdeklination des modifizierten Nomens vom Determinator:

⟨*Grammatiken/np.grm*⟩+≡
```
np([Def,3,Gen],[Num,Kas]) -->
        det([Def,AdeklD],[GenD,Num,Kas]),
        n1([Gen],[Komp,Adekl,Num,Kas]),
        {s([rel(GenD,Num)],[_Temp,_Mod,letzt])}
        :-
        % definitheit_np(Def),
        % kasus(Kas),
        % numerus(Num),
        % komparation(Komp),
        kongruenzGenusNP(Gen,GenD),
        kongruenzAdeklNP(Adekl,AdeklD).

kongruenzGenusNP(Gen,GenD) :-
        % genus(Gen),
        (GenD = Gen ; GenD = unsp).
kongruenzAdeklNP(Adekl,AdeklD) :-
        % adjektivdeklination(AdeklD),
        (Adekl = AdeklD ; Adekl = -).
```

⟨*Testbett/Beispiele/np.txte*⟩≡
```
kein Zufall. keine Fragen. einige Menschen. diese Frau. welcher Mann.
solchen popularisierenden Schlussfolgerungen. % ?? solche:det
```

Das modifizierte Nomen kann durch eine vorangestellte (deklinierte) Adjektivphrase und durch nachgestellte Genitiv- oder Präpositional-Attribute modifiziert sein. Das adjektivisch modifizierte Nomen erbt Komparation und Adjektivdeklination vom Adjektiv:

⟨*Grammatiken/np.grm*⟩+≡
```
n1([Gen],[Komp,Adekl,Num,Kas]) -->
        ap([Komp,Adekl,GenA,Num,Kas]),
        (n([Gen],[Num,Kas]) ; en([Gen],[Num,Kas]) ; en([unsp],[Num,Kas])),
        {np([Def2,3,Gen2],[Num2,gen]) ; pp([DefPP],[Praep])}
        :-
        % kasus(Kas),
        % numerus(Num),
        % komparation(Komp),
        % adjektivdeklination(Adekl),
        kongruenzGenusNP(Gen,GenA).
```

⟨*Testbett/Beispiele/np.txte*⟩+≡
```
technische Eingriffe. viele Romane. eine verantwortliche Person.
```

Für den Fall, daß keine Adjektivphrase auftritt, geben wir durch `Komp = -` und `Adekl = -` an, daß keine Adjektiverweiterung vorliegt. (Phrasenflexionsprogramme können daran erkennen, daß diese Nominalphrasen nicht komparierbar sind):[16]

⟨*Grammatiken/np.grm*⟩+≡
```
n1([Gen],[-,-,Num,Kas]) -->
```

[16] 1. Entsprechend sollte das Nichtvorliegen/Vorliegen einer NP-Genitivergänzung kodiert werden: diese kollidiert ggf. mit einem Possesivum als Determinator. 2. Ich habe nicht gesehen, wie man das in *einer* Regel mit einer *optionalen* Adjektivphrase formulieren kann: wie sollte man in der Bedingung unterscheiden, ob die optionale Adjektivphrase vorhanden ist oder nicht?

```
        n([Gen],[Num,Kas]),
        {np([Def2,3,Gen2],[Num2,gen]) ; pp([DefPP],[Praep])}.
        % :-
        % kasus(Kas), genus(Gen), numerus(Num).
```

⟨*Testbett/Beispiele/np.txte*⟩≡

```
ein Problem der Naturwissenschaften. die Abwesenheit von Zwang.
Mittel der Evolution. Versuche zur Optimierung.
```

Bem.: Man kann zwischen 'freien' Ergänzungen und Komplementen unterscheiden, wenn man bei den Nomen einen Komplementrahmen einfügt - was sicher für die nominalisierten Verben sinnvoll ist.

(ii) Nominalisierte Adjektive, die durch vorangestellte adjektivische oder nachgestelle nominale oder präpositionale Ergänzungen erweitert sein können, sind erweiterte Nomina:

⟨*Grammatiken/np.grm*⟩+≡

```
n1([Gen],[Komp,Adekl,Num,Kas]) -->
        {ap([KompA,Adekl,GenA,Num,Kas])},
        na([Gen],[Komp,Adekl,Num,Kas]),
        {np([Def2,3,Gen2],[Num2,gen]) ; pp([DefPP],[Praep])}
        :-
        % kasus(Kas),
        % numerus(Num),
        % komparation(Komp),
        % adjektivdeklination(Adekl),
        kongruenzGenusNP(Gen,GenA).
```

⟨*Testbett/Beispiele/np.txte*⟩+≡

```
ein Betroffener.
der Betroffene. % scheitert.
der Betroffenen.
die Betroffenen.
den Betroffenen.
das Kontroverse an der Globalisierung.
```

Bem.: Die Lexikoneinträge zu nominalisierten Adjektiven scheinen mir nicht etwas unvollständig und fehlerhaft zu sein. Möglicherweise bewirkt auch ein Genus `undef` im Singular ein Problem mit der Kongruenz zum Artikel.

(iii) Eigennamen, die durch eine vorangestellte Tätigkeitsbezeichnung oder eine Anrede erweitert sein können, bilden ein erweitertes Nomen, das aber nicht adjektivisch erweitert ist:

⟨*Grammatiken/np.grm*⟩+≡

```
n1([Gen],[-,-,Num,Kas]) -->
        {n([Gen],[Num,Kas])}, en([Gen], [Num,Kas]).
```

⟨*Testbett/Beispiels/np.txte*⟩≡

```
der Wissenschaftler Sigmund Freud.
der Wissenschaftler Darwin.        % geht nicht
```

(iv) Modifizierte Nomina im Plural sind volle (indefinite) Nominalphrasen:

⟨*Grammatiken/np.grm*⟩+≡

```
np([indef,3,Gen],[pl,Kas]) -->
        n1([Gen],[Komp,Adekl,pl,Kas]),
        {s([rel(unsp,pl)],[_Temp,_Mod,letzt])} :-
        persGenNum(3,Gen,pl),
        (Adekl = st; (Adekl = -, Komp = -)).
        % kasus(Kas).
```

⟨*Testbett/Beispiele/np.txte*⟩+≡

```
Männer.
logische Probleme.
```

(v) Modifizierte Nomina im Singular sind ebenfalls vollständige (indefinite) Nominalphrasen, wenn das Adjektivattribut stark dekliniert ist:

⟨*Grammatiken/np.grm*⟩+≡

```
np([indef,3,Gen],[sg,Kas]) -->
        n1([Gen],[Komp,Adekl,sg,Kas]),
        {s([rel(Gen,sg)],[_Temp,_Mod,letzt])} :-
        persGenNum(3,Gen,sg),
        (Adekl = st; (Adekl = -, Komp = -, Kas = nom)).
        % kasus(Kas).
```

⟨*Testbett/Beispiele/np.txte*⟩+≡

```
Kunst.
wissenschaftlicher Erkenntnis.
großem Zweifel.
größeren Zweifel.
von jeglichem Zweifel.
in immer größerem Umfang.
die Entwicklung intelligenter Software.
```

Nur im Nominativ (und im Akkusativ?) braucht kein Adjektiv vorhanden zu sein: `Haß ist eine Leidenschaft`. Aber im Genitiv und Dativ braucht man einen Artikel oder ein Attribut: `* die Entwicklung Software`.

(vi) Definite, interrogative und relativierende Personalpronomina sind Nominalphrasen, die nach der Person und dem Genus klassifiziert sind:

⟨*Grammatiken/np.grm*⟩+≡

```
np([Def,Pers,Gen],[Num,Kas]) -->
        pron([Def,Pron],[Gen,Num,Kas,Pers]) :-
        (Pron = pers ; Pron = pos ; Pron = neg),
        ((Pers = 3, genusErw(Gen)) ; ((Pers = 2 ; Pers = 1), Gen = unsp)).

np([Def,Pers,Gen],[Num,Kas]) -->
        pron([Def],[Gen,Num,Kas,Pers]) :-
        (Def = refl ; Def = qu),
        (Pers = 3 ; ((Pers = 2 ; Pers = 1), Gen = unsp)).

np([Def,Pers,Gen],[Num,Kas]) -->
        pron([Def],[Gen,Num,Kas]) :-
        Def = rel,
        (Pers = 3 ; ((Pers = 2 ; Pers = 1), Gen = unsp)).

np([Def,Pers,Gen],[Num,Kas]) -->
        pron([Def],[Gen,Num,Kas,Pers]),
        {s([rel(Gen,Num)],[_Temp,_Mod,letzt])} :-    % Fehler Pers=3: GenPron diff GenSrel
        Def = qu,
        (Pers = 3 ; ((Pers = 2 ; Pers = 1), Gen = unsp)).
```

Die Klassenmerkmale der Pronomina müssen hier und im Lexikon zum Teil noch überdacht bzw. an die obige Spezifikation angepaßt werden.

⟨*Testbett/Beispiele/np.txte*⟩+≡

```
wir. uns. sie. wem. wessen. nichts. etwas. einer. jemand. niemand.
man. sich. seiner.
```

(vii) Konjunktion und Disjunktion von Nominalphrasen sind Nominalphrasen im Plural:[17]

⟨*Grammatiken/np.grm*⟩+≡

```
np([Def,3,Gen],[pl,Kas]) -->
        np([Def1,3,Gen1],[Num1,Kas]),
        conj([],[]),
        np([Def2,3,Gen2],[Num2,Kas])
        :-
        % genusErw(Gen1), genus(Gen2),
        ((Gen1 = Gen2, Gen = Gen1) ; Gen = unsp),
        % definitheit_np(Def1), definitheit_np(Def_2),
        ((Def1 = Def2, Def = Def1) ; Def = Def1).
        % numerus(Num1),kasus(Kas).
```

Ihr Genus kann unspezifiziert sein, wenn eine der Nominalphrasen ein unspezifiziertes Genus hat, oder wenn die Teilphrasen unterschiedliche Genera haben. Unterschiedliche Definitheitseigenschaften der Teilphrasen sind hier ausgeschlossen worden.

⟨*Testbett/Beispiele/np.txte*⟩+≡

```
Romane oder Filme.
Wissenschaft und Technik.
Virchow und Haeckel.
Experten und Gutachtern.
Hirnforschern, Psychologen und Neurophilosophen.
sozialen Veränderungen und wissenschaftlicher Erkenntnis.
die Zustimmung gleicher und freier Staatsbürgerinnen und Staatsbürger.
```

(viii) Eigennamen (inklusive Anrede und Titel) sowie Vornamen bilden eine Nominalphrase der dritten Person (nach den `n1`-Regeln aber auch adjektivisch erweiterte Nomina):

⟨*Grammatiken/np.grm*⟩+≡

```
np([def,3,Gen],[Num,Kas]) -->
        anrede([Gen],[Num]), en([Gen2],[Num,Kas]),
        {s([rel(Gen,Num)],[_Temp,_Mod,letzt])}
        :- kongruenzGenusNP(Gen,Gen2).
np([def,3,Gen],[Num,Kas]) --> en([Gen],[Num,Kas]), {s([rel(Gen,Num)],[_Temp,_Mod,letzt])}.
np([def,3,Gen],[Num,Kas]) --> vn([Gen],[Num,Kas]), {s([rel(Gen,Num)],[_Temp,_Mod,letzt])}.
```

(Das Genus der Anrede überschreibt das ggf. unspezifische Genus des Eigennamens, daher geht es nicht in *einer* Regel mit optionaler Anrede vor dem Eigennamen.)

⟨*Testbett/Beispiele/np.txte*⟩+≡

```
Hr. Freud.
Hr. Dr. Freud.
Fr. Freud.
Sigmund. Maria.
```

Bem.: Die Anrede sollte einen Kasus tragen, um etwa zwischen `Hr.` und `Hrn.` unterscheiden zu können; ist `Hrn.` tatsächlich noch eine Anrede, d.h. muß die Anrede nicht doch stets im Nominativ erfolgen?

Obwohl Numerusmarkierung kaum an Vornamen, Genusunterschiede (im Deutschen) kaum bei Nachnamen auftreten, und Kasusunterschiede bei beiden fast nur beim Genitiv vorkommen, sehen wir diese Merkmale für den zusammengesetzten Namen vor (z.B. für den Vergleich mit

[17] Diese Regel bringt die Vorberechnung des Predictors an Platzgrenzen, wenn die Merkmalbereiche nicht auskommentiert werden!

einer Anrede oder einer Tätigkeits- oder Funktionsbezeichnung wie in `der Mediziner Sigmund Freud`):

⟨*Grammatiken/np.grm*⟩+≡
```
en([unsp], [Num,Kas]) -->
        {titel([unsp],[Num])}, nn([],[Num,Kas]).
```

(Für seltene Formen, wie denen in `die Meiers`, `alle Hänse`, `den Fritzen`, `Franzens Feste`, kann man dann Einträge im Lexikon angeben.)

⟨*Testbett/Beispiele/np.txte*⟩+≡
```
Platon.
Prof. Darwin.
```

Falls ein Vorname vor dem Nachnamen auftritt, lassen wir nur den Singular zu:

⟨*Grammatiken/np.grm*⟩+≡
```
en([Gen], [sg,Kas]) -->
        {titel([unsp],[sg])},
        vn([Gen],[sg,nom]), nn([],[sg,Kas]).
```

⟨*Testbett/Beispiele/np.txte*⟩+≡
```
Dr. Sigmund Freud.
```

Fehler: Korrekt wäre es, wenn erst die um Artikel und Adjektive erweiterten `en`'s zu Nominalphrasen würden. Außerdem sollten aufeinanderfolgende Titel und Vornamen erlaubt sein.[18]

(ix) Hilfsweise wird eine Löschregel angegeben, damit `del/1` existiert (sonst bekommt man einen Fehler):

⟨*Grammatiken/np.grm*⟩+≡
```
loeschbar --> [].
```

Bemerkung 6.2 Zu den Konstruktionen von Nominalphrasen, die noch nachzutragen sind, gehören:

(viii) Nominalphrasen mit integriertem Relativsatz.

(ix) Nominalphrasen mit rechtsextrahiertem Relativsatz, wenn ein allgemeiner Mechanismus zur Konstituentenextraktion implementiert wird.

(x) Nominalphrasen mit Weglassung des Nomens, wie in

⟨*Testbett/Beispiele/np.txte (geht nicht)*⟩≡
```
eines seiner Probleme. viele der schwierigsten Probleme.
das schwierigste seiner Probleme.
```

Mit den bisherigen Regeln erhält man nur die nicht-elliptischen Formen wie `das schwierigste [Problem] seiner Probleme`.

(xi) Nominalphrasen der dritten Person, die aus einem Eigennamen für Sachen (Städte, Länder, Flüsse) bestehen. Diese Eigennamen werden im Lexikon (noch) als einfache Nomina klassifiziert. Das erlaubt zwar

⟨*Testbett/Beispiele/np.txte*⟩+≡
```
das schöne München. der Main.
```

[18] Das könnte man –auch für Reihungen von Adjektiven– durch eine Notation `{Cat}+` erreichen, die im Parser je nach 1-Vorausschau gelöscht, zu `Cat, {Cat}+` oder nur zu `Cat` expandiert wird.

aber nicht die Verwendung von `München` als volle Nominalphrase, wie in `Münchens Staßen. die Straßen Münchens. Frankfurt am Main`.

(xii) Nominalphrasen mit einer Funktions- oder Typbezeichnung, wie `die Stadt Frankfurt`, `das Altersheim Seeblick`, oder `der Metzger Hans Wurst`.

Man muß auch noch erlauben, daß die Possessivpronomina als Determinatoren verwendet werden können (auch für die Relativsatzanschlüsse).

6.3.3 Erkennung und Analyse von Beispielphrasen

Im Verzeichnis `Testbett/Beispiele` finden sich Dateien mit Nominalphrasen, die dem Referenztext `Testbett/wissenschaftsreligion.txt` entnommen oder zumindest mit Wörtern aus einem der Referenztexte gebildet sind.

Mit `erkennePhrasen/0` (oder mit `Erkenne Testphrasen` aus dem Menü) kann man eine Datei lesen, in Phrasen zerlegen und die Phrasen mit dem Erkenner auf grammatiksche Korrektheit überprüfen lassen:

⟨*Erkennen von Beispielphrasen*⟩≡

```
?- erkennePhrasen.
Eingabedatei: 'Testbett/Beispiele/np.txte'.
Ausgabedatei: Testbett/Beispiele/np.txte.erk.app.epl.21.4.2002:22:51
...........................
Done.
```

Die Ergebnisse werden in eine Ausgabedatei im gleichen Verzeichnis geschrieben; sie erhält ein Suffix `.erk.<Grammatik>.<Parser>.<Datum>:<Uhrzeit>`. Für jede bearbeitete Phrase wird ein Punkt `.` am Bildschirm ausgegeben.

Entsprechend kann man mit `analysierePhrasen/0` (oder aus dem Menü mit `Analysiere Testphrasen`) die Eingabedatei lesen, in Phrasen zerlegen, die Phrasen mit dem Parser analysieren und die Ergebnisse auf die Ausgabedatei mit dem Suffix `.ana.<Grammatik>.<Parser>.<Datum>:<Uhrzeit>` schreiben:

⟨*Erkennen von Beispielphrasen*⟩+≡

```
?- analysierePhrasen.
Eingabedatei: 'Testbett/Beispiele/np.txte'.
Ausgabedatei: Testbett/Beispiele/np.txte.ana.app.epl.21.4.2002:23:2
...........................
Done.
```

An der Uhrzeit (Minute) kann man die Ergebnisse verschiedener Tests unterscheiden, sodaß man nach einer Änderung der Grammatik, des Lexikons oder des Parsers prüfen kann, ob die bisher erkannten oder analysierten Beispiele vom geänderten Programm ebenfalls erkannt oder analysiert werden. (Die Analysen muß man allerdings von Hand oder mit `ediff` vergleichen.)

Die Testdatei muß nicht im Verzeichnis `Testbett/Beispiele` stehen. Dort stehen weitere Dateien mit Nominalphrasen aus dem Referenztext `wissenschaftsreligion.txt`, nämlich `Testbett/Beispiele/np.txt` und (sortierter) `Testbett/Beispiele/np.etxt`; sie enthalten aber schon Adjektive und Präpositionalobjekte; Dateien mit den jeweiligen Ergebnissen des Erkenners und Parsers finden sich ebenfalls dort.

Einige Beispiele, die nicht funktionieren (manche funktionierten in früheren Versionen, wie man an den Ergebnisdateien sehen kann), sind:

⟨*Testbett/Beispiele/np.fehler*⟩≡
```
nichts anderem.
anderes.
eine solche Erklärung.
der Berliner Ausstellung 'Sieben Hügel'.
Freuds Entdeckung des unbewussten.
sich selbst reproduzierende Maschinen.
menschlicher Natur und Technik.
Naturwissenschaften, technischen Wissenschaften und Ökonomie.
```

Bem. Die Kategorie für `und` und `oder` ist `conj([],[])`, ein Mißbrauch der Kategorie der Satzjunktoren.

6.4 Präpositionalphrasen

Wir geben eine Beispielgrammatik für Präpositionalphrasen an und hoffen, damit die vorhandenen Beispiele in solche mit gleichen Analysen und unanalysierbare Beispiele einteilen zu können, mit denen die Grammatik verbessert werden kann.

6.4.1 Klassen- und Formparameter

In der Datei `Grammatiken/pp.cat.pl` werden die Merkmale und ihre Wertebereiche von Präpositionalphrasen definiert. Sie werden bei den Vorberechnungen der Grammatiken und später bei der Satzanalyse gebraucht. Damit sie bei den Vorberechnungen in die an die Parser angepassten Grammatiken übertragen werden, müssen sie zu Hilfsprädikaten erklärt werden:

⟨*Grammatiken/pp.cat.pl*⟩≡
```
auxiliary_predicates([definitheit_pp/1]).
```

Präpositionalphrasen klassifizieren wir nach ihrer Definitheit und der einleitenden Präposition (mit Kasus), die für die Abhängigkeit vom Verb bei Präpositionalobjekten benötigt wird:[19]

⟨*Grammatiken/pp.cat.pl*⟩+≡
```
kategorie(pp([Def],[Praep])) :-
        definitheit_pp(Def),
        praeposition(Praep).
```

Zur Vorberechung der First-Relation sollen sowohl die Definitheit wie die Form (in der die Präposition vermerkt ist) berücksichtigt werden:

⟨*Grammatiken/pp.cat.pl*⟩+≡
```
restrict(pp([Def],Form),pp([Def],Form)) :- !.
```

[19] Das Prädikat `kategorie/1` wird z.Zt. nicht benutzt und dient mehr der Dokumentation.

Bei den Merkmalen wird die Definitheit zur Unterscheidung von definiten, interrogativen, und relativierenden Präpositionalphrasen wie `auf dem Stuhl`, `auf welchem Stuhl`, `auf dem` benutzt. (Beachte, daß `praeposition/1` vom Lexikon abhängt.) Mit der Definitheit werden, wie bei anderen Phrasen, Unterschiede in der Verwendung beschrieben, die u.a. für die Bestimmung der Satzart wichtig sind.

Definite Präpositionalphrasen sind z.B. `in der Grube`, `darin`, *indefinite* sind z.B. `in einer Grube`, `in Gruben`.

⟨*Grammatiken/pp.cat.pl*⟩+≡
```
definitheit_pp(def).
definitheit_pp(indef).
```

Für Präpositionalobjekte von Verben, die sich reflexiv auf das Subjekt beziehen, z.B. bei `(es) mit sich bringen`, braucht man eine Reflexiv-Markierung, für die Kongruenz eines eingebetteten Reflexivpronomen mit dem Satzsubjekt:[20]

⟨*Grammatiken/pp.cat.pl*⟩+≡
```
definitheit_pp(def(refl)).
```

Relativierende Präpositionalphrasen wie in `der Fehler, auf den niemand achtet`, werden nach dem Genus des Bezugsnomens unterklassifiziert:

⟨*Grammatiken/pp.cat.pl*⟩+≡
```
definitheit_pp(rel(Gen,sg)) :-
        genus(Gen).
definitheit_pp(rel(unsp,pl)).
```

Interrogative Präpositionalphrasen, z.B. `mit welchem Trick`, erfordern normalerweise keine Unterscheidung nach dem Genus; wenn die eingebettete Nominalphrase ein Interrogativpronomen ist, kann man aber nach dessen Genus unterscheiden, um den (semantisch begründeten) Unterschied im Gebrauch von `mit wem` und `mit was` nicht zu verwischen:

⟨*Grammatiken/pp.cat.pl*⟩+≡
```
definitheit_pp(qu).
definitheit_pp(qu(Gen)) :-
        genus(Gen).
```

Als Formparameter geben wir bei Präpositionalphrasen die jeweilige (ggf. zweiteilige) Präposition an. Dann können Präpositionalobjekte in Komplementrahmen durch die jeweilige Präposition angegeben werden: sie regieren die Form des Präpositionalobjekts.

⟨*Grammatiken/pp.cat.pl*⟩+≡
```
praeposition(Praep) :-
        setof(P,word(P,pr(_,_)),Ps),
        member(Praep,Ps).
```

[20] Braucht man auch eine Markierung für Kongruenz in Person und Numerus: `Ich bin stolz (auf dich:def):def.` `Du bist stolz (auf dich:refl):refl.` ? Allerdings scheint das Präpositionalobjekt mit eingebettetem Personalpronomen nie indefinit sein zu können (auch bei indefinitem Subjekt). Man sollte Reflexivpronomen benutzen, um Gleichheit des Objekts mit dem Subjekt auszudrücken, unabhängig davon, ob das Verb reflexiv ist, d.h. *nur* ein mit dem Subjekt gleiches Objekt erlaubt.

6.4.2 Grammatik für Präpositionalphrasen

Eine Grammatik für Präpositionalphrasen setzt neben den Parametern der Kategorie auch die Eigenschaften und Grammatikregeln (mindestens) für Nominalphrasen (siehe Abschnitt 6.3) voraus. Sie werden von der Grammatik für Präpositionalphrasen geladen:

⟨*Grammatiken/pp.grm*⟩≡

```
:- ensure_loaded('Grammatiken/pp.cat.pl').
:- ensure_loaded('Grammatiken/np.cat.pl').

% Deklarartionen:

:- op(1100,xfx,'-->').

% Hilfsprädikate: (ausser denen aus den .cat-Dateien)

auxiliary_predicates([start_symbol/1,ppDefinitheit/5]).

start_symbol(pp([Def],[Form])).
```

Die Grammatikregeln sind (im Vergleich zum Lexikon) relativ einfach.

(i) Präpositionalphrasen können aus einer Präposition vor einer Nominalphrase bestehen; die Präposition regiert (normalerweise) den Kasus der Nominalphrase:

⟨*Grammatiken/pp.grm*⟩+≡

```
pp([Def],[Praep]) -->
        pr([[Kas]],[Praep]), np([DefNP,Pers,Gen],[Num,Kas]):-
%       word(Praep,pr([[Kas]],[Praep])),
%       ((Praep = als ; Praep = wie) -> Pers = 3 ; true),
%       persGenNum(Pers,Gen,Num),
%       kasus(Kas),
        ppDefinitheit(DefNP,Pers,Gen,Num,Def).
```

Die Präpositionen, die den Kasus nicht regieren, z.B. `in,an,auf`, die mit 2 Kasus verwendet werden und `wie,als`, die mit allen 4 Kasus verwendet werden, werden im Formmerkmal `Praep` zusammen mit einem Kasus angegeben, z.B. `auf(akk)`, also quasi als unterschiedliche Präpositionen behandelt. (Prüfen, ob das nur bei denen so gemacht ist!)

Die Definitheit der Präpositionalphrase hängt in diesem Fall von der Definitheit der Nominalphrase und deren Genus, Numerus und Person ab:

⟨*Grammatiken/pp.grm*⟩+≡

```
ppDefinitheit(DefNP,Pers,Gen,Num,Def) :-
        definitheit_np(DefNP),
        ((DefNP = rel ; DefNP = qu)
        -> Def =.. [DefNP,Pers,Gen,Num]
        ; Def =.. [DefNP,Pers,Num]).
```

(Wegen `Def =.. [DefNP,...]` muß `DefNP` ein Prolog-Atom sein, keine Variable. Daher ist `ppDefinitheit` nötig!)

⟨*Testbett/Beispiele/pp.etxt*⟩≡

```
durch die Evolution. auf einen Gedanken. durch technische Eingriffe.
als eine gesicherte Erkenntnis.
mit mir. durch dich. % nur .ep-Parser
```

Fehler: `als fortschrittliche Maßnahme` geht nicht, da nur eine `np` zugelassen ist; wir brauchen auch `n1([Gen], [Komp, Adekl, sg, Kas])`.

(ii) Indefinite Präpositionalphrasen können auch aus einer Präposition und einem einfachen Nomen im Singular bestehen, wie bei `aus Eifersucht`. (Diese Regel ist unnötig, *wenn* die entsprechenden Nomenformen stets –wie im Plural– volle Nominalphrasen sind.)

⟨*Grammatiken/pp.grm*⟩+≡
```
pp([indef],[Praep]) -->
        pr([[Kas]],[Praep]), n([Gen],[sg,Kas]).
```

Allerdings scheinen semantisch bedingte Einschränkungen erforderlich zu sein, die hier ignoriert werden: `mit Lust/Geld/*Maschine`, `aus Eifersucht/*Geld`, zumindest bei den adverbial gebrauchten Präpositionalphrasen.

⟨*Testbett/Beispiele/pp.etxt*⟩+≡
```
von Darwin. außer Zweifel. nach Frankfurt.
```

(iii) Dieselbe Regel wie für Präposition mit Nominalphrase gilt auch für Postposition und Nominalphrase:

⟨*Grammatiken/pp.grm*⟩+≡
```
pp([Def],[Postp]) -->
        np([DefNP,Pers,Gen],[Num,Kas]), po([[Kas]],[Postp]) :-
%       word(Postp,po([[Kas]],[Postp])),
%       persGenNum(Pers,Gen,Num),
%       kasus(Kas),
        ppDefinitheit(DefNP,Pers,Gen,Num,Def).
```

⟨*Testbett/Beispiele/pp.etxt*⟩+≡
```
der Chancen wegen. seiner Behauptung nach.
```

(iv) Ebenso können Circumpositionen mit einer Nominalphrase eine Präpositionalphrase bilden:

⟨*Grammatiken/pp.grm*⟩+≡
```
pp([Def],[(Prae,Post)]) -->
        pc1([[Kas]],[(Prae,Post)]),
        np([DefNP,Pers,Gen],[Num,Kas]),
        pc2([[Kas]],[(Prae,Post)]) :-
%       word(Prae,pc1([[Kas]],[(Prae,Post)])),
%       persGenNum(Pers,Gen,Num),
%       kasus(Kas),
        ppDefinitheit(DefNP,Pers,Gen,Num,Def).
```

Wir geben die aus den zwei Bestandteilen `(Prae,Post)` bestehende Circumposition durch zwei Konstituenten mit den Kategorien `pc1` und `pc2` an und notieren den regierten Kasus bei beiden als Klassenparameter.

⟨*Testbett/Beispiele/pp.etxt*⟩+≡
```
in das 20. Jahrhundert hinein. um der Zukunft Willen.
um das Haus herum.
```

(v) Präpositionen, die mit einem Artikel verschmolzen sind, bilden mit einem artikellosen Nominalausdruck im Singular und von entsprechendem Genus eine definite Präpositionalphrase:

⟨*Grammatiken/pp.grm*⟩+≡
```
pp([def],[Praep]) -->
        pdet([[Kas],Gen],[Praep]), n1([Gen],[Komp,Adekl,Num,Kas])
        :- (Adekl = sw ; Adekl = -).
```

Die Adjektivdeklination der `n1` ist undefiniert oder die schwache Deklination.

⟨*Testbett/Beispiele/pp.etxt*⟩+≡
```
im Kampf gegen das Verbrechen. ins 20. Jahrhundert. am Main. im schönen München.
```

(vi) Gewisse Präpositionalphrasen können durch eine Präposition zu einer Präpositionalphrase erweitert werden, z.B. `bis an die Grenze`:

⟨*Grammatiken/pp.grm*⟩+≡
```
pp([Def],[Praep]) -->
        pr([[PraepArg]],[Praep]), pp([Def],[PraepArg]).
```

⟨*Testbett/Beispiele/pp.etxt*⟩+≡
```
% bis ins 20. Jahrhundert. % scheitert
% bis in das 20. Jahrhundert.
bis auf Darwin. bis zum Main.
```

Fehlerquelle: geht nur für wenige Präpositionen, wo `PraepArg` die Kasusmarkierung enthält.

(vii) Schließlich treten Präpositionen auch vor Adverbien auf und erweitern diese zu adverbial verwendeten Präpositionalphrasen, z.B. `ab heute`: *Dazu müssen die Adverbien von Hand klassifiziert werden und die adverbiale Funktion der Präpositionen muß (durch eine Variante von* `pr`*) zugänglich gemacht werden.*

⟨*Grammatiken/pp.grm*⟩+≡
```
pp([Def],[Praep]) -->
        pr([[AdvF]],[Praep]), adv([Def,AdvF],[]).
%       :- adverbialeFunktion(AdvF).
```

Dies sollte auf Adverbialphrasen verallgemeinert werden. Die Form der Adverbien ist vorläufig.

(viii) Definite Pronominaladverbien und Konjunktionaladverbien, die wir schon im Lexikon als Präpositionalphrasen klassifiziert haben, lassen als Ergänzungen einen Verbletztsatz zu, z.B. `darauf, daß jemand hilft,` oder `deswegen, weil niemand half,`. Diese Phrasen sollten als Präpositionalphrasen gelten, damit man bei den Verben die entsprechenden Objekte als Präpositionalobjekte angeben kann: `word(hoffen, v([ [nom,auf] ],Form))` erlaubt dann sowohl `auf Hilfe hoffen`, `worauf hoffen`, `darauf hoffen` als auch `darauf, daß jemand hilft, hoffen`.

Dafür sehen wir eine Regel der Form

⟨*Grammatiken/pp.grm (anpassen)*⟩≡
```
pp([Def],[Praep]) -->
        proAdv([def],[Praep]), s([daß,...],[vl,...]).
```
vor, die aber noch eine Änderung der Lexikoneinträge von Pronominal- und Konjunktionaladverbien und die genaue Form der Satzkategorie erfordert.

(ix) Hilfsweise wird eine Löschregel angegeben, damit `del/1` existiert (sonst bekommt man einen Fehler):

⟨*Grammatiken/pp.grm*⟩+≡
```
loeschbar --> [].
```

Zum Testen dieser Grammatik braucht man ein paar Regeln für eingebettete Phrasen. (Ohne gewisse Teilphrasengrammatiken macht die Vorberechnung hier keinen großen Sinn!)

Beispiel 6.3 Wir kombinieren die obige Grammatik mit der vorausgegangenen Beispielgrammatik für Nominalphrasen. Das Prädikat `auxiliary_predicates/1` wird in verschiedenen Dateien teilweise definiert und muß daher besonders deklariert werden:

⟨*Grammatiken/pp1.grm*⟩≡

```
% :- multifile auxiliary_predicates/1. % schon in Grammatiken/np.grm
⟨Grammatiken/np.grm⟩
⟨Grammatiken/pp.grm⟩
```

Zu beachten ist, daß im allgemeinen die Hilfsprädikate von verschiedenen Teilgrammatiken benötigt werden, aber nur einmal geladen und in die vorberechnete Grammatik exportiert werden dürfen.

6.4.3 Extraktion von Beispielen

Zu den Beispieltexten `Testbett/*.txt` wurden mit Werkzeugen des CISLEX von Petra Maier Vollformenlexika erstellt. Einige Päpositionalphrasen des Texts `wissenschaftsreligion.txt` sind in einer Datei `Testbett/Beispiele/pp.txt` gesammelt.

⟨*Testbett/Beispiele/pp.txt*⟩≡

```
als fortschrittliche Maßnahme im Kampf gegen das Verbrechen.
als gesicherte wissenschaftliche Erkenntnis.
als Bedrohung und als glückliche Chance.
als falsch.
anders als bei Platon.
als wissenschafts- und technikfeindlich.
spannender als viele Romane oder Filme.
als eine menschliche Problemlösungspraxis.
als eine ideale Form, um etwas zu tun.
größere Gefahren als die einzelnen Ergebnisse der Wissenschaft selbst.
als mit Freuds Entdeckung des Unbewussten.
eine größere Rolle als bislang angenommen wurde.
er bestehe letztlich in nichts anderem als verschiedenen Versuchen zur Optimierung.
als Arena für sein Balzverhalten.
als biologische Wesen in der natürlichen Welt und als Kulturwesen in einer Welt symbolischer Bedeutungen.
als ein erfolgreiches Mittel der Evolution.
durch das.
nichts als eine Illusion.
sich wechselseitig als Personen zu deuten.
als Personen.
die Deutung des Menschen als freie, verantwortliche Person.
die Deutung des Menschen als eine verantwortliche Person.
als Geschlechtswesen.
als determinierte Wesen.
als eine verantwortliche Person.
als verantwortliche Personen.
als "Züchtungsregeln".
als Züchtungsregeln.
in der sie nichts aussagen als sich selbst.
an der Endlichkeit.
an den pädagogischen und sozialhistorischen Ausstellungen.
an den Ende des 19. Jahrhunderts ausgetragenen Streit.
an kulturell vermittelten Kriterien.
an diese Illusion.
```

an der Vorstellung eines sich selbst reproduzierenden Roboters.
an naturwissenschaftlichen Erklärungen und technischen Eingriffsmöglichkeiten.
an dem großen Zulauf.
an einem Kunstobjekt.
an diesem zufälligen Ort.
an der Universität Frankfurt am Main.
angesichts der Berliner Ausstellung 'Sieben Hügel'.
angesichts der Bedrohungsszenarien.
auf wissenschaftliche Autorität.
auf eine auffallende Bereitschaft, die Strapazen zu verdammen.
auf Ethnien, Gesellschaften und Kulturen am Anfang des 20. Jahrhunderts.
auf den logischen Fehler.
auf ihre Funktion im Wettbewerb um die günstigsten Fortpflanzungschancen.
auf den Gedanken.
auf Natur.
Auf dieses Symbolsystem.
auf die Deutung des Menschen als freie, verantwortliche Person.
auf die Zustimmung gleicher und freier Staatsbürgerinnen und Staatsbürger.
auf genetisch determinierte, den Gesetzen der Soziobiologie unterworfene Wesen.
die Seiten, auf denen sie angeboten werden.
auf denen.
auf seine Funktion für die Evolution.
aus dem Thriller und aus dem Neuen Testament.
aus einem Sein ein Sollen, aus der Natur verbindliche Verhaltensnormen herleiten.
aus der Soziobiologie.
aus der Perspektive eines naturwissenschaftlichen Beobachters.
aus Freiheit.
aus unserer Selbstdeutung.
aus naturwissenschaftlicher Sicht.
dass diese Verhaltensweisen ausschließlich damit erklärt werden können.
dass wir uns ausschließlich als determinierte Wesen verstehen.
wo wir unser Selbstverständnis ausschließlich an naturwissenschaftlichen
Erklärungen und technischen Eingriffsmöglichkeiten ausrichten.
außer Zweifel.
bei Platon.
bei solchen popularisierenden Schlussfolgerungen.
bei weitem übertreffen.
bei Abwesenheit von Zwang.
durch die Wissenschaft.
durch Männer.
durch technische Eingriffe.
durch Kunst.
durch den Prestigegewinn.
durch Roboter oder sich selbst reproduzierende Maschinen.
durch Maschinen.
durch die Sprache.
durch Kommunikation.
ein erfolgreiches Mittel der Evolution, durch das sie es beeinflussen.
durch eine solche Erklärung.
durch denselben Prozess, der dazu führt.
durch eine göttliche Strafe.
mithilfe von Wissenschaft und Technik.
zwischen Virchow und Haeckel.

```
zwischen Naturwissenschaften, technischen Wissenschaften und Ökonomie.
zwischen sozialen Veränderungen und wissenschaftlicher Erkenntnis.
zwischen Experten und Gutachtern.
zwischen Hirnforschern, Psychologen und Neurophilosophen.
zwischen menschlicher Natur und Technik.
von dem methodischen Zweifel.
vom methodischen Zweifel.
```

Davon können manche schon mit der Grammatik analysisert werden, andere nicht. Neben den wegen der Koordination ausgeschlossenen Fällen handelt es sich hauptsächlich um Probleme mit `als` vor `n1`-Ausdrücken und Vorkommen von `sich` oder `selbst`:

⟨*Negativbeispiele*⟩≡

```
0: als fortschrittliche Maßnahme im Kampf gegen das Verbrechen. - Ungrammatisch.
1: als gesicherte wissenschaftliche Erkenntnis. - Ungrammatisch.
9: größere Gefahren als die einzelnen Ergebnisse der Wissenschaft selbst. - Ungrammatisch.
13: als Arena für sein Balzverhalten. - Ungrammatisch.
16: durch das. - Ungrammatisch.
20: die Deutung des Menschen als freie, verantwortliche Person. - Ungrammatisch.
28: in der sie nichts aussagen als sich selbst. - Ungrammatisch.
31: an den Ende des 19. Jahrhunderts ausgetragenen Streit. - Ungrammatisch.
32: an kulturell vermittelten Kriterien. - Ungrammatisch.
34: an der Vorstellung eines sich selbst reproduzierenden Roboters. - Ungrammatisch.
39: an der Universität Frankfurt am Main. - Ungrammatisch.
42: auf wissenschaftliche Autorität. - Ungrammatisch.
50: auf die Deutung des Menschen als freie, verantwortliche Person. - Ungrammatisch.
54: auf denen. - Ungrammatisch.
62: aus naturwissenschaftlicher Sicht. - Ungrammatisch.
70: bei Abwesenheit von Zwang. - Ungrammatisch.
81: durch eine solche Erklärung. - Ungrammatisch.
```

6.5 Adjektivphrasen

6.5.1 Klassen- und Formparameter

Die (Form-)Merkmalbereiche der attributiv verwendeten Adjektivphrase sind von Gradpartikeln regierte Komparation, die von Artikeln regierte Adjektivdeklination, und die Kongruenzmerkmale Genus, Numerus und Kasus:

⟨*Grammatiken/ap.cat.pl*⟩≡

```
kategorie(ap([Komp,ADekl,Gen,Num,Kas])) :-
        komparation(Komp),
        adjektivdeklination(ADekl),
        genusErw(Gen),
        numerus(Num),
        kasus(Kas).
```

Zur Vorberechnung der First-Relation sollen –wie bei den Adjektiven– die Merkmale Genus, Numerus und Kasus ignoriert werden:

⟨*Grammatiken/ap.cat.pl*⟩+≡

```
restrict(ap([Komp,ADekl,Gen,Num,Kas]),ap([_Komp,_ADekl,_,_,_])) :- !.
```

Durch Testen des erzeugten `.epl`-Parsers muß aber überprüft werden, ob (noch?) andere Merkmale ignoriert werden sollten.

Die Merkmalwerte der Komparation und Adjektivdeklination werden in `Lexika/wortarten.cat.pl` erklärt. (Klassenmerkmale haben wir nicht angegeben. Als Parameter der Klasseneinteilung von Adjektivphrasen kommen adverbiale und prädikative Verwendbarkeit in Frage.)

Mit folgender Hilfskategorie werden Ergänzungen durch Modifikatoren (wie Adverbien und Gradpartikel) zusammengefaßt, die die Komparation der Adjektivphrase regieren:

⟨*Grammatiken/ap.cat.pl*⟩+≡
```
kategorie(ergaenzung([Komp])) :-
        komparation(Komp).

restrict(ergaenzung([Komp]),ergaenzung([Komp])) :- !.
```

Zur Konstruktion von Prädikativsätzen werden modifizierte unflektierte Adjektive erlaubt; das geschieht mit der Hilfskategorie

⟨*Grammatiken/ap.cat.pl*⟩+≡
```
praedikat([Komp]) :-
        komparation(Komp).
restrict(praedikat([Komp]),praedikat([Komp])) :- !.
```

6.5.2 Grammatikregeln

Die folgende Grammatik für Adjektivphrasen ist eine[21] Anpassung der Grammatik von `Seminarbeitraege/LenkLeonardi/ap_praed_a.grm` Da sie unten mit der Grammatik für Nominalphrasen und (Prädikativ-)Sätze verbunden wird, ist kein eigenes Startsymbol für Adjektivphrasen erklärt.

⟨*Grammatiken/ap.grm*⟩≡
```
% Deklarationen und Hilfsprädikate

:- op(1100,xfx,'-->').
:- ensure_loaded('Grammatiken/np.cat.pl').
:- ensure_loaded('Grammatiken/ap.cat.pl').
```

(i) Für den attributiven Gebrauch beim Nomen werden Adjektivphrasen aus deklinierten Adjektiven gebildet, und zwar:

   (a) einfache, eventuell um nachgestellte Adjektivphrasen erweiterte Adjektive:

   ⟨*Grammatiken/ap.grm*⟩+≡
   ```
   ap([Komp,Adekl,Gen,Num,Kas]) -->
           a([], [Komp,Adekl,Gen,Num,Kas]),
           { ap([Komp2,Adekl,Gen,Num,Kas]) }
           :- true.
           % komparation(Komp), adjektivdeklination(Adekl),
           % genusErw(Gen), numerus(Num), kasus(Kas).
   ```

[21] Man muß wohl zwischen `a` und `ap` eine Kategorie `a1` einfügen, die um Komplemente oder Modifikatoren erweiterte Adjektive enthält. Damit kann man die Bildung der Adjektivphrasen genauer formulieren, indem man die rekursiven Vorkommen eingeschränkt. Außerdem ist in den verschiedenen Regeln ggf. nur eine Teilklasse aller Adjektive erlaubt, sodaß man eine Klasseneinteilung der Adjektive braucht.

⟨*Testbett/Beispiele/ap.txt*⟩≡

```
technische Eingriffe.
schwierigere Probleme.
schwierige logische Probleme.
gewisse mathematische Fähigkeiten.
die schwierigen logischen Probleme.
einige schwierige logische Probleme.
die schwierigsten logischen Probleme.
die glatten, durchtrainierten Körper. % da Satzzeichen entfernt werden
als freie, verantwortliche Person.    % scheitert an NP
```

(b) Ordinalzahlen können als vollständige, attributiv gebrauchte Adjektivphrasen verwendet werden:

⟨*Grammatiken/ap.grm*⟩+≡

```
ap([pos,Adekl,unsp,sg,_Kas]) --> ordinalzahl
        :- (Adekl = ge ; Adekl = sw). % kasus(Kas).
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡

```
der 3. Mann.
eine 2. Frage.
das 19. Jahrhundert.
```

Wir lassen hier die Pluralformen, wie in `die 2.en Auflagen`, weg. (Die Ordinalzahlen sollten aber wie Adjektive deklinierbar sein: `das 17te Jahr, ein 17ter Versuch`, insbesondere für die nicht durch Ziffern ausgeschriebenen Ordinalzahlen.)

(c) Kardinalzahlen können ebenfalls als vollständige, attributiv gebrauchte Adjektivphrasen verwendet werden. Sie sind nicht durch Adverbien oder Gradpartikel modifizierbar, nicht komparierbar und nicht nach der Adjektivdeklination flektierbar:[22]

⟨*Grammatiken/ap.grm*⟩+≡

```
ap([-,-,unsp,pl,_Kas]) --> kardinalzahl.
```

Man müßte die `1` noch ausschließen (außer bei `der eine Fehler` statt `der 1 Fehler`, oder dafür den Singular vergeben. (Dazu müßte aber die Tokenisierung feinere Unterschiede bei den Zahlausdrücken machen.)

⟨*Testbett/Beispiele/ap.txt*⟩+≡

```
5 Fragen.
den 3 Männern.
einige 100 Fragen.
```

Bem.: Wenn man Kardinalzahlen als attributiv verwendbare Adjektive betrachtet, via

⟨*Grammatiken/ap.grm (unbenutzt)*⟩≡

```
a([kard],[-,-,unsp,pl,_Kas]) --> kardinalzahl.
```

so wird `a([kard],...)` eine *syntaktische* statt einer lexikalischen Kategorie, und das beeinflußt die vorhersagbaren Regeln! Siehe die Bem. 5.5 zur Berechnung von `lexCat`.

(d) Kardinal- und Ordinalzahlen, die um nachgestellte Adjektivphrasen erweitert sind:

⟨*Grammatiken/ap.grm*⟩+≡

```
ap([Komp,Adekl,Gen,Num,Kas]) -->
        (kardinalzahl; ordinalzahl),
        ap([Komp,Adekl,Gen,Num,Kas]).
```

[22] Wenn man Kardinalzahlen als Quantoren betrachtet, fehlen einem Ausdrücke wie `die 3 Tenöre`. Man könnte vielleicht die Kardinalzahlen als stark flektierte Zahladjektive im Positiv ansehen und die Ordinalzahlen als gemischt oder schwach flektierte Zahladjektive im (verallgemeinerten) Superlativ, d.h. Ordinalzahlen als Komparationsstufen der Zahladjektive behandeln.

⟨*Testbett/Beispiele/ap.txt*⟩+≡

```
die 3 guten Fragen.
eine 2. gute Frage.
```

Es ist klar, daß das zu einfach ist: man kann Zahladjektive wiederholen, was semantisch unsinnig ist (und daher syntaktisch nicht auftritt).

⟨*Testbett/Beispiele/ap.txt*⟩+≡

```
die 3. 2 Fragen.  % geht
meine 3 2. Zähne. % geht nicht
```

(e) Adjektivphrasen, die durch vorangestellte Partikel modifiziert sind:

⟨*Grammatiken/ap.grm*⟩+≡

```
ap([Komp,Adekl,Gen,Num,Kas]) -->
        ergaenzung([Komp]),
        ap([Komp,Adekl,Gen,Num,Kas]).
        % :- komparation(Komp), adjektivdeklination(Adekl),
        % genusErw(Gen), numerus(Num), kasus(Kas).
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡

```
scheinbar gesicherter Erkenntnisse.
eine so einschneidende Maßnahme.
so genannte Gewohnheitsverbrecher.
unsere heute praktizierte Kultur.
in unserer gemeinsam geteilten Selbstdeutung.
immer noch viele Menschen.
zumeist nackte menschliche Körper.
ihrer bloß raumfüllenden Anwesenheit.
der viel dominierendere Individualismus.
eine noch viel stärkere Überwältigung der Öffentlichkeit.
das zugleich schrecklichste Beispiel.
in einem immer größeren Umfang.
```

(f) Adjektive, die um ein vorangestelltes Nominalkomplement erweitert sind:

⟨*Grammatiken/ap.grm*⟩+≡

```
ap([Komp,ADekl,Gen,Num,Kas]) -->
        np([Def,3,GenNP],[NumNP,KasNP]),
        a([KasNP], [Komp,ADekl,Gen,Num,Kas]).
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡

```
die den Gesetzen unterworfenen Hirnforscher.
einige den Gesetzen unterworfene Hirnforscher.
den Gesetzen der Soziobiologie unterworfene Wesen.
```

(g) Adjektive, die um ein vorangestelltes Präpositionalkomplement erweitert sind:

⟨*Grammatiken/ap.grm*⟩+≡

```
ap([Komp,ADekl,Gen,Num,Kas]) -->
        pp([Def],[Praep]),
        a([Praep],[Komp,ADekl,Gen,Num,Kas]).
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡

```
der an ihm ausgetragene Streit.
ein an ihm ausgetragener Streit.
an den am Ende des 19. Jahrhunderts ausgetragenen Streit. % ?
die von Darwin begründete Evolutionstheorie.
```

(h) Koordination von Adjektivphrasen:

⟨*Grammatiken/ap.grm*⟩+≡
```
ap([Komp,Adekl,Gen,Num,Kas]) -->
        ap([Komp,Adekl,Gen,Num,Kas]),
        conj([],[]),
        ap([Komp,Adekl,Gen,Num,Kas])
        :- true.
        % komparation(Komp), adjektivdeklination(Adekl),
        % genusErw(Gen), numerus(Num), kasus(Kas).
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡
```
deutende und verstehende Personen.
ihre kulturellen und politischen Relativierungen.
über den demokratischen oder autoritären Aufbau.
das eindrucksvollste und zugleich schrecklichste Beispiel.
eine einmalige, unwiederholbare und deshalb unendliche Lebensgeschichte.
```

(ii) Für den prädikativen Gebrauch (bei wenigen Verben wie `sein`, `werden`) werden Adjektivphrasen aus undeklinierten Adjektiven aufgebaut:

(a) einfache, aus einem möglicherweise durch Partikel oder Adverbien modifizierten Adjektiv bestehende Prädikative (im Positiv oder Komparativ):

⟨*Grammatiken/ap.grm*⟩+≡
```
praedikat([Komp]) -->
        { ergaenzung([Komp]) },
        a([], [Komp,praedik])
        :- (Komp = pos ; Komp = komp).
```

(b) einfache, nicht durch Partikel oder Adverbien modifizierte Adjektive im Superlativ:

⟨*Grammatiken/ap.grm*⟩+≡
```
praedikat([sup]) -->
        a([], [sup,praedik]).
```

(c) koordinierte prädikativ verwendete Adjektivphrasen:

⟨*Grammatiken/ap.grm*⟩+≡
```
praedikat([Komp]) -->
        praedikat([Komp]), conj([],[]), praedikat([Komp])
        :- true.
        % komparation(Komp).
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡
```
Seine quantentheoretischen Lösungen sind wahr oder falsch.
Seine quantentheoretischen Lösungen sind wahrer oder falscher.
```

(d) als Ergänzungen sind Adverbien und (Grad-)Partikel möglich, die die Adjektivdeklination regieren:

⟨*Grammatiken/ap.grm*⟩+≡
```
ergaenzung([Komp]) -->
        (adv([Komp],[]) ; part([Komp],[])).
        % { ergaenzung([Komp]) } :- komparation(Komp).
```

Damit man Prädikativsätze bilden kann, sollte die Satzgrammatik um Regeln erweitert werden, die Prädikative in den drei Kompartationsstufen des Adjektivs erlauben. (Sollte die Komparationsstufe ein Formmerkmal der Prädikativsätze sein?)

Zum Testen fügen wir ein Startsymbol für Prädikativsätze in Verbzweitstellung ein:

⟨*Grammatiken/ap.grm*⟩+≡
```
kategorie(s(zweit)).
restrict(s(zweit),s(zweit)).

start_symbol(s(zweit)).
```

Für Prädikativsätze[23] im Positiv:

⟨*Grammatiken/ap.grm*⟩+≡
```
s(zweit) -->
        np([Def,Pers,Gen],[Num,nom]),
        v([hilfsverb,sein],[Pers,Num,Temp,Mod]),
        praedikat([pos]).
        % :- numerus(Num), genus(Gen).
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡
```
Es ist gelungen.
Jener genchirurgische Eingriff ist zulässig.
Die Menschen sind leichtgläubig.
Ihr wart sehr leichtgläubig.
```

Für Prädikativsätze im Komparativ kann auch ein Vergleichsglied folgen, eine mit `als` gebildete Präpositionalphrase:

⟨*Grammatiken/ap.grm*⟩+≡
```
s(zweit) -->
        np([Def,Pers,Gen],[Num,nom]),
        v([hilfsverb,sein],[Pers,Num,Temp,Mod]),
        praedikat([komp]),
        { pp([DefPP],[als(nom)]) }.
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡
```
Wir waren noch leichtgläubiger.
Sie sind leichtgläubiger als die Hirnforscher.
Die Wissenschaft ist spannender als viele Romane.
```

Für Prädikativsätze im Superlativ ist das Prädikativ mit `am` zu bilden:

⟨*Grammatiken/ap.grm*⟩+≡
```
s(zweit) -->
        np([Def,Pers,Gen],[Num,nom]),
        v([hilfsverb,sein],[Pers,Num,Temp,Mod]),
        am, praedikat([sup]).  % Lexikoneintrag fr "am"  s. adjektive.wrd
```

⟨*Testbett/Beispiele/ap.txt*⟩+≡
```
Das ist am besten. Thriller sind am spannendsten.
```

[23] Bem. Die `np`-Merkmale und die `v`-Merkmale sind an die `vp`-Grammatik angepaßt; bisher wird in Prädikativsätzen nur das Hilfsverb `sein` erlaubt, man muß aber noch andere Hilfsverben (z.B. `werden, bleiben`) zulassen.

Für Prädikativsätze mit Adjektiven, die ein Komplement erfordern, dienen folgende Regeln:

⟨*Grammatiken/ap.grm*⟩+≡
```
praedikat([Komp]) -->
        np([Def,3,Gen],[Num,Kas]),
        a([Kas], [Komp,praedik]).

praedikat([Komp]) -->
        a([],[Komp,praedik]),
        pp([Def],[wie]).

praedikat([Komp]) -->
        pp([Def],[Praep]),
        a([Praep],[Komp,praedik]).
```

Eine Verbindung der Adjektivphrasen mit den Nominal- und Präpositionalphrasen erhält man in der Datei `Grammatiken/app.grm`:

⟨*Grammatiken/app.grm*⟩≡
```
% :- multifile auxiliary_predicates/1. % schon in Grammatiken/np.grm
⟨Grammatiken/np.grm⟩
⟨Grammatiken/pp.grm⟩

:- consult('Grammatiken/ap.cat.pl').
⟨Grammatiken/ap.grm⟩
```

Für Analysen von Adjektivphrasen und Prädikativsätzen siehe die Dateien `Testbett/Beispiele/ap.txt.<Suffix>`. Weitere Beispiele findet man in `Seminarbeitraege/LenkLeonardi/beispiele.ap.<Suffix>`.

Beispiel 6.4 Für den Satz `Sie sind leichtgläubiger als die Hirnforscher` erhält man die folgende Analyse:

⟨*Beispiel*⟩≡
```
1. s(zweit)
      np([def, 3, unsp], [pl, nom])
         pron([def, pers], [unsp, pl, nom, 3])
            Sie
      v([hilfsverb, sein], [3, pl, perf, ind])    % (Tempus falsch)
         sind
      praedikat([komp])
         a([], [komp, praedik])
            leichtgläubiger
      pp([def(3, pl)], [als(nom)])
         pr([[nom]], [als(nom)])
            als
         np([def, 3, mask], [pl, nom])
            det([def, sw], [unsp, pl, nom])
               die
            n1([mask], [-, -, pl, nom])
               n([mask], [pl, nom])
                  Hirnforscher
```

Ist es nicht sinnvoller, das Vergleichsglied als vom komparierten Adjektiv regiert darzustellen?

⟨*scheiternde Beispiele*⟩≡
```
der Wissenschaftler Sigmund Freud.
von jeglichem Zweifel.
in immer größerem Umfang.
```

Fehler: eine `n1` ohne Artikel gilt nur im Plural als `np`.

Bemerkung 6.5 (i) Die Klassifizierung der Adjektive muß verfeinert werden; die im CISLEX scheint nur nach morphologische Eigenschaften getroffen zu sein. Für die Syntax braucht man aber Kriterien wie attributive, adverbiale und prädikative Verwendbarkeit, sowie Graduierbarkeit. (Z.B. Ordinal- und Kardinalzahlen sind besondere Fälle in diesen Hinsichten.)

(ii) Wo stehen die Komplemente des Adjektivs beim attributiven, prädikativen und adverbialen Gebrauch des Adjektivs, und wo stehen die Modifikatoren?

(iii) Partizipien werden (nach dem CISLEX?) als Adjektive behandelt, aber die Grammatik behandelt sie nicht richtig; das Partizip-1 wird nicht einmal von der lexikalischen Analyse erkannt.

Partizipien fungieren wie Adjektive als Nomenattribute, aber sie können (analog zum Passiv des Verbs) ein optionales Präpositionalobjekt haben, das echte Adjektive nicht haben: das wegprojizierte Subjekt des entsprechenden Verbs. Dieses Präpositionalobjekt kann i.a. mit verschiedenen Präpositionen angeschlossen werden: `die {von|durch euch} gestörte Sitzung`. Wann sind welche der Präpositionen erlaubt?

(iv) Ebenso muß die Klassifizierung der Partikel und Adverbien verbessert bzw. eingeführt werden. (Siehe `Seminarbeitraege/LenkLeonardi`). Wie die Ergänzungen einander modifizieren, bedarf einer näheren Überlegung und einer entsprechend feinen Klassifizierung. Zur Zeit sind z.B. `noch viel stärker`, aber auch `viel noch stärker` parsebar, weil sowohl `noch` als auch `viel` als Adverb und Partikel im Lexikon vermerkt sind.

(v) Das Format der Lexikoneinträge für undeklinierte Adjektive ist noch vorläufig.

6.6 Verbalphrasen, finite und infinite Sätze

Die Beispielgrammatik von A. Hassel, S. Tonelli und Y. Li (vgl. Seminarbeiträge) wird hier ein wenig der bisherigen Beschreibung der Teilgrammatiken angepaßt. *Das ist noch nicht abgeschlossen und noch nicht getestet. Sowohl die Satzmerkmale als auch gewisse Lexikoneinträge sind noch zu ändern und zu vereinheitlichen! Die unveränderte Version kann mit* `auswahl(grammar(vp)). auswahl(parser(earley_doerre)).` *benutzt werden.*

Bemerkungen zur Anpassung an das bisherige Format:

(i) Alle Prädikate aus `HasselLiTonelli/np.cat.ddm.pl` sind in `Grammatiken/np.cat.pl` enthalten, und das wird schon von der Teilgrammatik `np.grm` geladen.

(ii) Die Teilgrammatik `HasselLiTonelli/pp.grm.ddm.ed.pl` hat

```
np([def, A, B], [C, D])-->[pron([], [B, C, D, A])]. ,
```

wo `Grammatiken/pp.grm.ed` das allgemeinere (aber offenbar falsche)

```
np([def, A, B], [C, D])-->[pron([], [E, C, D, A])].
```
hat.

6.6.1 Klassen- und Formparameter

⟨*Grammatiken/vp.cat.pl*⟩≡
```
auxiliary_predicates([satzteil/1,satzstellung/1]).

satzteil(refpron(_)).
satzteil(infzu(_)).
satzteil(praep(_)).
satzteil(Kasus) :- kasus(Kasus).
```

Eine Formeigenschaft des deutschen Satzes ist die Verbstellung:

⟨*Grammatiken/vp.cat.pl*⟩+≡
```
satzstellung(erst).
satzstellung(zweit).
%satzstellung(letzt).
```

6.6.2 Grammatik für einfache Sätze

Unter *einfachen Sätzen* werden hier die aus finiten Verben und ihren Komplementen gebildeten Sätze verstanden, wobei die Verben höchstens dreistellig sind (einschließlich dem Subjekt), aber als Komplement auch (einfache) Komplementsätze gelten.

Etwas ungewöhnlich ist hier, daß die grammatischen *Funktionen* wie Subjekt und Objekt zu grammatischen Kategorien gemacht werden. Das dient dazu, viele Regeln in ein Schema zu bringen. (Das könnte man mit Metaregeln beschreiben, ohne explizit die grammatischen Funktionen als Kategorien zu benutzen.) Andererseits ist es üblich, `det` als Kategorie zu verwenden, obwohl es sich beim Determinator auch eher um eine grammatische Funktion handelt: sie kann durch Artikel, Possesivpronomen, Nominalphrasen im Genitiv usw. ausgefüllt werden.

⟨*Grammatiken/vp.grm*⟩≡
```
:- multifile auxiliary_predicates/1.

⟨Grammatiken/pp1.grm⟩

:- consult('Grammatiken/vp.cat.pl').

auxiliary_predicates([start_symbol/1]).

start_symbol(s(Satzstellung)) :-
        satzstellung(Satzstellung).
start_symbol(konj_satz([_], [])).

s(erst) -->    vp([[_Kasus]], [_Per, _Num, erst]).

s(zweit) -->   subjekt([Kasus],[Per, Num]),
                vp([[Kasus]], [Per, Num, zweit]).

s(letzt) -->   subjekt([Kasus],[Per, Num]),
                vp([[Kasus]], [Per, Num, letzt]).

subjekt([conj(Konj)],[3, sg]) --> konj_satz([Konj], []).
subjekt([Kasus],[Per, Num]) --> np([_Def, Per, _Genus], [Num, Kasus]).

objekt([Kasus],[]) --> np([_Def, _Per, _Genus], [_Num, Kasus]).
```

```
objekt([conj(Konj)], []) --> konj_satz([Konj], []).
objekt([infzu([Num,Per])], []) --> satz_infzu([infzu],[Num,Per]).
objekt([refpron([Num,Kasus,Per])], []) --> refpron([],[Num,Kasus,Per]).
objekt([praep(Praep)],[]) --> pp(_, [Praep]).
```

(i) Verbzweitstellung

(a) bei Verben ohne abtrennbares Präfix im Präsens und Präteritum:

⟨*Grammatiken/vp.grm*⟩+≡

```
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([[Kasus1]], [Per1, Num1, _Tempus, _Modus]).
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([[Kasus1, Kasus2]], [Per1, Num1, _Tempus, _Modus]),
        objekt([Kasus2], []).
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([[Kasus1, Kasus2, Kasus3]], [Per1, Num1, _Tempus, _Modus]),
        objekt([Kasus2], []),
        objekt([Kasus3], []).
```

(b) bei Verben mit abtrennbarem Präfix, im Präsens und Präteritum:
Das abtrennbare Präfix wird neben dem Komplementrahmen ein Klassenparameter; es wird im Präsens und Präteritum abgetrennt:

⟨*Grammatiken/vp.grm*⟩+≡

```
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([[Kasus1], Praef], [Per1, Num1, Tempus, _Modus]),
        praef([Praef], []) :- Tempus = praes ; Tempus = praet.

(vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([[Kasus1, Kasus2], Praef], [Per1, Num1, _Tempus, _Modus]),
        objekt([Kasus2], []),
        praef([Praef], [])).

vp([[Kasus1]], [Per1, Num1, zweit]) -->
         v([[Kasus1, Kasus2, Kasus3], Praef], [Per1, Num1, _Tempus, _Modus]),
         objekt([Kasus2], []),
         objekt([Kasus3], []),
         praef([Praef], []).
```

(c) bei Verben mit oder ohne abtrennbarem Präfix im Perfekt:
Hier wird das abtrennbare Präfix nicht abgetrennt, also können beide Verbklassen mit denselben Regeln behandelt werden:

⟨*Grammatiken/vp.grm*⟩+≡

```
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]),
        v([[Kasus1], VerbAux], [part2]).

vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]),
        objekt([Kasus2], []),
        v([[Kasus1, Kasus2], VerbAux], [part2]) :-
        % satzteil(Kasus2),
        (Kasus2 = refpron([Num1,akk,Per1]) ; Kasus2 = infzu([Num1,Per1])).
```

```
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]),
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[Kasus1, Kasus2, Kasus3], VerbAux], [part2]) :-
        % satzteil(Kasus2),
%       % satzteil(Kasus3),
        (Kasus2 = refpron([Num1,dat,Per1]) ; Kasus3 = infzu([Num1,Per1])). /* Problem! */
```

(ii) Verbletztstellung

(a) im Präsens

⟨*Grammatiken/vp.grm*⟩+≡

```
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        v([[Kasus1]], [Per1, Num1, _Tempus, _Modus]).

vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        v([[Kasus1, Kasus2]], [Per1, Num1, _Tempus, _Modus]).

vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[Kasus1, Kasus2, Kasus3]], [Per1, Num1, _Tempus, _Modus]).
```

(b) und im Perfekt:

⟨*Grammatiken/vp.grm*⟩+≡

```
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        v([[Kasus1],VerbAux], [part2]),
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]).
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        v([[Kasus1,Kasus2],VerbAux], [part2]),
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus])
        :-
        % satzteil(Kasus2),
        (Kasus2 = refpron([Num1,akk,Per1]) ;  Kasus2 = infzu([Num1,Per1])).

vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[Kasus1,Kasus2,Kasus3],VerbAux], [part2]),
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus])
        :-
        % satzteil(Kasus2),
        % satzteil(Kasus3),
        (Kasus2 = refpron([Num1,dat,Per1]) ; Kasus3 = infzu([Num1,Per1])).
```

(iii) Verberststellung:

(a) Bei Verben ohne abtrennbarem Präfix im Präsens:

⟨*Grammatiken/vp.grm*⟩+≡

```
vp([[Kasus1]], [Per1, Num1, erst]) -->
```

```
        v([[Kasus1]], [Per1, Num1, _Tempus, _Modus]),
        np([_Def1, Per1, _Genus1], [Num1, Kasus1]).

vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([[Kasus1, Kasus2]], [Per1, Num1, _Tempus, _Modus]),
        subjekt([Kasus1],[Per1, Num1]),
        objekt([Kasus2], []).

vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([[Kasus1, Kasus2, Kasus3]], [Per1, Num1, _Tempus, _Modus]),
        subjekt([Kasus1],[Per1, Num1]),
        objekt([Kasus2], []),
        objekt([Kasus3], []).
```

(b) Bei Verben mit abtrennbarem Präfix, im Präsens:

⟨*Grammatiken/vp.grm*⟩+≡

```
vp([[Kasus]], [Per, Num, erst]) -->
        v([[Kasus], Praef], [Per, Num, _Tempus, _Modus]),
        subjekt([Kasus],[Per, Num]),
        praef([Praef], []).
/*stelle ich es um*/
vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([[Kasus1, Kasus2], Praef], [Per1, Num1, _Tempus, _Modus]),
        subjekt([Kasus1],[Per1, Num1]),
        objekt([Kasus2], []),
        praef([Praef], []).
/*gebe ich das buch bei dir ab*/
vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([[Kasus1, Kasus2, Kasus3], Praef], [Per1, Num1, _Tempus, _Modus]),
        subjekt([Kasus1],[Per1, Num1]),
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        praef([Praef], []).
```

(c) Bei Verben mit oder ohne abrennbarem Präfix, im Perfekt:

⟨*Grammatiken/vp.grm*⟩+≡

```
vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]),
        subjekt([Kasus1],[Per1, Num1]),
        v([[Kasus1], VerbAux], [part2]).

vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]),
        subjekt([Kasus1],[Per1, Num1]),
        objekt([Kasus2], []),
        v([[Kasus1, Kasus2], VerbAux], [part2])
        :-
        % satzteil(Kasus2),
        (Kasus2 = refpron([Num1,akk,Per1]) ; Kasus2 = infzu([Num1,Per1])).

vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]),
        subjekt([Kasus1],[Per1, Num1]),
        objekt([Kasus2], []),
```

```
            objekt([Kasus3], []),
            v([[Kasus1, Kasus2, Kasus3], VerbAux], [part2])
            :-
            % satzteil(Kasus2),
            % satzteil(Kasus3),
            (Kasus2 = refpron([Num1,dat,Per1]) ; Kasus3 = infzu([Num1,Per1])).
```

Alle ursprünglichen Bedingungen der Form `Formel;true` habe ich zu `Formel` vereinfacht.

6.6.3 Infinitivsätze mit zu

⟨*Grammatiken/vp.grm*⟩+≡

```
satz_infzu([infzu], _) -->
        conj([infzu],[zu]),
        v([[nom],nt],[infzu]).

satz_infzu([infzu],[Num,Per]) -->
        objekt([Kasus2], []),
        conj([infzu],[zu]),
        v([[nom, Kasus2],nt], [infzu])
        :-
        % satzteil(Kasus2),
        Kasus2 = refpron([Num,akk,Per]).

satz_infzu([infzu],[Num,Per]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        conj([infzu],[zu]),
        v([[nom, Kasus2, Kasus3],nt], [infzu])
        :-
        % satzteil(Kasus2),
        % satzteil(Kasus3),
        Kasus2 = refpron([Num,dat,Per]).
```

Bei Verben mit abtrennbarem Präfix wird die Infinitivmarkierung `zu` in das Verb eingefügt:

⟨*Grammatiken/vp.grm*⟩+≡

```
satz_infzu([infzu],_) --> v([[nom],tr],[infzu]).

satz_infzu([infzu],[Num,Per]) -->
        objekt([Kasus2], []),
        v([[nom, Kasus2],tr], [infzu])
        :-
        % satzteil(Kasus2),
        Kasus2 = refpron([Num,akk,Per]).

satz_infzu([infzu],[Num,Per]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[nom, Kasus2, Kasus3],tr], [infzu])
        :-
        % satzteil(Kasus2),
        % satzteil(Kasus3),
```

```
        Kasus2 = refpron([Num,dat,Per]).
```

⟨*Grammatiken/vpinf.grm*⟩

6.6.4 Infinitive ohne zu

(i) Reine Infinitive im Präsens, für Verben mit oder ohne abtrennbarem Präfix:

⟨*Grammatiken/vpinf.grm*⟩≡

```
vp_inf([[Kasus1]], [_Per,_Num, praes]) -->
        v([[Kasus1]], [inf]).

vp_inf([[Kasus1]], [Per1, Num1, praes]) -->
        objekt([Kasus2], []),
        v([[Kasus1, Kasus2]], [inf])
        :-
        % satzteil(Kasus2),
        (Kasus2 = refpron([Num1,akk,Per1]) ; Kasus2 = infzu([Num1,Per1])).

vp_inf([[Kasus1]], [Per1, Num1, praes]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[Kasus1, Kasus2, Kasus3]], [inf])
        :-
        % satzteil(Kasus2),
        % satzteil(Kasus3),
        (Kasus2 = refpron([Num1,dat,Per1]); Kasus3 = infzu([Num1,Per1])).
```

(ii) Reine Infinitive im Perfekt, für Verben mit oder ohne abtrennbarem Präfix:

⟨*Grammatiken/vpinf.grm*⟩+≡

```
vp_inf([[Kasus1]], [_Per,_Num, perf]) -->
        v([[Kasus1], VerbAux], [part2]),
        v([hilfsverb,VerbAux],[inf]).

vp_inf([[Kasus1]], [Per1, Num1, perf]) -->
        objekt([Kasus2], []),
        v([[Kasus1,Kasus2], VerbAux], [part2]),
        v([hilfsverb,VerbAux],[inf])
        :-
        % satzteil(Kasus2),
        (Kasus2 = refpron([Num1,akk,Per1]) ; Kasus2 = infzu([Num1,Per1])).

vp_inf([[Kasus1]], [Per1, Num1, perf]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[Kasus1,Kasus2,Kasus3], VerbAux], [part2]),
        v([hilfsverb,VerbAux],[inf])
        :-
        % satzteil(Kasus2),
        % satzteil(Kasus3),
        (Kasus2 = refpron([Num1,dat,Per1]) ; Kasus3 = infzu([Num1,Per1])).
```

⟨*Grammatiken/vpmod.grm*⟩

6.6.5 Modalsätze

Mit *Modalsätzen* sind Sätze mit einem Modalverb als Hauptverb gemeint; die Modalverben haben Infinitive ohne **zu** als Komplement:

(i) Modalsätze mit Verbzweitstellung:

(a) im Präsens:

⟨*Grammatiken/vpmod.grm*⟩≡
```
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([modal], [Per1, Num1, praes, _Modus]),
        vp_inf([[Kasus1]], [Per1, Num1, praes]).

vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([hilfsverb,haben], [Per1, Num1, _Tempus, _Modus]),
        vp_inf([[Kasus1]], [Per1, Num1,praes]),
        v([modal], [part2]).
```

(b) im Perfekt:

⟨*Grammatiken/vpmod.grm*⟩+≡
```
/*"Vermutung": ich muss den Mann abgeholt haben.*/
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([modal], [Per1, Num1, praes, _Modus]),
        vp_inf([[Kasus1]], [_Per,_Num, perf]).
```

(ii) Modalsätze mit Verberststellung

(a) im Präsens:

⟨*Grammatiken/vpmod.grm*⟩+≡
```
/*(jedoch) muss ich dir das Buch geben*/
vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([modal], [Per1, Num1, praes, _Modus]),
        np([_Def1, Per1, _Genus1], [Num1, Kasus1]),
        vp_inf([[Kasus1]], [Per1, Num1,praes]).

/*(jedoch) habe ich dir das Buch geben müssen*/
vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([hilfsverb,haben], [Per1, Num1, _Tempus, _Modus]),
        np([_Def1, Per1, _Genus1], [Num1, Kasus1]),
        vp_inf([[Kasus1]], [Per1, Num1,praes]),
        v([modal], [part2]).
```

(b) im Perfekt:

⟨*Grammatiken/vpmod.grm*⟩+≡
```
/*"Vermutung" (jedoch) muss ich den Mann abgeholt haben.*/
vp([[Kasus1]], [Per1, Num1, erst]) -->
        v([modal], [Per1, Num1, praes, _Modus]),
        np([_Def1, Per1, _Genus1], [Num1, Kasus1]),
        vp_inf([[Kasus1]], [_Per,_Num, perf]).
```

(iii) Modalsätze mit Verbletztstellung

(a) im Präsens:

⟨*Grammatiken/vpmod.grm*⟩+≡
```
/*(weil ich) dir das Buch geben muss */
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        vp_inf([[Kasus1]], [Per1, Num1,praes]),
        v([modal], [Per1, Num1, praes, _Modus]).
```

(b) im Perfekt:

⟨*Grammatiken/vpmod.grm*⟩+≡
```
/*(weil ich) dir das Buch habe geben müssen */
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        v([hilfsverb,haben],[Per1, Num1, _Temp, _Mod]),
        v([[Kasus1,Kasus2,Kasus3]], [inf]),
        v([modal], [inf]).

vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        v([hilfsverb,haben],[Per1, Num1, _Temp, _Mod]),
        v([[Kasus1,Kasus2]], [inf]),
        v([modal], [inf])
        :-
        % satzteil(Kasus2),
        (Kasus2 = refpron([Num1,akk,Per1]); Kasus2 = infzu([Num1,Per1])).

vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([hilfsverb,VerbAux],[Per1, Num1, _Temp, _Mod]),
        v([[Kasus1,Kasus2,Kasus3]], [inf]),
        v([modal], [part2])
        :-
        % satzteil(Kasus2),
        % satzteil(Kasus3),
        (Kasus2 = refpron([Num1,dat,Per1]); Kasus3 = infzu([Num1,Per1])).

/*"Vermutung"; (weil ich) dir das Buch gegeben haben muss */
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        vp_inf([[Kasus1]], [Per1,Num1, perf]),
        v([modal], [Per1, Num1, praes, _Modus]).
```

6.6.6 Zusammengesetzte Sätze

Zusammengesetzte Sätze: Einfache Sätze mit folgendem Konjunktionalsatz:

⟨*Grammatiken/vp.grm*⟩+≡
```
s(erst) -->    vp([[_Kasus]], [_Per, _Num, erst]),
                konj_satz([_Konj], []).
s(zweit) -->   subjekt([Kasus],[Per, Num]),
                vp([[Kasus]], [Per, Num, zweit]),
                konj_satz([_Konj], []).
s(letzt) -->   subjekt([Kasus],[Per, Num]),
```

```
                vp([[Kasus]], [Per, Num, letzt]),
                konj_satz([_Konj], []).
```

Satzjunktoren sollten im Lexikon die Wortart `conj` erhalten, als Unterklassifizierung werden die Verbstellung des eingebetteten Satzes und als Form die jeweilige Konjunktion benutzt. Die Einträge im Lexikon sollten also wie bei `word(wenn, conj([letzt], [wenn]))` aussehen.

⟨*Grammatiken/vp.grm*⟩+≡
```
  konj_satz([Konj], []) -->
          conj([Satzstellung],[Konj]), s(Satzstellung).
```

Für zweiteilige Junktoren wie `(je, desto|umso|je)`, deren Argumente jeweils mit einem Teil des Junktors eingeleitet werden, sind folgende Regeln vorgesehen; die Junktoren müssen aber noch ins Lexikon aufgenommen werden:

⟨*Grammatiken/vp.grm*⟩+≡
```
  konj_satz([Konj1], []) -->
          conj([Satzstellung1,[Konj2]],[Konj1]),
          s_komp(Satzstellung1),
          conj([[Konj1],Satzstellung2],[Konj2]),
          s_komp(Satzstellung2)
          :-
          % satzstellung(Satzstellung1),
          % satzstellung(Satzstellung2).
          true.
```

Für zweiteilige Junktoren, deren Teile nicht durch einen Argumentsatz getrennt werden, wie `ohne daß`, `so wie`, `so daß`, `um zu`

⟨*Grammatiken/vp.grm*⟩+≡
```
  konj_satz([[Konj1, Konj2]], []) -->
          conj([[Konj2]],[Konj1, Konj2]),
          konj_satz([Konj2], []).
```

Bem: `um zu` muß anders behandelt werden: als Konjunktion `um` mit einem `zu`-Infinitiv als Komplement.

6.6.7 Beispiele

Die Teilgrammatik `vp.grm` kann z.B. folgende Ausdrücke aus dem Text `Testbett/wissenschaftsreligion.txt` analysieren, wobei jeweils Adjektive und andere Teile des Originaltext weggelassen oder ersetzt sind, sofern sie von `vp.grm` nicht behandelt werden.

⟨*Testbett/Beispiele/vp.txt*⟩≡
```
  die Reichstagsfraktion bereitet einen Gesetzentwurf vor.
  der Gesetzentwurf sieht die Sterilisation für Straftäter vor.
  die Nazis verabschieden ein Gesetz.
  die Nazis haben ein Gesetz verabschiedet.
  die Nazis haben das Gesetz begrüßt.
  dass die Kriminalität ein Fehler ist gilt als Erkenntnis.
  ein Kriminalitätsgen ist den Nazis gelungen.
  man schreitet zu einer Sterilisation, aber man versucht einen Eingriff.
  sie ziehen Schlussfolgerungen aus dieser Vermutung.
  sie haben Schlussfolgerungen aus dieser Vermutung gezogen.
  die Autorität steht außer Zweifel so dass sie der Politik die Wahl lässt.
```

```
die Autorität hat außer Zweifel gestanden, so dass sie der Politik die Wahl gelassen hat.
eine Überwältigung scheint stattzufinden.
die Erkenntnisse haben ein Erschauern hervorgerufen.
die Erkenntnisse müssen ein Erschauern hervorgerufen haben.
die Computertechnik verknüpft die Humangenetik mit Soziobiologie.
die Computertechnik hat die Humangenetik mit Soziobiologie verknüpft.
die Möglichkeiten erscheinen als Bedrohung.
man glaubt, dass die Möglichkeiten als Bedrohung erschienen sind.
man soll die Gefahren herunterspielen.
man soll die Gefahren herunterspielen bis sie sich als ein Fehler herausstellen.
man soll die Gefahren leugnen.
man soll die Augen vor den Möglichkeiten verschließen.
man schreibt der Wissenschaft die Autorität zu.
es gilt für die Gegner.
sie wittern hinter der Forschungshypothese den Faschismus.
die Religion möchte jede Forschungshypothese verbieten.
die Kardinaltugend ist die Skepsis, denn der Zweifel motiviert zu der Überprüfung der Erkenntnisse.
bis sie sich als ein Fehler herausstellen.
die Wissenschaft beachtet die Tugend, wenn sie von den Möglichkeiten schwärmt.
die Möglichkeiten treffen auf eine Bereitschaft.
man hat lesen können.
man kann lesen.
man versteht, weil die Kunst dem Betrachter die Fragwürdigkeit seiner Weltdeutung offenbart.
die Kunst relativiert die Gewissheiten.
die Kunst hat die Gewissheiten relativieren müssen.
Eine Überwältigung der Öffentlichkeit durch die Wissenschaft scheint stattzufinden.
```

In der Datei `Testbett/Beispiele/vp.txt.vp.ana.ed.<Datum>` stehen die mit `analysierePhrasen.` und `vp.ed` hierzu erzeugten Analysen.

Anmerkungen: (H.L.)

(i) Der Satz `die Möglichkeiten sind als Bedrohung erschienen.` funktioniert mysteriöserweise nicht!

Der Verbeintrag `v([[nom,praep(als(nom))],haben],[part2]))` hat das falsche Hilfsverb, aber auch mit `haben` geht es nicht.

(ii) Komplemente von Nomina sind in `np.grm` nicht behandelt. Daher wird z.B. die Präpositionalphrase `durch die Wissenschaft` in `Eine Überwältigung der Öffentlichkeit durch die Wissenschaft` im letzten Beispiel fälschlich als Attribut an die `Öffentlichkeit` angeschlossen.

Ebenso wird der Infinitiv in `Sie treffen auf eine Bereitschaft, die Strapazen der Kritik nicht mehr auf sich zu nehmen` nicht als Komplement der `Bereitschaft` erkannt.

6.7 Fragesätze

Hier wird der Seminarbeitrag von Paula Barros und Inger Zahrenhusen so weit dokumentiert, daß er leichter mit den Teilen für die definiten Sätze und die Relativsätze zusammengefügt werden kann.

Die ursprünglichen Lexika `Seminarbeiträge/PaulaInger` habe ich mit denen in `Lexika` verglichen und die Ergänzungen separat in den Teillexika der Seminarbeiträge gelassen. Was geladen werden muß, ergibt sich aus der Datei `auswahl.hl.pl`. (Ein überflüssiges Startsymbol habe ich beseitigt, ebenso die Parameterbereiche beim Startsybol auskommentiert, und zum leichteren Testen ein `np`-Startsymbol eingefügt.) Eine Regel ist ergänzt, mit der `ob`-Sätze als Fragesätze gebildet werden und dann als Konstituenten wie andere Fragen verendet werden können.

Im Prinzip kommen die Parser (`.ed` und `.epl`) mit den Fragen zurecht. Ein paar weitere Beispielsätze sind unten angegeben und in `Testbett/Beispiele/qu.txt` mit der Ausgabed der Analysen unter `Testbett/Beispiele/qu.epl.1.2.02` angegeben.

Als Nächstes sollte in den Regeln die Konstituente `(pp([qu],Form); pp([qu(_)],Form); pp([qu(_,_)],Form))` durch eine Konstituente `pp([Def],Form)` und eine disjunktive Anwendungsbedingung `Def = qu; Def = qu(_); Def = qu(_,_)` ersetzt werden. Danach kann man die Bedingungen mit denen der definiten (und relativierenden) Sätze vereinigen.

6.7.1 Klassen- und Formparameter

Diese sind wie bei Sätzen resp. Verbalphrasen und werden hier nicht nochmal beschrieben. Allerdings wird die Definitheit von Sätzen (durch `definitheit_s/1`) anders definiert als in `s.cat.pl` vorgesehen.

In den folgenden Regeln wird allerdings die Definitheit von Sätzen nicht so wie in Abschnitt 6.10.1 geplant dargestellt, sondern ohne die Information über die erfragte Konstituente. Daher ändern wir das Prädikat vorläufig ab zu

⟨*Grammatiken/qu.grm*⟩≡
```
% Grammatik fuer Fragesaetze
:- ensure_loaded('Grammatiken/s.cat.pl'). % statt vp.cat.fragen1.pl (fehlte)

definitheit_s(qu).
definitheit_s(def).
```

Das muß aber wieder geändert werden!

6.7.2 Grammatikregeln

⟨*Grammatiken/qu.grm*⟩+≡
⟨*Grammatiken/pp1.grm*⟩
```
auxiliary_predicates([start_symbol/1,definitheit_s/1,verbstellung/1,satztempus/1,modus/1]).

start_symbol(s([Def],[Tempus,Modus,Verbstellung])).
%         :-
%         definitheit_s(Def),
%         verbstellung(Verbstellung),
%         satztempus(Tempus),
%         modus(Modus).
start_symbol(np([Def,Pers,Gen],[Num,Kas])).
```

(i) Konstituentenfragen mit einstelligem Hauptverb (i.e. nach dem Subjekt) (in Verberst-, Verbzweit- und Verbletztstellung): der Satz wird durch sein Definitheitsmerkmal `qu` als Fragesatz klassifiziert, weil eine der (optionalen) Präpositional- oder Adverbialergänzungen interrogativ sind:

⟨*Grammatiken/qu.grm*⟩+≡
```
s([qu],[Tempus,Modus,erst]) -->
        {pp([qu],[_Praep]); pp([qu(_,_,_)],[_Praep]);
         pp([qu(_,_)],[_Praep]); adv([qu], [_])},
```

```
        v([[Kasus]], [Per, Num, Tempus, Modus]),
        np([_Def, Per, _Genus], [Num, Kasus]).

s([Def],[Tempus,Modus,zweit]) -->
        np([Def, Per1, _Genus], [Num1, Kasus1]),
        v([[Kasus1]], [Per1, Num1, Tempus, Modus]).

s([Def],[Tempus,Modus,letzt]) -->
        np([Def, Per1, _Genus], [Num1, Kasus1]),
        v([[Kasus1]], [Per1, Num1, Tempus, Modus]).
```

Da die Ergänzungen in der ersten Regel optional sind, wird der Satz auch dann als interrogativ gekennzeichnet, wenn die Ergänzungen fehlen. Dann erhält man eine Satz- oder ja/nein-Frage (bei entsprechender Intonation). Das Ergebnis muß in der Satzgrammatik aber auch als Konditionalsatz beschrieben werden, wie bei `Kommt Zeit, kommt Rat.`).

(ii) Konstituentenfragen nach Präpositionalobjekten: (in Verberst-, Verbzweit- und Verbletztstellung)

⟨*Grammatiken/qu.grm*⟩+≡

```
s([qu],[Tempus,Modus,zweit]) -->
        (pp([qu],[Praep]); pp([qu(_,_,_)], [Praep]); pp([qu(_,_)], [Praep])),
        v([[nom,praep(Praep)]], [Per,Num, Tempus, Modus]),
        np([Def2, Per, _Genus], [Num, nom]).

s([qu],[Tempus,Modus,erst]) -->
        v([[nom,praep(Praep)]], [Per,Num, Tempus, Modus]),
        np([Def2, Per, _Genus], [Num, nom]),
        (pp([qu],[Praep]); pp([qu(_,_,_)],[Praep]); pp([qu(_,_)],[Praep])).

s([qu],[Tempus,Modus,letzt]) -->
        (pp([qu],[Praep]); pp([qu(_,_,_)],[Praep]); pp([qu(_,_)],[Praep])),
        np([_Def2, Per, _Genus], [Num, nom]),
        v([[nom,praep(Praep)]], [Per,Num, Tempus, Modus]).

s([def],[Tempus,Modus,letzt]) -->
        np([def, Per, _Genus], [Num, nom]),
        (pp([def],[Praep]); pp([def(_,_,_)],[Praep]); pp([def(_,_)],[Praep])),
        v([[nom,praep(Praep)]], [Per,Num,Tempus,Modus]).
```

(iii) Fragen nach dem Subjekt (und ggf. gleichzeitig nach dem Objekt) bei Verben mit Präpositionalobjekt:

⟨*Grammatiken/qu.grm*⟩+≡

```
s([qu],[Tempus,Modus,letzt]) -->
        np([qu, Per, _Genus], [Num, nom]),
        (pp([qu],[Praep]);  pp([qu(_,_,_)],[Praep]);  pp([qu(_,_)],[Praep]);
         pp([def],[Praep]); pp([def(_,_,_)],[Praep]); pp([def(_,_)],[Praep])),
        v([[nom,praep(Praep)]], [Per,Num, Tempus, Modus]).
```

(iv) Konstituentenfragen mit zweistelligen Verben:

⟨*Grammatiken/qu.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np([Def1, Per1, _Genus1], [Num1, Kas1]),
        v([[nom,Kasus2]], [Per,Num, Tempus, Modus]),
```

```
        np([Def2, Per2, _Genus2], [Num2, Kas2])
        :-
        (((nom,Per,Num)=(Kas1,Per1,Num1),(Kas2=Kasus2));
          (nom,Per,Num)=(Kas2,Per2,Num2),(Kas1=Kasus2)),
        (   (Def1=qu; Def2=qu),Def=qu);
        (not(Def1=qu; Def2=qu),Def=def).
```

In der Anwendungsbedingung der Regel wird gleichzeitig eine Vertauschung der Komplemente erlaubt:

⟨*Grammatiken/qu.grm*⟩+≡

```
s([qu],[Tempus,Modus,erst]) -->
        {pp([qu],[_Praep]); pp([qu(_,_,_)],[_Praep]);
         pp([qu(_,_)],[_Praep]); adv([qu],[_])},
        v([[nom,Kasus2]],[Per,Num, Tempus, Modus]),
        np([_Def1, Per1, _Genus1],[Num1, Kas1]),
        np([_Def2, Per2, _Genus2],[Num2, Kas2])
        :-
        (((nom,Per,Num)=(Kas1,Per1,Num1),(Kas2=Kasus2));
          (nom,Per,Num)=(Kas2,Per2,Num2),(Kas1=Kasus2)).
```

⟨*Grammatiken/qu.grm*⟩+≡

```
s([Def],[Tempus,Modus,letzt]) -->
        np([Def1, Per1, _Genus1],[Num1, Kas1]),
        np([Def2, Per2, _Genus2],[Num2, Kas2]),
        v([[nom,Kasus2]],[Per,Num, Tempus, Modus])
        :-
        (((nom,Per,Num)=(Kas1,Per1,Num1),(Kas2=Kasus2));
          (nom,Per,Num)=(Kas2,Per2,Num2),(Kas1=Kasus2)),
        ((Def1=qu;Def2=qu),Def=qu);(not(Def1=qu;Def2=qu),Def=def).
```

(v) Konstituentenfragen mit 3-stelligen Verben:

⟨*Grammatiken/qu.grm*⟩+≡

```
s([qu],[Tempus,Modus,erst]) -->
        {pp([qu],[_Praep]); pp([qu(_,_,_)],[_Praep]);
         pp([qu(_,_)],[_Praep]); adv([qu],[_])},
        v([[nom,Kasus2,Kasus3]],[Per, Num, Tempus, Modus]),
        np([_Def1, Per1, _Genus1],[Num1, Kas1]),
        np([_Def2, Per2, _Genus2],[Num2, Kas2]),
        np([_Def3, _Per3, _Genus3],[_Num3, Kas3])
        :-
        (nom,Per,Num)=(Kas1,Per1,Num1),
        ((Kas2=Kasus2,Kas3=Kasus3) ; (Kas3=Kasus2,Kas2=Kasus3)).

s([Def],[Tempus,Modus,zweit]) -->
        np([Def1, Per1, _Genus1],[Num1, Kas1]),
        v([[nom,Kasus2,Kasus3]],[Per, Num, Tempus, Modus]),
        np([Def2, Per2, _Genus2],[Num2, Kas2]),
        np([Def3, Per3, _Genus3],[Num3, Kas3])
        :-
        (((nom,Per,Num)=(Kas1,Per1,Num1),
          ((Kas2=Kasus2,Kas3=Kasus3) ; (Kas3=Kasus2,Kas2=Kasus3)));
          (nom,Per,Num)=(Kas2,Per2,Num2),
          ((Kas1=Kasus2,Kas3=Kasus3) ; (Kas1=Kasus3,Kas3=Kasus2))),
```

```
            (   (Def1=qu;Def2=qu;Def3=qu),Def=qu);
            (not(Def1=qu;Def2=qu;Def3=qu),Def=def).

    s([Def],[Tempus,Modus,letzt]) -->
            np([Def1, Per1, _Genus1],[Num1, Kas1]),
            np([Def2, Per2, _Genus2],[Num2, Kas2]),
            np([Def3, Per3, _Genus3],[Num3, Kas3]),
            v([[nom,Kasus2,Kasus3]],[Per, Num, Tempus, Modus]):-
            (((nom,Per,Num)=(Kas1,Per1,Num1),
              ((Kas2=Kasus2,Kas3=Kasus3);(Kas3=Kasus2,Kas2=Kasus3)))
            ; (nom,Per,Num)=(Kas2,Per2,Num2),
                ((Kas1=Kasus2,Kas3=Kasus3) ;(Kas1=Kasus3,Kas3=Kasus2))),
            (   (Def1=qu;Def2=qu;Def3=qu),Def=qu);
            (not(Def1=qu;Def2=qu;Def3=qu),Def=def).
```

(vi) Eingebette Fragesätze: (indirekte Fragesätze)

⟨*Grammatiken/qu.grm*⟩+≡

```
    s([def],[Tempus,Modus,zweit]) -->
            np([Def1, Per, _Genus1],[Num, nom]),
            v([[nom,conj(ob)]],[Per,Num, Tempus, Modus]),
            s([qu],[TempQ,ModQ,letzt]).

    s([qu],[Temp,Mod,letzt]) -->
             conj([letzt],[ob]), s([def],[Temp,Mod,letzt]).
```

Bem. Die letzte Regel habe ich statt der ursprünglichen Regel

⟨*Grammatik/qu.grm (ersetzt)*⟩≡

```
    s([def],[Tempus,Modus,zweit]) -->
            np([def, Per, _Genus1],[Num, nom]),
            v([[nom,conj(ob)]],[Per,Num, Tempus, Modus]),
             conj([letzt],[ob]),s([def],[TempQ,ModQ,letzt]).
```

eingeführt: der `ob`-Satz sollte ein untergeordneter Fragesatz sein, insbesondere eine (z.B. ins Vorfeld verschiebbare) Konstituente. Gleichzeitig erlaubt die Regel dann auch untergeordnete Konstituentenfragen.

In der Datei `Seminarbeitraege/PaulaInger/qu.grm` standen einige der folgenden Beispielsätze, die aber bei mir nicht alle vom Parser erkannt werden:

⟨*Testbett/Beispiele/qu.txt*⟩≡

```
  Kommt er?
  Wer kommt?
  Kommt wer?
  Wer kauft das Buch?  % nom v obj
  Begrüßt er sie?      % v nom obj
  ob er sie begrüßt?   % ob nom obj v
  Gibt er es ihr?      % v nom obj obj
  Wer gibt ihr was?    % nom v obj obj
  Wer es ihr gibt?     % nom obj obj v
  wer in der Stadt lebt.
  wer in welcher Stadt lebt.
  Wer leugnet die Gefahr?
  Wer fragt den Mann?
  Die Gefahr leugnet wer?
```

```
Der Mann fragt wen?
wer die Gefahr leugnet.
der Mann die Gefahr leugnet.
leugnet der Mann die Gefahr?
Wann begrüßt er sie?
die Freundin fragt, ob der Mann kommt.
Du fragst, wer kommt?
Ob er in München lebt?
ob er in der Stadt lebt.

% Folgende Beispiele funktionieren bei mir nicht:
In welcher Stadt lebst du.  % falls man die PP als Objekt betrachtet
Ich frage, in welcher Stadt er lebt.
Lebt er in München?
```

Mit `analysierePhrasen.` und Eingabe des Dateinamens `'Testbett/Beispiele/qu.txt'` erhält man die Analysen in einer Ausgabedatei mit angehängtem Suffix `ana.qu.<Parser>.<Datum>`.

Beispiel 6.6 Die Frage `Von welcher Freundin schwärmt der Mann?` hat folgende Analyse:

```
1. s([qu], [praes, ind, zweit])
      pp([qu(3, fem, sg)], [von])
         pr([ [dat] ], [von])
            von
         np([qu, 3, fem], [sg, dat])
            det([qu, sw], [fem, sg, dat])
               welcher
            n1([fem], [-, -, sg, dat])
               n([fem], [sg, dat])
                  Freundin
      v([ [nom, praep(von)] ], [3, sg, praes, ind])
         schwärmt
      np([def, 3, mask], [sg, nom])
         det([def, sw], [mask, sg, nom])
            der
         n1([mask], [-, -, sg, nom])
            n([mask], [sg, nom])
               Mann
```

Manche der möglichen Eingaben und Stellungsvarianten werden nicht erkannt (sowohl mit `.ed` wie mit `.epl`). Z.B. nicht die mit PP-Objekten: `Schwärmt der Mann von der Freundin?`

Die Grammtik hat offensichtlich unnötige Einschränkungen an die erlaubten Verben bei den verschiedenen Verbstellungen oder eingebetteten Fragen, die man beseitigen sollte.

6.8 Relativsätze

Der Seminarbeitrag ist noch nicht verfügbar. Die separate Grammatik sollte als Teil der Verbletztsätze integriert werden, wobei man hauptsächlich die Definitheit des Satzes von der ersten Konstituente abhängig machen muß.

Für kompliziertere Phänomene wie die Rattenfängerkonstruktionen wäre ein Abgleich mit und eine Verwendung der Teilgrammatik für Infinitivsätze erwünscht.

6.9 Gesamtgrammatik

⟨*Grammatiken/saetze.grm*⟩≡
```
:- multifile auxiliary_predicates/1.
```

⟨*Grammatiken/np.grm*⟩
⟨*Grammatiken/pp.grm*⟩
⟨*Grammatiken/vp.grm*⟩
⟨*Grammatiken/qu.grm*⟩

Damit sollte alles (ohne die AP-Grammatik und die Relativsätze) geladen werden.

Bemerkung 6.7 Für den Beispielsatz `Von welchen Gesetzen schwärmt der Mann?` braucht der Parser mit `s.grm.ed` `18,491,915 inferences in 53.91 seconds (343015 Lips)`, mit `s.grm.epl` dagegen `1,235,791 inferences in 3.37 seconds (366704 Lips)`.

Hat man *nur* die Teilgrammatik `qu.grm.ed` geladen, so werden nur `1,029,193 inferences in 2.58 seconds (398912 Lips)` und mit `qu.grm.epl` nur `588,456 inferences in 1.42 seconds (414406 Lips)` gebraucht.

6.10 Sätze (Neufassung)

Dieser Abschnitt ist der Anfang einer Überarbeitung der Grammatiken `vp.grm` *und* `qu.grm`*, bei der die* `vp`*-Knoten abgeschafft und die Definitheitsmerkmale bei Sätzen vereinheitlicht werden sollen.*

Das Parsen wird aber schon bei den wenigen bisher behandelten Regeln viel zu langsam; die Einschränkung (mit `basefirst(restrict)` bei der Grundrelation von `first/2` ist so schwach, daß der Parser langsamer wird als der `.ed`-Parser, d.h. die Filterung der Regeln ist zu schwach und kostet nur Zeit. (Mit `basefirst(explizit)`, d.h. mit den Merkmalen, platzt der Stack bei der Vorberechnung.)

6.10.1 Klassen- und Formparameter

⟨*Grammatiken/s.cat.pl*⟩≡
```
auxiliary_predicates([verbstellung/1]).

kategorie(s([Def],[Temp,Mod,Vst])) :-
        definitheit_s(Def),
        satztempus(Temp),
        modus(Mod),
        verbstellung(Vst).

restrict(s([Def],[Temp,Mod,Vst]),s([_],[_,_,Vst])).

kategorie(s([infzu], [Num,Per])) :-
        numerus(Num), person(Per).
restrict(s([infzu],[Num,Per]),s([infzu],[_,_])).
```

Klassenparameter Es genügt vorläufig ein einziger Klassenparameter, die Definitheit. Mit ihr stellen wir die üblichen Satzarten *Aussagesatz*, *Fragesatz*, *Relativsatz* und die *Infinitive* dar, die hier also als indefinite *Sätze* (statt Verbalphrasen) bezeichnet werden.

(i) Aussagesätze: sie werden durch den Definitheitswert `def` angegeben:

⟨*Grammatiken/s.cat.pl*⟩+≡
```
definitheit_s(def).
```

(ii) Relativsätze: sie werden durch das Merkmal `rel(Gen,Num)` angegeben:

⟨*Grammatiken/s.cat.pl*⟩+≡
```
definitheit_s(rel(Gen,Num)) :-
        genus(Gen), numerus(Num).
```

Aus dem Unterklassenparameter muß hervorgehen, auf mit welchen Bezugsnomen der Relativsatz zu einer Nominalphrase verbunden werden kann; daher treten Genus und Numerus im Parameter auf – die Person nur deshalb nicht, weil nur Relativsätze für die 3. Person betrachtet werden.

(iii) Fragesätze: Satzfragen, oder Fragen nach dem Wahrheitswert von Aussagen, geben wir durch den Parameter `qu(def)` an.

⟨*Grammatiken/s.cat.pl*⟩+≡
```
definitheit_s(qu(def)).
```

Von den Grammatikregeln (bzw. von `sDefinitheit` wird aber bisher nur `qu` als Definitheit ausgegeben. Daher voläufig auch:

⟨*Grammatiken/s.cat.pl*⟩+≡
```
definitheit_s(qu).
```

Fragen nach Satzgliedern werden durch den Parameter `qu(C)` mit der Kategorie `C` der erfragten Konstituente angegeben:

⟨*Grammatiken/s.cat.pl*⟩+≡
```
% definitheit_s(qu(C)) :-
%       satzglied(C).
```

Die Kategorien der Satzglieder sollten mit den bisher verwendeten Abkürzungen bezeichnet werden, also `dat` für Nominalphrasen im Dativ usw. (Vielleicht braucht man mehr Information, z.B. den Numerus, um die syntaktische Korrektheit der Antwort relativ zur Frage überprüfen zu können.)

(iv) Infinitivsätze: mit den Definitheitswerten `inf` und `infzu` unterscheiden wir reine Infinitive und die mit `zu` gebildeten Infinitive.

⟨*Grammatiken/s.cat.pl*⟩+≡
```
definitheit_s(inf).
definitheit_s(infzu).
definitheit_s(infzu(Pers,Num)) :-
        person(Pers), numerus(Num).
```

Für die Verwendung als Komplement von Kontrollverben brauchen wir Person und Numerus von Reflexivpronomina im Infinitivsatz, damit diese mit den entsprechenden Merkmalen des (durch das Kontrollverb gegebenen) impliziten Subjekts des Infinitivsatzes verglichen werden können.

(v) Imperativsätze: werden hier nicht behandelt.

(vi) Konditionalsätze: (vielleicht brauchen wir die als eigene Satzart)

Formparameter Die Formeigenschaften der finiten und der infiniten Sätze unterscheiden sich stark, so daß es vielleicht sinnvoll ist, unterschiedliche Listen von Merkmalen zu benutzen.

(i) Verbstellung:

⟨*Grammatiken/s.cat.pl*⟩+≡
```
verbstellung(erst).
verbstellung(zweit).
verbstellung(letzt).
```

(ii) Satztempus: im Unterschied zum Verbempus, wo (als finite Verbformen) nur die Werte `praes`, `praet` unterschieden werden müssen, hat der Satz weitere Tempuswerte:

⟨*Grammatiken/s.cat.pl*⟩+≡
```
satztempus(praes).
satztempus(praet).
satztempus(perf).
satztempus(plusq).
satztempus(fut).
satztempus(fut2).
```

Hiervon wird in der Grammatik nur das Präsens und das Perfekt behandelt.

(iii) Modus: Die Werte für den Indikativ und den Konjunktiv seien (wie in `v.cat.pl`)

⟨*Grammatiken/s.cat.pl*⟩+≡
```
modus(ind).
modus(konj).
```

(iv) Genus Verbi: Die Werte für Aktiv und Passiv sind: (wie in `v.cat.pl`)

⟨*Grammatiken/s.cat.pl*⟩+≡
```
genusv(aktiv).
genusv(passiv).
```

Da für das Passiv wird hier nichts implementiert wird, lassen wir den Wert des Genus Verbi unter den Formmerkmalen weg.

6.10.2 Grammatikregeln

Die Grammatik `s.grm` enthält die Regeln der Adjektivprasen nicht; sie wird unten in `sap.grm` um die Adjektivprasen erweitert.

⟨*Grammatiken/s.grm*⟩≡
```
:- multifile auxiliary_predicates/1, kategorie/1.

⟨Grammatiken/np.grm⟩
⟨Grammatiken/pp.grm⟩

:- ensure_loaded('Grammatiken/s.cat.pl').

auxiliary_predicates([start_symbol/1, expandiereCat/2]).

start_symbol(s([Def],[Temp,Mod,Vst])).

expandiereCat(Kas, np([_Def,_Per,_Genus],[_Num,Kas])) :- kasus(Kas).
expandiereCat(akk(refl), np([refl,_Per,_Genus],[_Num,akk])).
expandiereCat(dat(refl), np([refl,_Per,_Genus],[_Num,dat])).

expandiereCat(infzu([Num,Per]),s([infzu],[Num,Per])).
expandiereCat(refpron([Num,Kas,Per]),pron([refl],[unsp,Num,Kas,Per])).
expandiereCat(praep(Praep),pp(_, [Praep])).
% expandiereCat(conj(Konj),s([Konj],[Temp,Mod])).
expandiereCat(daß,s([daß],[Temp,Mod])).
expandiereCat(qu,s([ob],[Temp,Mod])).           % weitere Fälle ...
expandiereCat(qu,s([qu], [Temp, Mod, letzt])).
```

Bemerkung 6.8 Falls man in den Komplementrahmen der Verben usw. die Abkürzungen der Komplemente *typisiert*, d.h. `kas(nom),praep(an(akk)),conj(daß)` statt `nom,an(akk),daß` schreibt, wird die Fallunterscheidung nach dem Komplementrahmen in die Form der Regel verlagert, was den Parser erheblich schneller macht, aber das Lexikon und die Analysebäume mit unnötiger Information befrachtet.

Ohne die Typisierung erfolgt die Fallunterscheidung durch die Anwendungsbedingungen der Regel. Daher werden einige Operationen ausgeführt, bevor (durch `expandCat`) der jeweilige Fall vielleicht als irrelevant erkannt wird. Das verlangsamt der Parser, hält aber das Lexikon und die Analysen einfacher. Wichtig: Damit das aus der Grammatik abgelesene Prädikat `lexCat` alle relevanten lexikalischen Kategorien (bzw. deren Formen) erkennt, muß man im zweiten Fall die Kategorien durch `expandiereCat` instantiieren (damit man z.B. `mit` statt `praep(P)` in den Hilfstabellen erhält). Dazu braucht man *bei der Erzeugung des Parsers* nicht nur die Grammatik, sondern auch die Konjunktionen und Präpositionen, die als Abkürzungen in den Komplementrahmen auftreten können.
Dadurch wurde das Parsen ziemlich langsam. Auch wenn man bei der Erzeugung des Parsers mit `word/2` nach den Präpositionen sucht, zur Parsezeit aber nur mit `lexword/3`, wird der Predictor noch viel zu viele Regeln (statt Regelschemata) vorschlagen. (Es wurde aber nicht versucht, nur aus `predicted` die zu den in `lexword/3` auftretenden Komplementrahmen passenden Regeln herauszufiltern.)

⟨*zu aufwendig*⟩≡
```
:- ensure_loaded('Lexika/praepositionen.wrd').
:- ensure_loaded('Lexika/konjunktionen.wrd').

:- dynamic(lexword/3).          % noetig fuer expandiereCat!
% lexword(dummy,dummy,dummy).

expandiereCat(praep(Praep),pp(_, [Praep]))
    :- (lexword(_,_,_)
       -> lexword(_,_,pr([[Kas]],[Praep]))
       ; word(_W,pr(W,[Praep]))).
expandiereCat(conj(Konj),s([Konj],[Temp,Mod]))
    :- (lexword(_,_,_)
       -> lexword(_,Konj,conj([letzt],[Konj]))
       ; word(Konj,conj([letzt],[Konj]))). % pruefen!
```

Vor den Grammatikregeln definieren wir die darin verwendeten Prädikate zur Bestimmung der Kongruenz, Rektion und Merkmalvererbung für finite Sätze.

Kongruenz Die Kongruenzbedingung besagt, daß die Finitheitsmerkmale des Verbs, `[Pers,Num,Kas]`, und die Form des Subjekts zu einander passen. Falls das Subjekt eine Nominalphrase ist, muß sie in Person, Numerus und Kasus mit den Finitheitsmerkmalen des Verbs übereinstimmen:

⟨*Grammatiken/s.grm*⟩+≡
```
auxiliary_predicates([kongruenz_sfin/2,sDefinitheit/2,sDefinitheit/3,sDefinitheit/4]).
kongruenz_sfin(np([_Def,Pers,_Gen],[Num,Kas]),[Pers,Num,Kas]).
```

Neben der Kongruenz des Subjekts mit dem finiten Verb muß auch die grammatische Kongruenz von Objekt mit dem Subjekt in Person, und Numerus garantiert werden, wenn sich beide auf denselben Gegenstand beziehen. Das ist für die erste und zweite Person immer der Fall, aber bei der dritten Person kann das Objekt sich auf das Subjekt beziehen – dann: `sich`–, muß es aber nicht – dann: `er|sie|es|sie`.

⟨*Grammatiken/s.grm*⟩+≡
```
auxiliary_predicates([kongruenz_refl/3]).
kongruenz_refl(np([_Def,Pers,Gen],[Num,nom]),np([refl,Pers,unsp],[Num,Kas0])).
```

Falls das Verb reflexiv in einem Argument ist, muß das entsprechende Objekt als reflexiv klassifiziert sein:[24]

⟨*Grammatiken/s.grm*⟩+≡
```
kongruenz_refl(akk(refl),np([_Def,Pers,Gen],[Num,nom]),np([refl,Pers,unsp],[Num,akk])) :- !.
kongruenz_refl(dat(refl),np([_Def,Pers,Gen],[Num,nom]),np([refl,Pers,unsp],[Num,dat])) :- !.
```

Falls das Verb nicht reflexiv in seinem Argument ist, muß nur in dem Fall, wo Numerus und Person des Objekts mit dem Subjekt übereinstimmen, das Objekt reflexiv markiert sein, außer bei der 3. Person:

⟨*Grammatiken/s.grm (korrekt, aber viel zu langsam)*⟩≡
```
kongruenz_refl(Kas,np([_Def,Pers,Gen],[Num,nom]),np([Def2,Pers2,Gen2],[Num2,Kas])) :-
        person(Pers),person(Pers2),numerus(Num),numerus(Num2),
        (((Pers,Num) = (Pers2,Num2), Def2 = refl)
        ; ((Pers = 3 ; \+((Pers,Num) = (Pers2,Num2))), definitheit_np(Def2))).
```

Stattdessen benutzen wir nur:

⟨*Grammatiken/s.grm*⟩+≡
```
kongruenz_refl(Kas,np([_Def,Pers,Gen],[Num,nom]),np([Def2,Pers2,Gen2],[Num2,Kas])) :-
        definitheit_np(Def2).
```

Andere Fälle sind zu ergänzen: Subjektsätze, `es`-Subjekt.

Definitheit Die Definitheit des Satzes hängt von den Definitheitseigenschaften seiner Konstituenten und von der (hier in `sDefinitheit(Cat1+,Cat2+,Def-)` noch nicht berücksichtigten) Verbstellung ab:

⟨*Grammatiken/s.grm*⟩+≡
```
sDefinitheit(np([Def1,_,_],_),def) :-
        (Def1 = def ; Def1 = indef ; Def1 = poss).
sDefinitheit(np([Def1,_,_],_),qu) :-
        Def1 = qu.
sDefinitheit(np([rel,_,Gen],[Num,_]),rel(Gen,Num)).

sDefinitheit(np([Def1,_,_],_Form1),np([Def2,_,_],_Form2),qu) :-
        (Def1 = qu ; Def2 = qu).
sDefinitheit(np([Def1,_,_],_Form1),np([Def2,_,_],_Form2),def) :-
        (Def1 = def ; Def1 = indef ; Def1 = poss ; Def1 = refl),
        (Def2 = def ; Def2 = indef ; Def2 = poss ; Def2 = refl).
sDefinitheit(np([Def1,_,Gen1],[Num1,_]),np([Def2,_,Gen2],[Num2,_]),rel(Gen,Num)) :-
        ((Def1 = rel, Gen = Gen1, Num = Num1) ;
         (Def2 = rel, Gen = Gen2, Num = Num2)).

sDefinitheit(np([Def1,_,_],_Form1),pp([Def2],_Form2),qu) :-
        (Def1 = qu; Def2 = qu; Def2 = qu(_,_,_)).
sDefinitheit(np([Def1,_,_],_Form1),pp([DefPP],_Form2),def) :-
        (Def1 = def ; Def1 = indef ; Def1 = poss),
        (DefPP = def(_,_) ; DefPP = indef(_,_)).
sDefinitheit(np([Def1,_,Gen1],[Num1,_]),pp([Def2],_),rel(Gen,Num)) :-
        ((Def1 = rel, Gen = Gen1, Num = Num1) ; Def2 = rel(Pers,Gen,Num)).

sDefinitheit(np([Def1,_,_],_),np([Def2,_,_],_),pp([DefPP],_),def) :-
```

[24] Beispiele für Verben, die reflexiv in einem Präpositionalargument sind: *(in sich) gehen; (außer sich) sein/geraten; (zu sich) kommen; (mit sich) nehmen/führen; (von sich) geben; es (mit sich) bringen; etwas (mit sich) machen lassen.*

```
        (Def1 = def ; Def1 = indef ; Def1 = refl),
        (Def2 = def ; Def2 = indef ; Def2 = refl),
        (DefPP = def(_,_) ; DefPP = indef(_,_)).
sDefinitheit(np([Def1,_,_],_),np([Def2,_,_],_),pp([DefPP],_),qu) :-
        (Def1 = qu ; Def2 = qu ; DefPP = qu ; DefPP = qu(_,_) ; DefPP = qu(_,_,_)).
sDefinitheit(np([Def1,_,Gen1],[Num1,_]),
             np([Def2,_,Gen2],[Num2,_]),pp([Def3],_),rel(Gen,Num)) :-
        ((Def1 = rel, Gen = Gen1, Num = Num1) ;
         (Def2 = rel, Gen = Gen2, Num = Num2) ; Def3 = rel(Pers,Gen,Num)).

sDefinitheit(np([Def1,_,_],_),np([Def2,_,_],_),np([Def3,_,_],_),def) :-
        (Def1 = def ; Def1 = indef ; Def1 = refl),
        (Def2 = def ; Def2 = indef ; Def2 = refl),
        (Def3 = def ; Def3 = indef ; Def3 = refl).
sDefinitheit(np([Def1,_,_],_),np([Def2,_,_],_),np([Def3,_,_],_),qu) :-
        (Def1 = qu ; Def2 = qu ; Def3 = qu).
% sDefinitheit(np([Def1,_,Gen1],[Num1,_]),np([Def2,_,Gen2],[Num2,_]),
%            np([Def3,_,Gen3],[Num3,_]),rel(Gen,Num)) :-
%       ((Def1 = rel, Gen = Gen1, Num = Num1) ;
%        (Def2 = rel, Gen = Gen2, Num = Num2) ;
%        (Def3 = rel, Gen = Gen3, Num = Num3)).
% Relativsaetze: (provisorisch!)
```

Bem. Man muß die Definitheit der Konstituenten positiv aufzählen, nicht mit Negation darstellen (dann fehlen Lösungen) oder diese vorher durch Bereichsprädikate einschränken (dann platzt die Vorberechnung). (Fehler: `Er fragt mich.` erlaubt z.Zt. reflexive Lesart von `mich`.)

Man sollte allerdings das Prädikat so schreiben, daß die Anzahl und Form der Konstituenten keine Rolle spielt, d.h. man solle die Liste der Konstituentenkategorien durchlaufen und ihre Definitheitsmerkmale prüfen.

Konstituentenanordnung Die Regeln werden außerdem allgemeiner und übersichtlicher, wenn die Vertauschbarkeit von direkten Konstituenten durch Anordnungsregeln getrennt formuliert wird.

⟨*Grammatiken/s.grm*⟩+≡

```
auxiliary_predicates([anordnung/2]).
anordnung([A,B],[A,B]).
anordnung([A,B],[B,A]).

anordnung([A,B,C],[A,B,C]).
anordnung([A,B,C],[A,C,B]).
anordnung([A,B,C],[C,A,B]).
anordnung([A,B,C],[C,B,A]).
anordnung([A,B,C],[B,A,C]).
anordnung([A,B,C],[B,C,A]).
```

Bem. Vermutlich wird hierdurch die Anzahl der Regeln zu stark erhöht und damit das Parsen zu langsam.

Bemerkung 6.9 Wir würden gerne die Grammatikregeln allgemeiner schreiben, etwa in der Form

⟨*Wunschformat 1*⟩≡

```
s([Def],[Tempus,Modus,zweit]) -->
        Cat1,
        v([[nom, Objekt]], [Person, Numerus, Tempus, Modus]),
        Cat2
```

```
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Objekt,Cat),
        anordnung([NP,Cat],[Cat1,Cat2]),
        sDefinitheit(Cat1,Cat2,Def).
```

Man möchte auch noch offen lassen, was der Komplementrahmen des finiten Verbs enthält, z.B. wieviele Objekte das Verb hat; das entspräche einer $\bar{X}$-Regel. Solche Regeln machen aber unsere Vorberechnungen unmöglich: man kann für Variable wie `Cat1` z.B. nicht feststellen, ob es eine löschbare Kategorie ist! Dazu müßte man zur Zeit der Vorberechnung die Bedingungen (partiell) auswerten. Stattdessen könnte man den Trick aus `vp.grm` benutzen: man schreibt die Regel in der Form

⟨*Wunschformat 2*⟩≡
```
  s([Def],[Tempus,Modus,zweit]) -->
        complement(Cat1),
        v([[nom, Objekt]], [Person, Numerus, Tempus, Modus]),
        complement(Cat2)
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        anordnung([NP,Objekt],[Cat1,Cat2]),
        sDefinitheit(Cat1,Cat2,Def).
```

und benutzt Kettenregeln wie `(complement(X) --> Cat) :- expandiere(X,Cat)`. Damit sollten die Vorberechnungen möglich bleiben; notfalls entfernt man die `complement`-Knoten aus dem Baum.

Warnung: Die Grammatikvorberechnung für den `.epl`-Parser kann nur mit der Auswahl `auswahl(basefirst(restrict))` gemacht werden, andernfalls scheitert man an Stackgrenzen. Erst wenn man die Option `omitRulesWithUselessLexCat` einschaltet, wird der Parser deutlich schneller als der ohne Vorberechnungen; vermutlich kann man die Restriktion etwas genauer formulieren und dadurch noch Parsezeit auf Kosten der Vorberechnungszeit gewinnen.)

Den Rest der Grammatik unterteilen wir nach der Verbstellung in drei Teile.

⟨*Grammatiken/s.grm*⟩+≡
  ⟨*Grammatiken/svz.grm*⟩
  ⟨*Grammatiken/svl.grm*⟩
  ⟨*Grammatiken/sve.grm*⟩
```
  % :- ensure_loaded('Lexika/konjunktionen.wrd').
  % :- ensure_loaded('Lexika/praepositionen.wrd').
```

(Verbzweit- und Verbletztstellung sind teilweise gemacht; Verberststellung fehlt noch völlig).

(i) Verbzweitstellung[25]

(a) bei Verben ohne abtrennbares Präfix im Präsens und Präteritum:

⟨*Grammatiken/svz.grm*⟩≡
```
  s([Def],[Tempus,Modus,zweit]) -->
        np([Def, Per, Genus], [Num, Kas]),
        v([[Kas]], [Per, Num, Tempus, Modus]).
```

[25] Reflexive Objekte sind teilweise gemacht, aber keine reflexiven Subjekte; bei den anderen Verbstellungen fehlt beides noch völlig.

⟨*Testbett/Beispiele/s.txt*⟩≡

```
Das Gesetz kommt.
Welches Gesetz kommt?
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, Kasus]], [Person, Numerus, Tempus, Modus]),
        np(Art2, Form2)
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kasus,np(Art0,[Num0,Kas0])),
        kongruenz_refl(Kasus,NP,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,_),np(Art2,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Wer fragt ihn?
Welchen Menschen fragen wir?
Die Wissenschaft beachtet welche Tugend?
Welche Tugend beachtet welche Wissenschaft?
Die Wissenschaft beachtet die Tugend der Skepsis.
Die Nazis verabschieden ein Gesetz über die Sterilisation von Gewohnheitsverbrechern.
Wir fürchten uns.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, praep(Praep)]], [Person, Numerus, Tempus, Modus]),
        pp(Art2, Form2)
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(praep(Praep),pp(Art2,Form2)),
        sDefinitheit(np(Art1,_),pp(Art2,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Sie glauben an diese Illusion.
Welcher Mensch glaubt an eine Illusion?
Sie treffen auf eine Bereitschaft.
Wir treffen auf welche Bereitschaft?
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        pp(Art2, Form2),
        v([[nom, praep(Praep)]], [Person, Numerus, Tempus, Modus]),
        np(Art1, Form1)
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(praep(Praep),pp(Art2,Form2)),
        sDefinitheit(np(Art1,_),pp(Art2,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
An welche Illusion glaubten die Menschen?
In welcher Illusion lebt ihr?      % Lexikon?
```

```
In dieser Welt leben die Menschen.  % Lexikon?
Zu einer Überprüfung motiviert der Zweifel.
Von welcher Freundin schwärmt der Mann?
```

Objektsätze:

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, daß]], [Person, Numerus, Tempus, Modus]),
        s([daß],Form)
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(daß,s([daß],Form)),
        sDefinitheit(np(Art1,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Wir behaupten, daß alle Versuche scheitern.
Wir behaupten, dass alle Versuche scheitern.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, qu]], [Person, Numerus, Tempus, Modus]),
        s([ObQu],Form)
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(qu,s([ObQu],Form)),
        sDefinitheit(np(Art1,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Wir diskutieren, ob alle Versuche scheitern.
Wir diskutieren, welche Versuche der Hirnforscher scheitern.
```

Diese Stellung geht nicht, aber warum?

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        s([daß],[Temp,Mod]),
        v([[nom, daß]], [Person, Numerus, Tempus, Modus]),
        np(Art1, Form1)
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(daß,s([daß],[Temp,Mod])),
        sDefinitheit(np(Art1,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Daß alle Versuche scheitern, behaupten wir. % scheitert
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, Kasus, praep(Praep)]], [Person, Numerus, Tempus, Modus]),
        np(Art2, Form2),
        pp(Art3, [Praep])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
```

```
        expandiereCat(Kasus,np(Art0,[Num0,Kas0])),
        kongruenz_refl(Kasus,NP,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,_),np(Art2,_),pp(Art3,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Ich verschließe die Augen vor den Möglichkeiten.
Sie fragen mich nach dem Gesetz.
die Augen verschließe ich vor den Möglichkeiten.
Ihre Augen verschließen sie vor welchen Möglichkeiten. % 6 Analysen!
Wir begrüßen es als eine Maßnahme.
Wir treffen uns mit euch.
Ihr trefft euch mit uns.
Er äußerte sich über die Evolution.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        pp(Art3, [Praep]),
        v([[nom, Kasus, praep(Praep)]], [Person, Numerus, Tempus, Modus]),
        np(Art1, Form1),
        np(Art2, Form2)
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kasus,np(Art0,[Num0,Kas0])),
        kongruenz_refl(Kasus,NP,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,_),np(Art2,_),pp(Art3,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Vor welchen Möglichkeiten verschließe ich die Augen?
Vor welchem Gesetz fürchten sich die Hirnforscher?
Nach diesem Gesetz fragen die Gewohnheitsverbrecher wen? % nicht erkannt
Nach diesem Gesetz fragen wen die Gewohnheitsverbrecher? % nicht erkannt
Nach welchem Gesetz frage ich diese Gewohnheitsverbrecher?
Nach welchem Gesetz frage diese Gewohnheitsverbrecher ich ? % Stellung?
Als eine Maßnahme begrüßen sie es.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
% Mit 3 Nominalobjekten:
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, KasusO1, KasusO2]], [Person, Numerus, Tempus, Modus]),
        np(Art2, Form2),
        np(Art3, Form3)
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(KasusO1,np(Art0,[Num0,Kas0])),
        kongruenz_refl(KasusO1,NP,np(Art0,[Num0,Kas0])),
        expandiereCat(KasusO2,np(ArtO2,[NumO2,KasO2])),
        % kongruenz_refl(KasusO2,NP,np(ArtO2,[NumO2,KasO2])),
        anordnung([NP,np(Art0,[Num0,Kas0]),np(ArtO2,[NumO2,KasO2])],
                  [np(Art1,Form1),np(Art2,Form2),np(Art3,Form3)]),
        sDefinitheit(np(Art1,_),np(Art2,_),np(Art3,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Der Hamster kommt. Wer fragt ihn? Welchen Hamster fragen wir?
Der Hamster lebt in München. Welcher Hamster lebt in München?
Der Hamster fragt den Hamster. Ein Hamster fragt den Hamster.
In München lebt welcher Hamster? In München lebt ein Hamster.
Sie schwärmen von welchen Hamstern? Von wem schwärmt ihr?
Ich treffe auf einen Hamster.
Welche Hamster glaubten an uns?
die Kunst offenbart dem Betrachter die Fragwürdigkeit. % scheitert
Die Evolution bedient sich der Gesetze der Evolution.
```

Bem.: An der Definitheit `quant` von `kein Hamster` und der `sDefinitheit` scheitert `In München lebt kein Hamster.`.

(b) bei Verben mit abtrennbarem Präfix, im Präsens und Präteritum:
Das abtrennbare Präfix wird neben dem Komplementrahmen ein Klassenparameter; es wird im Präsens und Präteritum abgetrennt und steht nach den dem Verb folgenden Komplementen (in der Position der rechten Satzklammer):

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom],Praef], [Person, Numerus, Tempus, Modus]),
        praef([Praef], [])
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        sDefinitheit(np(Art1,Form1),Def),
        (Tempus = praes ; Tempus = praet).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Die Untersuchung findet statt.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, Kasus], Praef], [Person, Numerus, Tempus, Modus]),
        np(Art2, Form2),
        praef([Praef], [])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kasus,np(Art0,[Num0,Kas0])),
        kongruenz_refl(Kasus,NP,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,_),np(Art2,_),Def),
        (Tempus = praes ; Tempus = praet).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Sie bereiten das Gesetz vor.
Das Gesetz bereiten sie vor.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, praep(Praep)], Praef], [Person, Numerus, Tempus, Modus]),
        pp(Art2, Form2),
        praef([Praef],[])
```

```
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(praep(Praep),pp(Art2,Form2)),
        sDefinitheit(np(Art1,_),pp(Art2,_),Def),
        (Tempus = praes ; Tempus = praet).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Die Auswahl findet in der Evolution statt.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        pp(Art2, Form2),
        v([[nom, praep(Praep)], Praef], [Person, Numerus, Tempus, Modus]),
        np(Art1, Form1),
        praef([Praef],[])
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(praep(Praep),pp(Art2,Form2)),
        sDefinitheit(np(Art1,_),pp(Art2,_),Def),
        (Tempus = praes ; Tempus = praet).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
In der Evolution findet eine Auswahl statt.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, Kasus, praep(Praep)], Praef], [Person, Numerus, Tempus, Modus]),
        np(Art2, Form2),
        pp(Art3, [Praep]),
        praef([Praef],[])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kasus,np(Art0,[Num0,Kas0])),
        kongruenz_refl(Kasus,NP,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,_),np(Art2,_),pp(Art3,_),Def),
        (Tempus = praes ; Tempus = praet).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Die Gesetze sehen eine Strafe für Straftäter vor.
Eine Strafe sehen die Gesetze für Straftäter vor.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        pp(Art3, [Praep]),
        v([[nom, Kasus, praep(Praep)], Praef], [Person, Numerus, Tempus, Modus]),
        np(Art1, Form1),
        np(Art2, Form2),
        praef([Praef],[])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kasus,np(Art0,[Num0,Kas0])),
        kongruenz_refl(Kasus,NP,np(Art0,[Num0,Kas0])),
```

```
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,_),np(Art2,_),pp(Art3,_),Def),
        (Tempus = praes ; Tempus = praet).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Für Straftäter sehen die Gesetze eine Strafe vor.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([[nom, Kasus01, Kasus02], Praef], [Person, Numerus, Tempus, Modus]),
        np(Art2, Form2),
        np(Art3, Form3),
        praef([Praef], [])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kasus01,np(Art0,[Num0,Kas0])),
        kongruenz_refl(Kasus01,NP,np(Art0,[Num0,Kas0])),
        expandiereCat(Kasus02,np(Art02,[Num02,Kas02])),
        % kongruenz_refl(Kasus02,NP,np(Art02,[Num02,Kas02])),
        anordnung([NP,np(Art0,[Num0,Kas0]),np(Art02,[Num02,Kas02])],
                  [np(Art1,Form1),np(Art2,Form2),np(Art3,Form3)]),
        sDefinitheit(np(Art1,_),np(Art2,_),np(Art3,_),Def),
        (Tempus = praes ; Tempus = praet).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

Bem. Verbeinträge?

(c) bei Verben mit oder ohne abtrennbarem Präfix im Perfekt: Da in der Verbzweitstellung abtrennbare Präfix nicht abgetrennt wird[26], können beide Verbklassen mit denselben Regeln behandelt werden:

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np([Def, Person, Genus], [Numerus, Kasus]),
        v([hilfsverb,VerbAux], [Person, Numerus, Tempus, Modus]),
        v([[Kasus], VerbAux], [part2]) :-
        (Tempus = perf ; Tempus = plusq).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Das Gesetz ist gekommen.
Welches Gesetz ist gekommen?
Die Gesetze sind gekommen.
Sie waren gekommen.
Die Entwicklungen haben stattgefunden.
Die Entwicklungen hatten stattgefunden.
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        np(Art1, Form1),
        v([hilfsverb,VerbAux], [Person, Numerus, Tempus, Modus]),
        np(Art2, Form2),
```

[26] Aber die Partizipien sind unterschiedlich: `umgefallen` versus `umworben`. Ich finde, man sollte das Tempus des Hilfsverbs im Lexikoneintrag nicht zum Tempus des Hauptverbs stilisieren, sondern in der Grammatikregel das Satztempus von einem `TempusAux` ableiten! Die Hilfsverben haben zur Zeit nur Lexikoneinträge im Indikativ.

```
        v([[nom, Kas0], VerbAux], [part2])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kas0,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,_),np(Art2,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
Welche Tugend hat die Wissenschaft beachtet?
Welche Tugend hatte die Wissenschaft beachtet?
```

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,zweit]) -->
        s([infzu], [Num1,Per1]),
        v([hilfsverb,VerbAux], [Person, Numerus, Tempus, Modus]),
        np(Art, Form),
        v([[nom, infzu([Num1,Per1])], VerbAux], [part2])
        :-
        kongruenz_sfin(np(Art,Form),[Person,Numerus,nom]),
        sDefinitheit(np(Art,_),Def).

s([Def],[Tempus,Modus,zweit]) -->
        np(Art, Form),
        v([hilfsverb,VerbAux], [Person, Numerus, Tempus, Modus]),
        v([[nom, infzu([Num1,Per1])], VerbAux], [part2]),
        s([infzu], [Num1,Per1])
        :-
        kongruenz_sfin(np(Art,Form),[Person,Numerus,nom]),
        sDefinitheit(np(Art,_),Def).

% Bem: da noch keine Infinitivregeln übertragen sind:

s([infzu], [Num,Per]) -->
        conj([infzu],[zu]),
        v([[nom],nt],[infzu]).
s([infzu],_) -->
        v([[nom],tr],[infzu]).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
ich beginne ihn abzubauen.
es beginnt stattzufinden.
% ungetestet! Zu wenig Lexikoneiträge mit 'zu'-Infinitivform
```

⟨*Grammatiken/svz.grm (unfertig)*⟩≡

```
vp([[Kasus1]], [Per1, Num1, zweit]) -->
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]),
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[Kasus1, Kasus2, Kasus3], VerbAux], [part2]) :-
        % satzteil(Kasus2),
%       % satzteil(Kasus3),
        (Kasus2 = refpron([Num1,dat,Per1]) ; Kasus3 = infzu([Num1,Per1])). /* Problem! */
```

(ii) Verbletztstellung

(a) im Präsens

⟨*Grammatiken/vp.grm (noch uebernehmen!)*⟩≡

```
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        v([[Kasus1]], [Per1, Num1, _Tempus, _Modus]).

vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        v([[Kasus1, Kasus2]], [Per1, Num1, _Tempus, _Modus]).

vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[Kasus1, Kasus2, Kasus3]], [Per1, Num1, _Tempus, _Modus]).
```

(b) und im Perfekt:

⟨*Grammatiken/vp.grm (noch uebernehmen)*⟩≡

```
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        v([[Kasus1],VerbAux], [part2]),
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus]).
vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        v([[Kasus1,Kasus2],VerbAux], [part2]),
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus])
        :-
        % satzteil(Kasus2),
        (Kasus2 = refpron([Num1,akk,Per1]) ;  Kasus2 = infzu([Num1,Per1])).

vp([[Kasus1]], [Per1, Num1, letzt]) -->
        objekt([Kasus2], []),
        objekt([Kasus3], []),
        v([[Kasus1,Kasus2,Kasus3],VerbAux], [part2]),
        v([hilfsverb,VerbAux], [Per1, Num1, _Tempus, _Modus])
        :-
        % satzteil(Kasus2),
        % satzteil(Kasus3),
        (Kasus2 = refpron([Num1,dat,Per1]) ; Kasus3 = infzu([Num1,Per1])).
```

(iii) bei einstelligen Verben ohne abtrennbares Präfix hat man dieselbe Konstituentenfolge wie bei Verbzweitstellung: (unnötige Mehrdeutigkeit!)

⟨*Grammatiken/svl.grm*⟩≡

```
s([Def],[Tempus,Modus,letzt]) -->
        np([DefNP, Per, Genus], [Num, Kas]),
        v([[Kas]], [Per, Num, Tempus, Modus]) :-
        sDefinitheit(np([DefNP,_,Genus],[Num,_]),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
das Gesetz kommt.
welches Gesetz kommt?
```

Bei zwei- und mehrstelligen Verben muß das finite Verb ans Ende:

⟨*Grammatiken/svz.grm*⟩+≡

```
s([Def],[Tempus,Modus,letzt]) -->
        np(Art1, Form1),
        np(Art2, Form2),
        v([[nom, Kas0]], [Person, Numerus, Tempus, Modus])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kasus,np(Art0,[Num0,Kas0])),
        kongruenz_refl(Kasus,NP,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,Form1),np(Art2,Form2),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
wer ihn fragt?
ihn wer fragt.
ihn jemand fragt.
wir welchen Menschen fragen?
die Wissenschaft welche Tugend beachtet?
welche Tugend welche Wissenschaft beachtet?
die Wissenschaft die Tugend der Skepsis beachtet.
die Nazis ein Gesetz über die Sterilisation von Gewohnheitsverbrechern verabschieden.
```

⟨*Grammatiken/svl.grm*⟩+≡

```
s([Def],[Tempus,Modus,letzt]) -->
        np(Art1, Form1),
        pp(Art2, Form2),
        v([[nom, praep(Praep)]], [Person, Numerus, Tempus, Modus])
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(praep(Praep),pp(Art2,Form2)),
        sDefinitheit(np(Art1,Form1),pp(Art2,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
sie an diese Illusion glauben.
welcher Mensch an eine Illusion glaubt?
sie auf eine Bereitschaft treffen.
wir auf welche Bereitschaft treffen?
```

⟨*Grammatiken/svl.grm*⟩+≡

```
s([Def],[Tempus,Modus,letzt]) -->
        pp(Art2, Form2),
        np(Art1, Form1),
        v([[nom, praep(Praep)]], [Person, Numerus, Tempus, Modus])
        :-
        kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
        expandiereCat(praep(Praep),pp(Art2,Form2)),
        sDefinitheit(np(Art1,Form1),pp(Art2,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡

```
an welche Illusion die Menschen glaubten.
in welcher Illusion ihr lebt?
in dieser Welt die Menschen leben.  % ausschließen?
zu einer Überprüfung der Zweifel motiviert.
von welcher Freundin der Mann schwärmt?
```

⟨*Grammatiken/svl.grm*⟩+≡
```
s([Def],[Tempus,Modus,letzt]) -->
        np(Art1, Form1),
        np(Art2, Form2),
        pp(Art3, [Praep]),
        v([[nom, Kas0, praep(Praep)]], [Person, Numerus, Tempus, Modus])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kas0,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,Form1),np(Art2,Form2),pp(Art3,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡
```
ich die Augen vor den Möglichkeiten verschließe.
sie mich nach dem Gesetz fragen.
die Augen ich vor den Möglichkeiten verschließe.
ihre Augen sie vor welchen Möglichkeiten verschließen. % 6 Analysen!
```

⟨*Grammatiken/svl.grm*⟩+≡
```
s([Def],[Tempus,Modus,letzt]) -->
        pp(Art3, [Praep]),
        np(Art1, Form1),
        np(Art2, Form2),
        v([[nom, Kas0, praep(Praep)]], [Person, Numerus, Tempus, Modus])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kas0,np(Art0,[Num0,Kas0])),
        anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
        sDefinitheit(np(Art1,Form1),np(Art2,Form2),pp(Art3,_),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡
```
vor welchen Möglichkeiten ich die Augen verschließe.
nach diesem Gesetz die Gewohnheitsverbrecher wen fragen? % nicht erkannt
nach diesem Gesetz wen die Gewohnheitsverbrecher fragen? % nicht erkannt
nach welchem Gesetz ich diese Gewohnheitsverbrecher frage?
nach welchem Gesetz diese Gewohnheitsverbrecher ich frage? % Stellung?
```

⟨*Grammatiken/svl.grm*⟩+≡
```
% Mit 3 Nominalobjekten:
s([Def],[Tempus,Modus,letzt]) -->
        np(Art1, Form1),
        np(Art2, Form2),
        np(Art3, Form3),
        v([[nom, Kas0, Kas02]], [Person, Numerus, Tempus, Modus])
        :-
        kongruenz_sfin(NP,[Person,Numerus,nom]),
        expandiereCat(Kas0,np(Art0,[Num0,Kas0])),
        expandiereCat(Kas02,np(Art02,[Num02,Kas02])),
        anordnung([NP,np(Art0,[Num0,Kas0]),np(Art02,[Num02,Kas02])],
                  [np(Art1,Form1),np(Art2,Form2),np(Art3,Form3)]),
        sDefinitheit(np(Art1,Form1),np(Art2,Form2),np(Art3,Form3),Def).
```

⟨*Testbett/Beispiele/s.txt*⟩+≡
```
die Kunst dem Betrachter die Fragwürdigkeit offenbart.
```

(iv) bei Verben mit abtrennbarem Präfix, im Präsens und Präteritum, braucht man bei der Verbletztstellung nichts besonderes zu tun (wenn die Lexikoneinträge passen).

⟨*Grammatiken/svz.grm*⟩+≡
```
% s([Def],[Tempus,Modus,zweit]) -->
%       np(Art1, Form1),
%       v([[nom]], [Person, Numerus, Tempus, Modus]),
%       praef([Praef], [])
%       :-
%       kongruenz_sfin(np(Art1,Form1),[Person,Numerus,nom]),
%       sDefinitheit(np(Art1,Form1),Def),
%       (Tempus = praes ; Tempus = praet).

% s([Def],[Tempus,Modus,zweit]) -->
%       np(Art1, Form1),
%       v([[nom, Kas0], Praef], [Person, Numerus, Tempus, Modus]),
%       np(Art2, Form2),
%       praef([Praef], [])
%         :-
%       kongruenz_sfin(NP,[Person,Numerus,nom]),
%       expandiereCat(Kas0,np(Art0,[Num0,Kas0])),
%       anordnung([NP,np(Art0,[Num0,Kas0])], [np(Art1,Form1),np(Art2,Form2)]),
%       sDefinitheit(np(Art1,_),np(Art2,_),Def),
%       (Tempus = praes ; Tempus = praet).

% s([Def],[Tempus,Modus,zweit]) -->
%       np(Art1, Form1),
%       v([[nom, Kas0, Kas02], Praef], [Person, Numerus, Tempus, Modus]),
%       np(Art2, Form2),
%       np(Art3, Form3),
%       praef([Praef], [])
%       :-
%       kongruenz_sfin(NP,[Person,Numerus,nom]),
%       expandiereCat(Kas0,np(Art0,[Num0,Kas0])),
%       expandiereCat(Kas02,np(Art02,[Num02,Kas02])),
%       anordnung([NP,np(Art0,[Num0,Kas0]),np(Art02,[Num02,Kas02])],
%                [np(Art1,Form1),np(Art2,Form2),np(Art3,Form3)]),
%       sDefinitheit(np(Art1,_),np(Art2,_),np(Art3,_),Def),
%       (Tempus = praes ; Tempus = praet).
```

(Sehr unvollständig: Modalverben sowie infinite Komplementesätze fehlen noch!)

Konjunktionalsätze

Satzjunktoren sollten im Lexikon die Wortart `conj` erhalten, als Unterklassifizierung werden die Verbstellung des eingebetteten Satzes und als Form die jeweilige Konjunktion benutzt. Die Einträge im Lexikon sollten also wie bei `word(wenn, conj([letzt], [wenn]))` aussehen.

Die Konjunktionalsätze werden nach der einleitenden Konjunktion in Arten eingeteilt und haben als Formmerkmal das Tempus und Modus des eingebetteten Satzes:

⟨*Grammatiken/s.cat.pl*⟩+≡
```
kategorie(s([Konj],[Temp,Mod])) :-
        % word(Konj, conj(_Verbstellung,[Konj])),
```

```
        satztempus(Temp),
        modus(Mod).

restrict(s([Konj],[Temp,Mod]),s([Konj],[Temp,Mod])).
```

⟨*Grammatiken/s.grm*⟩+≡
```
start_symbol(s([Konj],[Temp,Mod])).  % zumindest zum Testen

s([Konj], [Temp,Mod]) -->
        conj([Verbstellung],[Konj]), s(_Art,[Temp,Mod,Verbstellung]).
```

⟨*Testbett/Beispiele/scomp.txt*⟩≡
```
wenn wir ihnen folgen.
ob man die Gene verändert.
weil der Krieg ausbrach.
dass Männer in der Evolution ein Verhaltensprogramm ausdifferenzieren.
```

Für zweiteilige Junktoren wie `(je, desto|umso|je)`, deren Argumente jeweils mit einem Teil des Junktors eingeleitet werden, sind folgende Regeln vorgesehen; die Junktoren müssen aber noch ins Lexikon aufgenommen werden:

⟨*Grammatiken/s.grm*⟩+≡
```
s([Konj1], [Temp,Mod]) -->
        conj([Verbst1,[Konj2]],[Konj1]),
        s(Art,[Temp,Mod,Verbst1]),
        conj([[Konj1],Verbst2],[Konj2]),
        s(Art2,[Temp,Mod,Verbst2]).
```

Diese Regel muß noch berücksichtigen, daß die Teilsätze komparierte Prädikativsätze sind!

Für zweiteilige Junktoren, deren Teile nicht durch einen Argumentsatz getrennt werden, wie `so wie`, `so daß`, `ohne daß`, `um zu`, `ohne zu`:

⟨*Grammatiken/s.grm*⟩+≡
```
s([(Konj1, Konj2)], [Temp,Mod]) -->
        conj([[Konj2]],[(Konj1, Konj2)]),
        s([Konj2], [Temp,Mod]).
```

Dazu braucht man Lexikoneinträge der Form `word(Konj1, conj([ [Konj2] ],[(Konj1,Konj2)]))`.

⟨*Testbett/Beispiele/scomp.txt*⟩+≡
```
so dass der Krieg ausbrach.
```

Bem. Damit man die Finalsätze mit Einträgen wie `word(um, conj([ [infzu] ],[um]))` und `word(ohne, conj([ [infzu] ],[ohne]))` behandeln kann, müssen allerdings die infiniten Sätze die Parameter `s([infzu],[Temp,Mod])` haben, was im Modus keinen Sinn hat.

Bem. Man behandelt diese Sätze wohl besser als Adverbialsätze!

6.10.3 Zusammengesetzte Sätze

(i) Einfache Sätze mit vorangestelltem Konjunktionalsatz:

⟨*Grammatiken/s.grm*⟩+≡
```
s([Art],[Temp,Mod,zweit]) -->
        s([Konj],[TempK,ModK]), s([Art],[Temp,Mod,erst]).
```

Bem. Solange die Verberstsätze nicht angepasst sind, kann man das nicht testen.

(ii) Einfache Sätze mit nachgestelltem Konjunktionalsatz:

⟨*Grammatiken/s.grm*⟩+≡
```
  s([Art],[Temp,Mod,Vst]) -->
          s([Art],[Temp,Mod,Vst]), s([Konj],[TempK,ModK]).
```

Für Fragesätze ist hier die Verberststellung nützlich. Aber soll man auch die Verbletzstellung so behandeln? (Oder nicht eher als die Verschiebung eines Adverbialsatzes in das Nachfeld des Verbgletztsatzes?)

(iii) Koordinierte Sätze:

Bem. Vermutlich sollten komplexe Sätze besser nicht dieselbe Kategorie wie Basissätze erhalten. Dann würde automatisch ausgeschlossen, daß man wiederholt periphere Konjunktionalsätze angliedern kann.

Zum Abschluß erweitern wir die Satzregeln auch um die Konstruktionen für Adjektivphrasen:

⟨*Grammatiken/sap.grm*⟩≡
  ⟨*Grammatiken/s.grm*⟩
  ⟨*Grammatiken/ap.grm*⟩

Allerdings haben die in `ap.grm` definierten Prädikativsätze noch nicht die in `s.grm` festgelegten Merkmale.

7 Vorläufiges Fazit

Es fehlte leider die Zeit, die Grammatik vollständig zu vereinheitlichen und zu testen. Nach dem Bisherigen zeichnet sich folgendes Ergebnis ab:

(i) Der `.ed`-Parser ist bei kleineren Grammatiken gut genug, sofern die Bereichsprädikate in den Regeln weggelassen werden können.

Die Analysezeiten sind aber wesentlich besser, wenn man die Grammatik für den `.epl`-Parser vorberechnet.

(ii) Die Vorberechnungen für den `.epl`-Parser zeigen:

(a) Falls die Grammatik klein ist, kann man mit expliziten Grundrelationen (d.h. mit belegten Merkmalvariablen) die First-Relation und damit den statischen Predictor berechnen.

(b) Falls die Grammatik größer wird, muß man die Merkmale als freie Variable behandeln. Die Vorberechnung der transitiven Hülle der linken-Ecke-Beziehung der Regeln ist hier mit einer Modifikation des Warshall-Algorithmus berechnet, die nicht effizient mit dem Speicher umgeht.

Die Versionen 2 und 3 aus Abschnitt 9.1 sind unvollständig, die Version 4 könnte korrekt sein, was aber nicht sicher ist. Die Berechnungen scheitern am Stack, wenn die Grundrelation schon bescheiden groß ist. Bei der Berechnung ist keine Vorsorge getroffen, schon behandelte Knoten des Graphen nicht nocheinmal zu behandeln; das sollte einer der Gründe dafür sein, daß die Berechnung zu aufwendig wird. Beispiele zeigen, daß der Termgraph bei der Konstruktion der transitiven Hülle auf die doppelte Knotenzahl anwachsen kann, da man beim Verbinden von Kanten ggf. neue Instanzen vorhandener Knoten bilden muß.

Die alternative induktive Berechnung der transitiven Hülle kommt nicht so schnell an Speichergrenzen.

Sofern man den statischen Predictor noch berechnen kann, wird er mit wachsender Anzahl von Merkmalen immmer größer (6 MB für eine Version der Adjektivgrammatik). Bei `np`, `n1` und `n` habe ich weitere Merkmale beschränkt.

(c) Bei der Erstellung der Grundrelation der Linke-Ecke-Beziehung in den Regeln kann man mit `basefirst(restrict)` die Merkmale wegprojizieren; dann läßt sich die transitive Hülle (bei der `ap`-Grammatik) noch berechnen, sowohl mit `warshall-4` wie induktiv.

Mit `predictor(uniform)` werden die als brauchbar vorausgesagten Regeln bei der Vorberechnung nicht von einer Erwartungs- und einer lexikalischen Kategorie (des nächsten Worts) abhängig gemacht, sondern es werden nur die löschbaren Präfixe der rechten Seiten bestimmt und dann zur Laufzeit (oder, mit der Option `ein(predictUniform)`, bei der Vorberechnung) mit der `first`-Relation die brauchbaren Teile durch den (dynamischen) Predictor ausgesondert.

Die Berechnungen der transitiven Hülle im Anhang sind bei den größeren Grammatiken alle untauglich, wenn man nicht die Grundrelation durch Wegprojizieren der Merkmale massiv vereinfacht.

(iii) Ein Vergleich der beiden Parser bei größeren Grammatiken ist noch nicht gemacht. Bei der `vp`-Grammatik sind teilweise die `.ed`, teilweise die `.epl`-Versionen 'schneller', aber die Grammatikversionen sind nicht identisch, und zum Teil sind verschiedene Sätze analysierbar.

Man muß zuerst mit den Restriktoren in `compute_first_basecase` experimentieren und die Gesamtgrammatik konsistent machen.

(iv) Für größere Grammatiken ist es unbedingt nötig, daß man die Vorberechnung der brauchbaren Regeln (Top-Down-Filter) nicht nur vom nächsten Eingabesymbol abhängig macht.

Die Grammatik `s.grm` hat in den Regeln Permutationen der Konstituenten, und das scheint die Voraussage der brauchbaren Regeln (zur Parsezeit) stark negativ zu beeinflussen; auch die Belegung der Merkmale, die zur Bestimmung der Definitheit des Satzes notwendig sind, könnte eine Rolle spielen.

Die Vorberechnung der brauchbaren Regeln kann mit der Option `omitRulesWithUseLessLexCat` auch von der Menge der für die restlichen Eingabewörter gefundenen lexikalischen Kategorien abhängig gemacht werden. Das bewirkt bei der Grammatik `s.grm` eine Beschleunigung um den Faktor 80 gegenüber dem normalen Predictor, und eine Beschleunigung um den Faktor 40 gegenüber dem Earley-Dörre-Parser.

Man könnte hiervon noch Varianten testen: a) die Voraussage nur von den Kategorien noch nicht verarbeiteter Eingabewörter abhängig machen, die einen Komplementrahmen tragen, d.h. von Verben, nominalisierten Verben, Adjektiven etc. b) bei der Voraussage auch die Anordnung der lexikalischen Kategorien in der Regel und in der Resteingabe berücksichtigen.

Die Analyse mit `s.grm` zeigt jedenfalls, daß vom Predictor ohne die Option `omitRulesWithUseLessLexCat` viele Regeln mit mehrstelligen Verben auch dann vorgeschlagen werden, wenn nur einstellige Verben in der Eingabe vorkommen. (Zur Zeit werden allerdings einige Regeln entfernt, weil die Grammatik z.B. keine Verbkategorie mit bestimmtem Komplementrahmen kennt. Es kann sein, daß die Beschleunigung weniger stark wird, wenn mehr der Kategorien mit Komplementrahmen in der Grammatik auftreten.)

8 Transformationsregeln

Zu der im vorigen Kapitel angegebenen Grammatik des Deutschen sollen nun Transformationsregeln geschrieben werden, mit denen sich analysierte Phrasen in andere übersetzen lassen. Im allgemeinen ist die Übersetzung eine Relation, keine Funktion; das läßt sich aber durch Prolog-Klauseln gut wiedergeben.

Die Transformation einer komplexen Phrase erfolgt rekursiv nach dem Aufbau der Phrase, d.h. die durch den Parser gefundene syntaktische Analyse wird von der Wurzel zu den Blättern des Analysebaums durchlaufen. Jede Phrase wird nach einer Grammatikregel in ihre direkten Konstituenten zerlegt; nach der Transformation der Konstituenten werden diese nach einer (normalerweise) anderen Grammatikregel zusammengesetzt.

Normalerweise wird dabei eine atomare Konstituente durch Flexion in eine andere Wortform desselben Wortes übersetzt. Daher braucht man auf jeden Fall ein Flexionsprogramm für Wörter.

8.1 Flexionsprogramme für Wörter

Für unflektierbare Wortarten wird das Vollformenlexikon `word/2` beibehalten, dessen Einträge das Format `word(Vollform,Kategorie)` haben.

Für flektierbare Wortarten enthält das Vollformenlexikon `word/3` zu jeder Vollform einen Eintrag `word(Vollform,Kategorie,Stammform)` mit einer zugehörigen Stammform.

Jede Wortart hat nach ihrer Definition ein bestimmtes Format von Art- und Formmerkmalen und vorgegebenen Werten dafür. Das muß dem Flexionsprogramm bekannt sein:

⟨*Transformationen/paradigma.pl*⟩≡
```
:- ensure_loaded('Grammatiken/np.cat.pl').
:- ensure_loaded('Grammatiken/pp.cat.pl').
:- ensure_loaded('Grammatiken/ap.cat.pl').
:- ensure_loaded('Grammatiken/s.cat.pl').
```

Aus der Stammform und dem Paradigma der Wortart erhalten wir leicht ein (natürlich nicht gerade effizientes) Flexionsprogramm für flektierbare Wörter: [27]

⟨*Transformationen/paradigma.pl*⟩+≡
```
flektiere([Kat1,Vollform1], Form2, [Kat2,Vollform2]) :-
        Kat1 =.. [Cat,Art,Form1],
        ((Cat = v; Cat = a) -> ArtNeu = [] ; ArtNeu = Art),
        Kat1Neu =.. [Cat,ArtNeu,Form1],
        word(Vollform1, Kat1Neu, Stamm),
        Kat2 =.. [Cat,Art,Form2],
        Kat2Neu =.. [Cat,ArtNeu,Form2],
        word(Vollform2, Kat2Neu, Stamm).
```

Als Format der Eingabe wird hier das Ausgabeformat des Parsers benutzt, damit `flektiere/3` von den folgenden Phrasentransformationsprogrammen benutzt werden kann.

[27] Da bei Verben (ebenso einzubauen: für andere Wortarten, deren Artparameter mit dem Stammlexikon zusammengesetzt werden!) die Artparameter teilweise nicht in `word/3`, sondern nur in `word/2` stehen, muß man etwas umständlich die Art ignorieren.

Um das vollständige Paradigma einer Wortart (oder den jeweils implementierten Teil des Paradigmas) zu testen, d.h. ein eingegebenes Wort in alle vorhandenen Formen zu setzen, benutzen wir die Spezifikation `wortkategorie/1` der Wortart:

⟨*Transformationen/paradigma.pl*⟩+≡
```
wortparadigma :-
        write('Eingabe einer Vollform (mit <Punkt><Return> beenden):'),nl,
        read_sentence(user,Eingabewort,[]),
        stringToAtoms(Eingabewort,WortfolgeSz),
        entferneSatzzeichen(WortfolgeSz,Wortfolge),
        lexAnalyse(Wortfolge),
        (ein(pruefeLexWord) -> erkenneWortarten ; true),
        (gescheitert(lexAnalyse)
        -> write(user,'\nEs wird kein Paradigma erzeugt. ')
        ; lexword(_,W,Kat),
            Kat =.. [Cat,Art,_],
            (Cat = v -> ArtNeu = [] ; ArtNeu = Art),
            KatNeu =.. [Cat,ArtNeu,_],
            word(W,KatNeu,Stamm),
            % !, % erstmal nur die erste Wortart testen
            Kategorie =.. [Cat,Art,Form],
            repeat,
            ((wortkategorie(Kategorie),
              KategorieNeu =.. [Cat,ArtNeu,Form],
              word(Vollform,KategorieNeu,Stamm),
              write(user,Kategorie),write(user,':  '),
              write(user,Vollform), write(user,'\n'),
              fail)
            ; true)
        ).
```

Beachte, daß die Artparameter aus den Einträgen von `lexword/3` aus `word/2` stammen, also die Artparameter aus `word/3` um die Komplementrahmen erweitern. Das wird oben für Verben kompensiert.

Fehler:

(i) Nomen: Sobald man Komplementrahmen auch bei Nomen erlaubt, muß das wie für Verben kompensiert werden, da die Artparameter nicht in `word/3` aufgeführt sind.

(ii) Pronomen: Bei Personal- und Interrogativpronomen erhält man nur ein Teilparadigma, da bei der Wortkategorie das Genus `unsp` nicht erlaubt wurde. Das muß man also ändern. (Im

(iii) Determinator: Da z.B. zwischen dem Artikel `der` und dem Demonstrativpronomen `der` nicht unterschieden wird, erhält man im Dativ Plural zwei Formen:

⟨*Beispiel*⟩≡
```
81 ?- wortparadigma.
Eingabe einer Vollform (mit <Punkt><Return> beenden):
|: der.
det([def, sw], [mask, sg, nom]):  der
...
det([def, sw], [unsp, pl, dat]):  den
det([def, sw], [unsp, pl, dat]):  denen
det([def, sw], [unsp, pl, akk]):  die
```

(iv) Verb (und Adjektiv): In `wortparadigma/0` wird kompensiert, daß der Komplementrahmen aus dem Stammlexikon ergänzt wird. Man sollte das Paradigma (mindestens in der Anzeige) nicht für alle Komplementrahmen zeigen.

Bisher wird nur eine lexikalische Lesart des Eingabeworts genommen; gelegentlich gibt es mehrere. Daher ist `!` in `wortparadigma/0` geeignet zu beseitigen.

8.2 Transformation von Phrasen

Das *Phrasenparadigma* einer Phrase soll aus all den Varianten der Phrase bestehen, die sich durch andere Wahl der Formmerkmale der Phrase ergeben. Als Varianten gelten die Phrasen, die zur Eingabephrase in einer der unten definierten Transformationsrelationen stehen.

Zum Testen der Transformationsrelationen dient das folgende Prädikat `phrasenparadigma/0`, das alle Tranformationen einer Phrase anzeigt:

⟨*Transformationen/paradigma.pl*⟩+≡

```
phrasenparadigma :-
        write('Eingabe einer Phrase (mit <Punkt><Return> beenden):'),nl,
        read_sentence(user,Eingabe,[]),
        stringToAtoms(Eingabe,WortfolgeSz),
        entferneSatzzeichen(WortfolgeSz,Wortfolge),
        lexAnalyse(Wortfolge),
        (ein(pruefeLexWord) -> erkenneWortarten ; true),
        (gescheitert(lexAnalyse)
        -> (nl, write(user,'Keine syntaktische Analyse durchgeführt.'))
        ; ((baeume(Wortfolge,TreesW),
            baeume_pro_kategorie(TreesW,Trees))
          -> (transformationen(Trees,TreesTr)
             -> (ein(showTrees)
                -> nl, write(user,'Analyse(n) der Eingabe, pro Kategorie:'),
                    nl, showASCIItrees(Trees), nl,
                    nl, write(user,'Phrasenparadigma der Eingabe:'),
                    nl, showASCIItrees(TreesTr), nl
                ;  nl, blattfolgen(TreesTr,Phrasen),
                   writePhrasen(user,Phrasen))
             ; (nl, write(user,'Keine Transformation der Eingabe gefunden.')))
          ; (nl, write(user,'Keine syntaktische Analyse der Eingabe gefunden.'))
          )
        ).

baeume_pro_kategorie([[Kat|Args]|Trees],Bs) :-
        baeume_pro_kategorie(Trees,TreesBs),
        Kat =.. [Cat,Art,_],
        Tr =.. [Cat,Art,_Form],
        (memberchk([Tr|_],TreesBs) -> Bs = TreesBs ; Bs = [[Kat|Args]|TreesBs]).
baeume_pro_kategorie([],[]).

transformationen([Tree|Trees],Transformiert) :-
        setof(Tr, baumTr(Tree,Tr), TreeTr),
        transformationen(Trees,TreesTr),
        append(TreeTr,TreesTr,Transformierte),
        reverse(Transformierte,Transformiert).
```

```
transformationen([],[]).

blattfolgen([T|Ts], [B|Bs]) :-
        blattfolge(T,B),
        blattfolgen(Ts,Bs).
blattfolgen([],[]).

blattfolge([Wurzel|Teile],(Wurzel,Phrase)) :-
        blattfolge([Wurzel|Teile],[],Atome),
        reverse(Atome,Reversed),
        concat_atom(Reversed,' ',Phrase).

blattfolge([Wurzel|Teile],Acc,Atome) :-
        wortkategorie(Wurzel),
        !, Teile = [Wort],
        Atome = [Wort|Acc].

blattfolge([Wurzel],Acc,Atome) :-
        !, Atome = Acc.

blattfolge([Wurzel,T|Ts],Acc,Atome) :-
        !, blattfolge(T,Acc,AccT),
        blattfolge([Wurzel|Ts],AccT,Atome).

writePhrasen(Strom,[(Kat,Phrase)|Rest]) :-
        write(Strom,Kat),write(Strom,': '),
        write(Strom,Phrase),write(Strom,'\n'),
        writePhrasen(Strom,Rest).
writePhrasen(Strom,[]).
```

Die eigentlichen Transformationen sind Relationen zwischen Analysebäumen. An der Wurzel des Baums liest man die Kategorie und ihre Merkmale ab, wobei die Formmerkmale ignoriert werden.

Falls die Wurzel eine Phrasenkategorie ist, werden mit Hilfe der Definition `kategorie/1` der Phrasenkategorien alle möglichen Werte der Formmerkmale durchlaufen, und jeweils die Varianten der entsprechenden Form berechnet:

⟨*Transformationen/paradigma.pl*⟩+≡
```
baumTr([Wurzel|Teile],Tr) :-
        ground(Wurzel),
        kategorie(Wurzel),
        !,
        Wurzel =.. [Cat,Art,_],
        Variante =.. [Cat,Art,Form],
        kategorie(Variante),
        transformiere([Wurzel|Teile],Form,Tr).
```

Falls die Wurzel eine Wortkategorie ist, erhält man die möglichen Formmerkmale über die Definition `wortkategorie/1` der Wortkategorien.

⟨*Transformationen/paradigma.pl*⟩+≡
```
baumTr([Wurzel|Teile],Tr) :-
        ground(Wurzel),
        wortkategorie(Wurzel),
        !,
```

```
        Wurzel =.. [Cat,Art,_],
        Kat =.. [Cat,Art,Form],
        wortkategorie(Kat),
        flektiere([Wurzel|Teile],Form,Tr).
```

In den anderen Fällen wird eine Fehlermeldung ausgegeben:

⟨*Transformationen/paradigma.pl*⟩+≡

```
  baumTr([Wurzel|Teile],Tr) :-
        !,
        (ground(Wurzel)
        -> write(user,'\nFehler in baumTr/2: die Wurzelkategorie '),
            write(Wurzel), write(' ist nicht deklariert. ')
        ; write(user,'\nFehler in baumTr/2: die Wurzelkategorie enthält Variable. ')),
        Tr = Wurzel.
  baumTr(Atom,Tr) :-
        !, write(user,'Fehler in baumTr/2: die Wurzel ist atomar. '),
        Tr = Atom.
```

Nun müssen die Transformationsrelationen für jede Phrasenart entsprechend den Grammatikregeln definiert werden.

⟨*Transformationen/paradigma.pl*⟩+≡
  ⟨*Paradigma der Nominalphrasen*⟩
  ⟨*Paradigma der Adjektivphrasen*⟩
  ⟨*Paradigma einfacher Sätze*⟩

Schließlich muß in allen Fällen, wo die Wurzel des Analysebaums eine Wortkategorie ist, die gegebene Vollform des Worts durch `flektiere/3` in die gewünschte Form gesetzt werden:

⟨*Transformationen/paradigma.pl*⟩+≡

```
  transformiere([Kategorie,Vollform], Form, Tr) :-
        wortkategorie(Kategorie),
        !,
        flektiere([Kategorie, Vollform], Form, Tr).
```

8.2.1 Transformation einer Nominalphrase in eine andere Form

Die Numerierung der einzelnen Fälle sollte der Numerierung der Regeln der Grammatik `np.grm` entsprechen. Die Regeln aus `Seminarbeitraege/Voeltz/transformationen.av.pl` werden hier so wiedergegeben, daß Regeln mit optionalen Konstituenen und Attributen möglichst nicht in Einzelfälle aufgespalten werden:

(i) `np -> det n1 {s_rel}`:

⟨*Paradigma der Nominalphrasen*⟩≡

```
  transformiere([np(Art, [Num1, Kas1]),
                 [det(ArtDet, [GenDet, Num1, Kas1]), Det], % i.a. GenDet =/= Gen!
                 [n1([Gen], [Komp, Adekl, Num1, Kas1])|N]
                 | Relsatz
                ], [Num2, Kas2], Tr) :-
        !,
        (Num2 = pl
        -> (GenDet2 = GenDet ; GenDet2 = unsp)
```

```
      ;   GenDet2 = Gen
      ),
      transformiere([det(ArtDet, [GenDet, Num1, Kas1]), Det],
                    [GenDet2, Num2, Kas2], DetTr),
      transformiere([n1([Gen], [Komp, Adekl, Num1, Kas1])|N],
                    [Komp, Adekl, Num2, Kas2], NTr),
      anschluss(Relsatz,(GenDet,Num2),RelsatzTr),
      Tr = [np(Art, [Num2, Kas2]), DetTr, NTr | RelsatzTr ].
```

Falls kein Relativsatz vorhanden war, entfällt die Anpassung.

⟨*Paradigma der Nominalphrasen*⟩+≡

```
  anschluss([],_,[]) :- !.
```

Der Relativsatz muß seinerseits noch transformiert werden, wenn sich das Bezugsnomen im Numerus (und im Genus, wenn es ein Pronomen war) ändert. Dann ist die relativierende Konstituente anzupassen:

⟨*Paradigma der Nominalphrasen*⟩+≡

```
  anschluss([[s([rel(Gen1,Num1)],Form) | Rest]],(Gen2,Num2),[RelsatzTr]) :-
          !,anschliessen(Rest,(Gen2,Num2),RestTr),
          RelsatzTr = [s([rel(Gen2,Num2)],Form) | RestTr].
  anschluss([np([rel,Pers1,Gen1],[Num1,Kas1]),
                   [pron([rel],[Gen1,Num1,Kas1]), P]],
                  (Gen2,Num2),
                  [np([rel,Pers1,Gen2],[Num2,Kas1]), Pron2]) :-
          !,(Num2 = pl
          -> flektiere([pron([rel],[Gen1,Num1,Kas1]),P],[unsp,Num2,Kas1],Pron2)
          ; flektiere([pron([rel],[Gen1,Num1,Kas1]),P],[Gen2,Num2,Kas1],Pron2)).
  anschluss([pp([rel(Pers1,Gen1,Num1)],[Praep]),
               [pr([[Kas]],[Praep]),Pr],
               [np([rel,Pers1,Gen1],[Num1,Kas1])|NP]
              ],(Gen2,Num2),
              [pp([rel(Pers1,Gen2,Num2)],[Praep]),
               [pr([[Kas]],[Praep]),Pr],
               NpTr]) :-
          anschluss([np([rel,Pers1,Gen1],[Num1,Kas1])|NP],(Gen2,Num2),NpTr).
  % Postpositionen?
```

Falls das Relativpronomen im Nominativ stand und der Numerus wechselt, muß auch das Verb im Numerus angepaßt werden:

⟨*Paradigma der Nominalphrasen*⟩+≡

```
  anschliessen([[pp([rel(Pers,Gen,Num)],[Praep])|PP]|Cs],(Gen2,Num2),[PpTr|Cs]) :-
          !, anschluss([pp([rel(Pers,Gen,Num)],[Praep])|PP],(Gen2,Num2),PpTr).
  anschliessen([[np([rel,Pers,Gen],[Num,Kas]) | NP]|Cs],(Gen2,Num2),[NpTr|CsTr]) :-
          !, anschluss([np([rel,Pers,Gen],[Num,Kas]) | NP],(Gen2,Num2),NpTr),
          ((Kas = nom, \+ (Num2 = Num))
          -> anschliessen(Cs,(Gen2,Num2),CsTr)
          ; CsTr = Cs).
  anschliessen([C|Cs],(Gen2,Num2),[CTr|CsTr]) :-
          (C = [v(Art,[Pers,Num,Temp,Mod]), V]
          -> flektiere(C, [Pers,Num2,Temp,Mod], CTr)
          ; CTr = C),
          anschliessen(Cs,(Gen2,Num2),CsTr).
  anschliessen([],_,[]).
```

(ii) `n1 -> ap, (n ; en ; en_unsp), {np_gen ; pp}`

Die Adjektivphrase und das Kopfnomen oder der entsprechende Eigenname werden transformiert. Falls ein nachgestelltes Attribut vorhanden ist, wird es unverändert[28] übernommen:

⟨*Paradigma der Nominalphrasen*⟩+≡

```
transformiere(
    [
        n1([Gen], [Komp, Adekl, Num1, Kas1]),
        [ap([Komp, Adekl, GenA, Num1, Kas1])|A],
        Kopf
        | Attribut
    ],
    [Komp2, Adekl2, Num2, Kas2], Tr) :-
    kongruenzGenusNP(Gen,GenA2),
    transformiere([ap([Komp, Adekl, GenA, Num1, Kas1])|A],
                  [Komp2, Adekl2, GenA2, Num2, Kas2], ATr),
    (Kopf = [n([Gen], [Num1, Kas1]), N]
    ; Kopf = en([Gen],[Num1,Kas1])
    ; Kopf = en([unsp],[Num1,Kas1])),
    transformiere(Kopf, [Num2, Kas2], KopfTr),
    % (Attribut = [np([Def, 3, GenNP], [NumNP, gen])|NP]
    % ; Attribut = [pp([DefPP], [Praep])|PP]
    % ; Attribut = []),
    Tr = [n1([Gen], [Komp2, Adekl2, Num2, Kas2]), ATr, KopfTr | Attribut].
```

Durch `kongruenzGenusNP` wird erlaubt, daß das (un)spezifische Genus `GenA` der Eingabe in ein unspezifisches (spezifisches) Genus `GenA2` umgewandelt wird. Da es im Singular von der Adjektivdeklination und dem Kasus abhängt, ob das Genus spezifisch oder unspezifisch ist, ist das vorgefundene `GenA` nicht immer auch das Genus der transformierten Adjektivphrase.[29]

Man könnte sicherheitshalber testen, ob das Attribut von der gewünschten Form ist.

(iii) `n1 -> n {np_gen ; pp}`

Das Nomen wird transformiert; falls es ein (nominales oder präpositionales) nachgestelltes Attribut hat, wird dieses unverändert übernommen:

⟨*Paradigma der Nominalphrasen*⟩+≡

```
transformiere([
               n1(Art, [-, -, Num1, Kas1]),
               [n(Art, [Num1, Kas1]), N]
               | Attribut
              ],
              [-, -, Num2, Kas2], Tr) :-
    transformiere([n(Art, [Num1, Kas1]), N], [Num2, Kas2], NTr),
    % (Attribut = [np([Def, 3, GenNP], [NumNP, gen])|NP]
    % ; Attribut = [pp([DefPP], [Praep])|PP]
    % ; Attribut = []),
    Tr = [n1(Art, [-, -, Num2, Kas2]), NTr | Attribut].
```

[28] Vielleicht muß man Reflexivpronomina im Attribut anpassen!

[29] Bisher fehlen noch die Transformationen der Eigennamen `en`.

Man könnte sicherheitshalber testen, ob das Attribut von der gewünschten Form ist.

(iv) `n1 -> na`

⟨*Paradigma der Nominalphrasen*⟩+≡
```
  transformiere(
      [
          n1([Gen], [Komp, Adekl, Num1, Kas1]),
          [na([Gen], [Komp, Adekl, Num1, Kas1]), NA]
      ],
      [Komp2, Adekl2, Num2, Kas2], Tr) :-
      transformiere([na([Gen], [Komp, Adekl, Num1, Kas1]), NA],
                    [Komp2, Adekl2, Num2, Kas2], NATr),
      Tr = [n1([Gen], [Komp2, Adekl2, Num2, Kas2]), NATr].
```

(v) `n1 -> {n}, en`

(vi) `np_indef -> n1_plural`

⟨*Paradigma der Nominalphrasen*⟩+≡
```
  transformiere(
      [
          np([indef, 3, Gen], [pl, Kas1]),
          [n1([Gen], [Komp, Adekl, pl, Kas1]) | N1]
          | RelSatz
      ],
      [Num2, Kas2], Tr) :-
      transformiere([n1([Gen], [Komp, Adekl, pl, Kas1]) | N1],
                    [Komp, Adekl, Num2, Kas2], N1Tr),
      Tr = [np([indef, 3, Gen], [Num2, Kas2]), N1Tr | RelSatz].
```

Fehler: Der Relativsatz kann nicht unverändert übernommen werden, wenn der Numerus des Bezugsnomens sich ändert.

(vii) `np -> pron`:

⟨*Paradigma der Nominalphrasen*⟩+≡
```
  transformiere(
      [
          np(Art, [Num1, Kas1]),
          [pron(ArtPron, [Gen, Num1, Kas1, Pers]), W]
      ],
      [Num2, Kas2], Tr) :-
      !,
      transformiere([pron(ArtPron, [Gen, Num1, Kas1, Pers]), W],
                    [Gen, Num2, Kas2, Pers], WTr), % ggf Gen von Num abhängig machen!
      Tr = [np(Art, [Num2, Kas2]), WTr].
```

Eine Transformation mir Relativsatz beim Interrogativpronomen fehlt noch.

(viii) `np -> np, conj, np`

(ix) `np -> anrede, en`

(x) `np -> en`

(xi) `np -> vn`

(xii) `en -> {titel}, nn`

(xiii) `en -> {titel}, vn, nn`

Getrennte Kasus- und Numerustransformation von Nominalphrasen

Aus dem Flexionsprogramm `transformiere/3` erhält man speziellere Transformationen, bei denen gewisse Formparameter der Eingabe nicht verändert werden.

Beispiel 8.1 Wir leiten Das Programm `beuge(+KasusNumerus)` fordert den Benutzer zur Eingabe einer Nominalphrase auffordert; diese wird durch den Parser analysiert und schließlich werden die gefundenen Analysen in den gewünschten Kasus bzw. Numerus, `KasusNumerus`, umgerechnet:

⟨*Transformationen/paradigma.pl*⟩+≡

```
beuge(KasusNumerus) :-
        write('Eingabe einer Nominalphrase (mit <Punkt><Return> beenden):'),nl,
        read_sentence(user,EingabeNP,[]),
        stringToAtoms(EingabeNP,WortfolgeSz),
        entferneSatzzeichen(WortfolgeSz,Wortfolge),
        lexAnalyse(Wortfolge),
        (ein(pruefeLexWord) -> erkenneWortarten ; true),
        (gescheitert(lexAnalyse)
        -> (nl, write(user,'Keine syntaktische Analyse durchgeführt.'))
        ; (baeume(Wortfolge,Trees)
          -> (((kasus(KasusNumerus), beuge_in_kasus(KasusNumerus,Trees,TreesTr))
              ;(numerus(KasusNumerus), beuge_in_numerus(KasusNumerus,Trees,TreesTr)))
             -> nl, write(user,'Analyse(n) der Eingabe:'),
                nl, showASCIItrees(Trees), nl,
                nl, write(user,'Transformation(en) der Eingabe:'),
                nl, showASCIItrees(TreesTr), nl
             ; (nl, write(user,'Keine Transformation der Eingabe gefunden.')))
          ; (nl, write(user,'Keine syntaktische Analyse der Eingabe gefunden.'))
          )
        ).

beuge_in_kasus(Kasus,[Baum|Baeume],[BaumTr|BaeumeTr]) :-
        (Baum = [np(Art,[Num,Kas])|Teile]
        -> (transformiere([np(Art,[Num,Kas])|Teile],[Num,Kasus],BaumTr)
           -> true ; BaumTr = [])
        ; BaumTr = Baum),
        beuge_in_kasus(Kasus,Baeume,BaeumeTr).
beuge_in_kasus(Kasus,[],[]).

beuge_in_numerus(Numerus,[Baum|Baeume],[BaumTr|BaeumeTr]) :-
        (Baum = [np(Art,[Num,Kas])|Teile]
        -> (transformiere([np(Art,[Num,Kas])|Teile],[Numerus,Kas],BaumTr)
           -> true ; BaumTr = [])
        ; BaumTr = Baum),
        beuge_in_numerus(Numerus,Baeume,BaeumeTr).
beuge_in_numerus(Kasus,[],[]).
```

Der Parser liefert für die Nominalphrase `der Mensch` folgende Analyse:

⟨*beispiel*⟩≡

```
87 ?- parse. der Mensch.

1. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
```

```
        der
     n1([mask], [-, -, sg, nom])
        n([mask], [sg, nom])
           Mensch
```

Nach Laden des Parsers `app.grm.ed.pl` und der Datei `Transformationen/paradigma.pl` kann man dann Beispiele einfacher Nominalphrasen nach dem Kasus beugen:

⟨*beispiel*⟩+≡
```
66 ?- beuge(dat).
Eingabe einer Nominalphrase (mit <Punkt><Return> beenden):

Analyse(n) der Eingabe:

1. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
         der
      n1([mask], [-, -, sg, nom])
         n([mask], [sg, nom])
            Mensch

Transformation(en) der Eingabe:

1. np([def, 3, mask], [sg, dat])
      det([def, sw], [mask, sg, dat])
         dem
      n1([mask], [-, -, sg, dat])
         n([mask], [sg, dat])
            Menschen
```

⟨*Hilfsfunktion*⟩≡
```
% alternative Version von beuge/1: beuge/2 - zum besseren Tracen
% beuge(+Wortfolge, +KasusNumerus).

beuge(Wortfolge, KasusNumerus) :-
    lexAnalyse(Wortfolge),
    (ein(pruefeLexWord)
        -> erkenneWortarten
        ; true),
    (gescheitert(lexAnalyse)
        -> (nl, write(user, 'Keine syntaktische Analyse durchgeführt.'))
        ; (baeume(Wortfolge, Trees)
            -> (((kasus(KasusNumerus), beuge_in_kasus(KasusNumerus, Trees, TreesTr))
            ; (numerus(KasusNumerus), beuge_in_numerus(KasusNumerus, Trees, TreesTr)))
                ->  nl, write(user, 'Analyse(n) der Eingabe:'),
                    nl, showASCIItrees(Trees), nl,
                    nl, write(user, 'Transformation(en) der Eingabe:'),
                    nl, showASCIItrees(TreesTr), nl
                ; (nl, write(user, 'Keine Transformation der Eingabe gefunden.')))
            ; (nl, write(user,'Keine syntaktische Analyse der Eingabe gefunden.'))
            )
    ).
```

8.2.2 A.Völtz: Transformation einer Adjektivphrase in eine andere Form

(i) `ap -> a`

⟨*Paradigma der Adjektivphrasen*⟩≡

```
transformiere(
    [
        ap([Komp, Adekl, GenA, Num1, Kas1]),
        [a([], [Komp, Adekl, GenA, Num1, Kas1]), A]
    ],
    [Komp2, Adekl2, GenA2, Num2, Kas2], Tr) :-
    transformiere([a([], [Komp, Adekl, GenA, Num1, Kas1]), A],
                  [Komp2, Adekl2, GenA2, Num2, Kas2], ATr),
    Tr = [ap([Komp2, Adekl2, GenA2, Num2, Kas2]), ATr].
```

(ii) `ap -> a, ap`

⟨*Paradigma der Adjektivphrasen*⟩+≡

```
transformiere(
    [
        ap([Komp, Adekl, GenA, Num1, Kas1]),
        [a([], [Komp, Adekl, GenA, Num1, Kas1])|A],
        [ap([KompAP, Adekl, GenAP, Num1, Kas1])|AP]
    ],
    [Komp2, Adekl2, GenA2, Num2, Kas2], Tr) :-
    transformiere([a([], [Komp, Adekl, GenA, Num1, Kas1])|A],
                  [Komp2, Adekl2, GenA2, Num2, Kas2], ATr),
    transformiere([ap([KompAP, Adekl, GenA, Num1, Kas1])|AP],
                  [KompAP, Adekl2, GenAP, Num2, Kas2], APTr),
    Tr = [ap([Komp2, Adekl2, GenA2, Num2, Kas2]), ATr, APTr].
```

(iii) `ap -> ordinalzahl`

⟨*Paradigma der Adjektivphrasen*⟩+≡

```
transformiere(
    [
        ap([pos, Adekl, unsp, sg, Kas1]),
        [ordinalzahl, O]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    Tr = [ap([Komp, Adekl, unsp, sg, Kas2]), [ordinalzahl, O]].
```

(iv) `ap -> kardinalzahl`

⟨*Paradigma der Adjektivphrasen*⟩+≡

```
transformiere(
    [
        ap([-, -, unsp, pl, Kas1]),
        [kardinalzahl, K]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    Tr = [ap([-, -, unsp, pl, Kas2]), [kardinalzahl, K]].
```

```
ap --> ordinalzahl, ap
ap --> kardinalzahl, ap
```

```
PROBLEM:
* Soll AP *ein* Listenelement sein, funktioniert "die 3 großen
  Männer", nicht aber "die 3 großen auffälligen Männer".
* Ist AP eine *Restliste*, funktioniert "die 3 großen auffälligen
  Männer", nicht aber "die 3 großen Männer".

LÖSUNG: Zwei Regeln, je eine für jeden der Fälle.
```

(v) `ap -> ordinalzahl, ap_einfach`

⟨*Paradigma der Adjektivphrasen*⟩+≡
```
transformiere(
    [
        ap([Komp, Adekl, Gen, Num1, Kas1]),
        [ordinalzahl, O],
        [ap([Komp, Adekl, Gen, Num1, Kas1]), AP]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    transformiere([ap([Komp, Adekl, Gen, Num1, Kas1]), AP],
                  [Komp, Adekl, Gen, Num2, Kas2], APTr),
    Tr = [ap([Komp, Adekl, Gen, Num2, Kas2]), [ordinalzahl, O], APTr].
```

(vi) `ap -> ordinalzahl, ap_erweitert`

⟨*Paradigma der Adjektivphrasen*⟩+≡
```
transformiere(
    [
        ap([Komp, Adekl, Gen, Num1, Kas1]),
        [ordinalzahl, O],
        [ap([Komp, Adekl, Gen, Num1, Kas1])|AP]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    transformiere([ap([Komp, Adekl, Gen, Num1, Kas1])|AP],
                  [Komp, Adekl, Gen, Num2, Kas2], APTr),
    Tr = [ap([Komp, Adekl, Gen, Num2, Kas2]), [ordinalzahl, O], APTr].
```

(vii) `ap -> kardinalzahl, ap_einfach`

⟨*Paradigma der Adjektivphrasen*⟩+≡
```
transformiere(
    [
        ap([Komp, Adekl, Gen, Num1, Kas1]),
        [kardinalzahl, K],
        [ap([Komp, Adekl, Gen, Num1, Kas1]), AP]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    transformiere([ap([Komp, Adekl, Gen, Num1, Kas1]), AP],
                  [Komp, Adekl, Gen, Num2, Kas2], APTr),
    Tr = [ap([Komp, Adekl, Gen, Num2, Kas2]), [kardinalzahl, K], APTr].
```

(viii) `ap -> kardinalzahl, ap_erweitert`

⟨*Paradigma der Adjektivphrasen*⟩+≡
```
transformiere(
    [
        ap([Komp, Adekl, Gen, Num1, Kas1]),
```

```
        [kardinalzahl, K],
        [ap([Komp, Adekl, Gen, Num1, Kas1])|AP]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    transformiere([ap([Komp, Adekl, Gen, Num1, Kas1])|AP],
                  [Komp, Adekl, Gen, Num2, Kas2], APTr),
    Tr = [ap([Komp, Adekl, Gen, Num2, Kas2]), [kardinalzahl, K], APTr].
```

(ix) `ap -> ergaenzung, ap`

⟨*Paradigma der Adjektivphrasen*⟩+≡

```
transformiere(
    [
        ap([Komp, Adekl, Gen, Num1, Kas1]),
        [ergaenzung([Komp])|E],
        [ap([Komp, Adekl, Gen, Num1, Kas1])|AP]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    transformiere([ap([Komp, Adekl, Gen, Num1, Kas1])|AP],
                  [Komp, Adekl, Gen, Num2, Kas2], APTr),
    Tr = [ap([Komp, Adekl, Gen, Num2, Kas2]), [ergaenzung([Komp])|E],
         APTr].
```

(x) `ap -> np, a`

⟨*Paradigma der Adjektivphrasen*⟩+≡

```
transformiere(
    [
        ap([Komp, Adekl, Gen, Num1, Kas1]),
        NPvorangestellt,
        [a([KasNP], [Komp, Adekl, Gen, Num1, Kas1]), A]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    NPvorangestellt = [np([Def, 3, GenNP], [NumNP, KasNP])|NP],
    transformiere([a([KasNP], [Komp, Adekl, Gen, Num1, Kas1]), A],
                  [Komp, Adekl, Gen, Num2, Kas2], ATr),
    Tr = [ap([Komp, Adekl, Gen, Num2, Kas2]), NPvorangestellt, ATr].
```

(xi) `ap -> ap, conj, ap`

⟨*Paradigma der Adjektivphrasen*⟩+≡

```
transformiere(
    [
        ap([Komp, Adekl, Gen, Num1, Kas1]),
        [ap([Komp, Adekl, Gen, Num1, Kas1]), AP1],
        [conj([], []), C],
        [ap([Komp, Adekl, Gen, Num1, Kas1]), AP2]
    ],
    [Komp, Adekl, Gen, Num2, Kas2], Tr) :-
    transformiere([ap([Komp, Adekl, Gen, Num1, Kas1]), AP1],
                  [Komp, Adekl, Gen, Num2, Kas2], AP1Tr),
    transformiere([ap([Komp, Adekl, Gen, Num1, Kas1]), AP2],
                  [Komp, Adekl, Gen, Num2, Kas2], AP2Tr),
    Tr = [ap([Komp, Adekl, Gen, Num2, Kas2]), AP1Tr, [conj([], []), C],
         AP2Tr].
```

8.2.3 Transformation einfacher Sätze

(i) Transformation von Sätzen der Form `np v`: Um mit `phrasenparadigma.` auch Aussagesätze transformieren zu lassen, muß für jede Satzregel eine Transformationsregel angegeben werden, die den entsprechenden Analysebaum in den Baum mit anderen Formmerkmalen transformiert. Ein Beispiel hierfür ist:

⟨*Paradigma einfacher Sätze*⟩≡
```
transformiere([s([Def],[Tempus,Modus,VSt]),
               [np([Def, Per, Genus], [Num, Kas]) | NP],
               [v([[Kas]], [Per, Num, Tempus, Modus]),VFin]],
              [Tempus2,Modus2,VSt], Tr) :-
        (VSt = letzt ; VSt = zweit),!,
        NPTr = [np([Def, Per, Genus], [Num, Kas]) | NP],
        transformiere([v([[Kas]], [Per, Num, Tempus, Modus]),VFin],
                      [Per,Num,Tempus2,Modus2],VTr),
        Tr = [s([Def],[Tempus2,Modus2,VSt]), NPTr, VTr].
```

Das geht nur für definite Subjekte, wo die Definitheit des Satzes mit der des Subjekts übereinstimmt. *Man muß erst in der Satzgrammatik die Berechnung der Definitheit so anpassen, daß sie mit den deklarierten Satzkategorien übereinstimmt, bevor man die Transformationen schreibt.* Denn die Transformationen benutzen `kategorie/1`, um die alternativen Formen des Paradigmas zu durchlaufen.

(ii) Transformation einfacher Prädikativsätze in attributierte Nominalphrasen:

(iii) Transformation attributierter Nominalphrasen in Prädikativsätze: (unfertig!)

⟨*Transformationen/paradigma.pl*⟩+≡
```
npToS :-
        write('Eingabe einer Nominalphrase mit AP-Attribut (mit <Punkt><Return> beenden):'),nl,
        read_sentence(user,EingabeNP,[]),
        stringToAtoms(EingabeNP,WortfolgeSz),
        entferneSatzzeichen(WortfolgeSz,Wortfolge),
        lexAnalyse(Wortfolge),
        (ein(pruefeLexWord) -> erkenneWortarten ; true),
        (gescheitert(lexAnalyse)
        -> (nl, write(user,'Keine syntaktische Analyse durchgeführt.'))
        ; (baeume(Wortfolge,Trees)
          -> (npsToSs(Trees,TreesTr)
             -> nl, write(user,'Analyse(n) der Eingabe:'),
                nl, showASCIItrees(Trees), nl,
                nl, write(user,'Transformation(en) der Eingabe:'),
                nl, showASCIItrees(TreesTr), nl
             ; (nl, write(user,'Keine Transformation der Eingabe gefunden.')))
          ; (nl, write(user,'Keine syntaktische Analyse der Eingabe gefunden.'))
          )
        ).

npsToSs([],[]).
npsToSs([Tree|Trees],[TreeTr|TreesTr]) :-
        npToS(Tree,TreeTr),
        npsToSs(Trees,TreesTr).

npToS([np([def, Pers, Gen], [Num, nom]),
```

```
        [det([def, Adekl], [Gen, Num, nom]) | D],
        [n1([Gen], [Komp, Adekl, Num, nom]) | Rest]],
       [s(zweit), NP, HVerb, ApTr]) :-
        extrahiereAP([np([def, Pers, Gen], [Num, nom]),
                      [det([def, Adekl], [Gen, Num, nom])|D],
                      [n1([Gen], [Komp, Adekl, Num, nom])|Rest]],NP,AP),
        word(W,v([hilfsverb, sein], [Pers, Num, perf, ind])), % vorlaeufig. Langsam!
        HVerb = [v([hilfsverb, sein], [Pers, Num, perf, ind]), W],
        apToPraed(AP,ApTr).

extrahiereAP([np([def, Pers, Gen], [Num, nom]),
              [det([def, Adekl], [Gen, Num, nom])|D],
              [n1([Gen], [Komp, Adekl, Num, nom])|Rest]],NP,AP) :-
        extrahiereAP([n1([Gen], [Komp, Adekl, Num, nom])|Rest],N1,AP),
        NP =  [np([def, Pers, Gen], [Num, nom]),
               [det([def, Adekl], [Gen, Num, nom])|D], N1].

extrahiereAP([n1([Gen], [Komp, Adekl, Num, nom]),
              [ap([Komp, Adekl, GenA, Num, nom]) | A],
              [n([Gen], [Num, nom]), N]],
             [n1([Gen], [Komp, Adekl, Num, nom]),
              [n([Gen], [Num, nom]), N]],
             [ap([Komp, Adekl, GenA, Num, nom]) | A]).
```

Für manche Adjektivphrasen wird die gesamte Adjektivphrase zum Prädikat:

(a) `ap -> a, {ap}.`

⟨*Transformationen/paradigma.pl*⟩+≡
```
  apToPraed([ap([Komp,Adekl,GenA,Num,nom]),
             [a([],[Komp,Adekl,GenA,Num,nom]),A]
            ],ApTr) :-
          flektiere([a([],[Komp,Adekl,GenA,Num,nom]),A],[Komp,praedik],ApTr).
```

(b) `ap -> ordinalzahl ; kardinalzahl.` Diese Fälle übersetzen wir nicht in Prädikative. Ebenso bei

(c) `ap -> (kardinalzahl ; ordinalzahl), ap.`

(d) `ap -> ergaenzung, ap.` Das sollte in bestimmten Fällen prädikativ ausdrückbar sein: `die scheinbar gesicherten Erkenntnisse` ↦ `die Erkenntnisse sind scheinbar gesichert`. Aber wohl nicht: `das zugleich schrecklichste Beispiel`.

(e) `ap - np, a([Kas])`: Das Nominalkomplement des Adjektivs muß als Komplement vor das prädikativ verwendete Adjektiv gestellt werden:

⟨*Transformationen/paradigma.pl*⟩+≡
```
  apToPraed([ap([Komp,Adekl,GenA,Num,nom]),
             [np(Art,[Num2,Kas2]) | NP],
             [a([Kas2],[Komp,Adekl,GenA,Num,nom]),A]
            ],Praed) :-
          flektiere([a([Kas2],[Komp,Adekl,GenA,Num,nom]),A],[Komp,praedik],ATr),
          Praed = [praedikat([Komp]),
                   [np(Art,[Num2,Kas2]) | NP],
                   ATr].
  % man muss noch die Komplementrahmen in die flektierten Adjektive
```

```
% einsetzen!

apToPraed([ap([Komp,Adekl,GenA,Num,nom]),
           [pp(Art,[Praep])|PP],
           [a([Praep],[Komp,Adekl,GenA,Num,nom]),A]
          ],Praed) :-
        flektiere([a([Praep],[Komp,Adekl,GenA,Num,nom]),A],[Komp,praedik],ATr),
        Praed = [praedikat([Komp]),
                 [pp(Art,[Praep])|PP],
                 ATr].
```

⟨*beispiel*⟩+≡

`der an den Hirnforschern ausgetragene Streit.`

Fehler: Da die Partizipien im CISLEX nicht als solche, sondern als Adjektive klassifiziert werden, kann man hier das jeweils nötige Hilfsverb nicht eindeutig bestimmen. Das Beispiel wird übersetzt in: `der Streit ist an den Hirnforschern ausgetragen.` Man sollte (bei adjektivisch verwendeten Partizipien) `wird/wurde` statt `ist` einsetzen.

Fehler: Man muß die Hilfsverben zuerst flektierbar machen! Die entsprechenden Prädikativsätze werden vom Parser noch nicht analysiert.

8.3 Transformation einfacher Sätze in relativierte Nominalphrasen

Transformationsregeln von Christroph Ringlstetter, Regel für relativierte Nominalphrase von Liza Hassel. (Stand 24.6.2002)

Beginnend mit der NP im Nominativ werden alle NPs aus dem Eingabesatz genommen und durch den übrigen Satz relativiert. Dabei wird der Ursprungskasus der jeweiligen NP zunächst beibehalten. Wird also `die Hamsterin` im Satz `der Hamster verdankt der Hamsterin das Lied` relativiert, so soll die Transformation lauten: `der Hamsterin, der der Hamster das Lied verdankt`.

Falls die relativierte Nominalphrase in einem anderen Kasus gebraucht wird, muß man sie mit der Phrasenflexion (s.o.) in den gewünschten Kasus beugen.

Sätze mit intransitivem Verb Sätze mit transitivem Verb Sätze mit dreiargumentigen Verben: [nom, dat, akk], [nom, gen, akk] Sätze mit Präpositionalgefüge

(i) Relativierung des Subjekts eines einfachen Satzes mit intransitivem Verb `v([ [nom] ],Form)`:

⟨*Transformationen/transformationen.rel.pl*⟩≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
               [np([Def, Per, Genus], [Num, Kas]) | NP],
               [v([[Kas]], [Per, Num, Tempus, Modus]),VFin]],
              [Tempus,Modus,VSt], Tr) :-
        (VSt = letzt ; VSt = zweit),
        !,
        NP = [DET , N1],
        transformiere_rs([v([[Kas]], [Per, Num, Tempus, Modus]),VFin],
                      [Per,Num,Tempus2,Modus2],Genus,VTr),
         Tr = [np([Def, Per, Genus], [Num, Kas]), DET, N1, VTr].

transformiere_rs([v([[Kas]], [Per, Num, Tempus, Modus]),VFin],
                 [Per,Num,Tempus2,Modus2],Genus,RSTr) :-
```

```
                 RSTr = [s([rel(Genus,Num)],[Tempus,Modus,letzt]), NPTr, VTr],
                 NPTr = [np([rel, Per, Genus], [Num, Kas]) , RPTR],
                 VTr = [v([[Kas]], [Per, Num, Tempus, Modus]),VFin],
                 rp(Genus,Num,Kas,RPTR).
```

Regeln für die Relativpronomen (sollten durch die bisherigen Relativpronomen ersetzt werden)

⟨*Transformationen/transformationen.rel.pl*⟩+≡

```
rp(mask,sg,nom,RPTR) :- RPTR = [rp([mask],[sg,nom]),der].
rp(mask,sg,gen,RPTR) :- RPTR = [rp([mask],[sg,gen]),dessen].
rp(mask,sg,dat,RPTR) :- RPTR = [rp([mask],[sg,dat]),dem].
rp(mask,sg,akk,RPTR) :- RPTR = [rp([mask],[sg,akk]),den].

rp(fem,sg,nom,RPTR)  :- RPTR = [rp([fem],[sg,nom]),die].
rp(fem,sg,gen,RPTR)  :- RPTR = [rp([fem],[sg,gen]),deren].
rp(fem,sg,dat,RPTR)  :- RPTR = [rp([fem],[sg,dat]),der].
rp(fem,sg,akk,RPTR)  :- RPTR = [rp([fem],[sg,akk]),die].

rp(neut,sg,nom,RPTR)  :- RPTR = [rp([neut],[sg,nom]),das].
rp(neut,sg,gen,RPTR)  :- RPTR = [rp([neut],[sg,gen]),dessen].
rp(neut,sg,dat,RPTR)  :- RPTR = [rp([neut],[sg,dat]),dem].
rp(neut,sg,akk,RPTR)  :- RPTR = [rp([neut],[sg,akk]),das].
```

⟨*Beispiele*⟩≡

```
Der Hamster kommt
    Der Hamster der kommt

Die Hamsterin kommt
    Die Hamsterin die kommt

Analyse(n) der Eingabe:
------------

1. s([def], [praes, ind, zweit])
      np([def, 3, mask], [sg, nom])
         det([def, sw], [mask, sg, nom])
            der
         n1([mask], [-, -, sg, nom])
            n([mask], [sg, nom])
               Hamster
      v([[nom]], [3, sg, praes, ind])
         kommt

Transformation der Eingabe:
--------------

1. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
         der
      n1([mask], [-, -, sg, nom])
         n([mask], [sg, nom])
            Hamster
      s([rel(mask, sg)], [praes, ind, letzt])
         np([rel, 3, mask], [sg, nom])
```

```
              rp([mask], [sg, nom])
                 der
          v([[nom]], [3, sg, praes, ind])
              kommt
```

(ii) Relativierung des Subjekts eines Satzes mit transitivem Verb `v([ [nom,Kas] ],Form)`:

⟨*Transformationen/transformation.rel.pl*⟩≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
               [np([Def, Per, Genus], [Num, Kas]) | NP],
               [v([[Kas,Kas2]], [Per, Num, Tempus, Modus]),VFin],
               [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2]
               ],
                [Tempus,Modus,VSt], Tr) :-
                (VSt = letzt ; VSt = zweit),%!,
                NP = [DET , N1],
                transformiere_rs([[v([[Kas,Kas2]], [Per, Num, Tempus, Modus]),VFin],
                                  [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2]
                                 ],
                                  [Per,Num,Tempus2,Modus2],Genus,VTr),
                Tr = [np([Def, Per, Genus], [Num, Kas]), DET, N1, VTr].

transformiere_rs([[v([[Kas,Kas2]], [Per, Num, Tempus, Modus]),VFin],
                  [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2]
                 ],
                  [Per,Num,Tempus2,Modus2],Genus,RSTr) :-
                 RSTr = [s([rel(Genus, Num)],[Tempus,Modus,letzt]), RPTr, NPTr, VTr],
                 RPTr = [np([rel, Per, Genus], [Num, Kas]) , RPTR],
                 VTr = [v([[Kas,Kas2]], [Per, Num, Tempus, Modus]),VFin],
                 NPTr = [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2],
                 rp(Genus,Num,Kas,RPTR).
```

(iii) Relativierung des Akkusativ-Objekts eines Satzes mit transitivem Verb `v([ [nom,akk] ],Form)`:

⟨*Transformationen/transformation.rel.pl*⟩+≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
               [np([Def, Per, Genus], [Num, Kas]) | NP],
               [v([[Kas,Kas2]], [Per, Num, Tempus, Modus]),VFin],
               [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2]
               ],
              [Tempus,Modus,VSt], Tr) :-
               (VSt = letzt ; VSt = zweit),!,
               NP2 = [DET , N1],
               transformiere_rs([[v([[Kas,Kas2]], [Per, Num, Tempus, Modus]),VFin],
                                 [np([Def, Per, Genus], [Num, Kas]) | NP]
                                ],
                                [Per,Num,Tempus,Modus],[Genus2,Num2],VTr),
               Tr = [np([Def2, Per2, Genus2], [Num2, Kas2]), DET, N1, VTr].


transformiere_rs([[v([[Kas,Kas2]], [Per, Num, Tempus, Modus]),VFin],
                  [np([Def, Per, Genus], [Num, Kas]) | NP]
                 ],
                  [Per,Num,Tempus,Modus],[Genus2,Num2],RSTr) :-
```

```
                   RSTr = [s([rel(Genus2, Num2)],[Tempus,Modus,letzt]), RPTr, NPTr, VTr],
                   RPTr = [np([rel, Per, Genus2], [Num2, Kas2]) , RPTR],
                   VTr = [v([[Kas,Kas2]], [Per, Num, Tempus, Modus]),VFin],
                   NPTr = [np([Def, Per, Genus], [Num, Kas]) | NP],
                   rp(Genus2,Num2,Kas2,RPTR).
```

⟨*Beispiele*⟩+≡

```
Der Hamster hört die Hamsterin
    Der Hamster der die Hamsterin hört
    Die Hamstrin die der Hamster hört

Der Hamster hört das Lied
    Der Hamster der das Lied hört
    Das Lied das der Hamster hört

Analyse(n) der Eingabe:
------------

1. s([def], [praes, ind, zweit])
      np([def, 3, mask], [sg, nom])
         det([def, sw], [mask, sg, nom])
            der
         n1([mask], [-, -, sg, nom])
            n([mask], [sg, nom])
               Hamster
      v([[nom, akk]], [3, sg, praes, ind])
         hört
      np([def, 3, neut], [sg, akk])
         det([def, sw], [neut, sg, akk])
            das
         n1([neut], [-, -, sg, akk])
            n([neut], [sg, akk])
               Lied

Transformation der Eingabe:
--------------

1. np([def, 3, neut], [sg, akk])
      det([def, sw], [neut, sg, akk])
         das
      n1([neut], [-, -, sg, akk])
         n([neut], [sg, akk])
            Lied
      s([rel(neut, sg)], [praes, ind, letzt])
         np([rel, 3, neut], [sg, akk])
            rp([neut], [sg, akk])
               das
         np([def, 3, mask], [sg, nom])
            det([def, sw], [mask, sg, nom])
               der
            n1([mask], [-, -, sg, nom])
               n([mask], [sg, nom])
                  Hamster
         v([[nom, akk]], [3, sg, praes, ind])
```

```
            hört

2. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
         der
      n1([mask], [-, -, sg, nom])
         n([mask], [sg, nom])
            Hamster
      s([rel(mask, sg)], [praes, ind, letzt])
         np([rel, 3, mask], [sg, nom])
            rp([mask], [sg, nom])
               der
         np([def, 3, neut], [sg, akk])
            det([def, sw], [neut, sg, akk])
               das
            n1([neut], [-, -, sg, akk])
               n([neut], [sg, akk])
                  Lied
         v([[nom, akk]], [3, sg, praes, ind])
            hört
```

⟨*Transformiere/transformation.bsp.wrd*⟩≡

```
stamm(v, verdanken, [nom, akk, dat]).
stamm(v, beschuldigen, [nom, akk, gen]).
```

⟨*Beispiel*⟩+≡

```
Der Hamster verdankt der Hamsterin das Lied

    der Hamster der der Hamsterin das Lied verdankt
    das Lied das der Hamster der Hamsterin verdankt
    der Hamsterin der der Hamster das Lied verdankt

Der Hamster beschuldigt die Hamsterin der Liebe

    der Hamster der die Hamsterin der Liebe beschuldigt
    die Hamsterin die der Hamster der Liebe beschuldigt
    der Liebe, deren der Hamster die Hamsterin beschuldigt.

Analyse(n) der Eingabe:
------------

1. s([def], [praes, ind, zweit])
      np([def, 3, mask], [sg, nom])
         det([def, sw], [mask, sg, nom])
            der
         n1([mask], [-, -, sg, nom])
            n([mask], [sg, nom])
               Hamster
      v([[nom, akk, dat]], [3, sg, praes, ind])
         verdankt
      np([def, 3, fem], [sg, dat])
         det([def, sw], [fem, sg, dat])
            der
         n1([fem], [-, -, sg, dat])
```

```
        n([fem], [sg, dat])
           Hamsterin
    np([def, 3, neut], [sg, akk])
       det([def, sw], [neut, sg, akk])
          das
       n1([neut], [-, -, sg, akk])
          n([neut], [sg, akk])
             Lied

Transformation der Eingabe:
--------------

1. np([def, 3, neut], [sg, akk])
      det([def, sw], [neut, sg, akk])
         das
      n1([neut], [-, -, sg, akk])
         n([neut], [sg, akk])
            Lied
      s([rel(neut, sg)], [praes, ind, letzt])
         np([rel, 3, neut], [sg, akk])
            rp([neut], [sg, akk])
               das
         np([def, 3, mask], [sg, nom])
            det([def, sw], [mask, sg, nom])
               der
            n1([mask], [-, -, sg, nom])
               n([mask], [sg, nom])
                  Hamster
         np([def, 3, fem], [sg, dat])
            det([def, sw], [fem, sg, dat])
               der
            n1([fem], [-, -, sg, dat])
               n([fem], [sg, dat])
                  Hamsterin
         v([[nom, akk, dat]], [3, sg, praes, ind])
            verdankt

2. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
         der
      n1([mask], [-, -, sg, nom])
         n([mask], [sg, nom])
            Hamster
      s([rel(mask, sg)], [praes, ind, letzt])
         np([rel, 3, mask], [sg, nom])
            rp([mask], [sg, nom])
               der
         np([def, 3, fem], [sg, dat])
            det([def, sw], [fem, sg, dat])
               der
            n1([fem], [-, -, sg, dat])
               n([fem], [sg, dat])
                  Hamsterin
```

```
          np([def, 3, neut], [sg, akk])
             det([def, sw], [neut, sg, akk])
                das
             n1([neut], [-, -, sg, akk])
                n([neut], [sg, akk])
                   Lied
          v([[nom, akk, dat]], [3, sg, praes, ind])
             verdankt

3. np([def, 3, fem], [sg, dat])
       det([def, sw], [fem, sg, dat])
          der
       n1([fem], [-, -, sg, dat])
          n([fem], [sg, dat])
             Hamsterin
       s([rel(fem, sg)], [praes, ind, letzt])
          np([rel, 3, fem], [sg, dat])
             rp([fem], [sg, dat])
                der
          np([def, 3, mask], [sg, nom])
             det([def, sw], [mask, sg, nom])
                der
             n1([mask], [-, -, sg, nom])
                n([mask], [sg, nom])
                   Hamster
          np([def, 3, neut], [sg, akk])
             det([def, sw], [neut, sg, akk])
                das
             n1([neut], [-, -, sg, akk])
                n([neut], [sg, akk])
                   Lied
          v([[nom, akk, dat]], [3, sg, praes, ind])
             verdankt
```

(iv) Regel fuer bitransitive Verben, Relativierung nom np

⟨*Transformation/transformation.rel.pl*⟩≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
                [np([Def, Per, Genus], [Num, Kas]) | NP],
                [v([[Kas,Kas3,Kas2]], [Per, Num, Tempus, Modus]),VFin],
                [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2],
                [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3]
              ],
               [Tempus,Modus,VSt], Tr) :-
               (VSt = letzt ; VSt = zweit), %!,
                NP = [DET , N1],
               transformiere_rs([[v([[Kas,Kas3,Kas2]], [Per, Num,Tempus, Modus]), VFin],
                                 [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2],
                                 [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3]
                                ],
                                 [Per,Num,Tempus2,Modus2],Genus,VTr),
               Tr = [np([Def, Per, Genus], [Num, Kas]), DET, N1, VTr].
```

(v) Extra Regel für bitransitive Verben, Relativierung nom np Stellungsvariante [nom, akk, gen],

⟨*Transformation/transformation.rel.pl*⟩+≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
                [np([Def, Per, Genus], [Num, Kas]) | NP],
                [v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3],
                [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2]
              ],
               [Tempus,Modus,VSt], Tr) :-
               (VSt = letzt ; VSt = zweit), %!,
               NP = [DET , N1],
               transformiere_rs([[v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                                  [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3],
                                  [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2]
                                 ],
                                  [Per,Num,Tempus2,Modus2],Genus,VTr),
               Tr = [np([Def, Per, Genus], [Num, Kas]), DET, N1, VTr].


transformiere_rs([[v([[NOM,AKK,DATGEN]], [Per, Num,
Tempus,Modus]),VFin],
                   [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2],
                   [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3]
                 ],
                  [Per,Num,Tempus2,Modus2],Genus,RSTr) :-
                  RSTr = [s([rel(Genus, Num)],[Tempus,Modus,letzt]), RPTr, NPTr1,NPTr2,  VTr],
                  RPTr = [np([rel, Per, Genus], [Num, Kas]) , RPTR],
                  VTr = [v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                  NPTr1 = [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2],
                  NPTr2 = [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3],
                  rp(Genus,Num,Kas,RPTR),
                  NOM = Kas,
                  ((AKK = Kas2,DATGEN = Kas3);(AKK = Kas3;DATGEN = Kas2)).
```

(vi) Regel fuer bitransitive Verben, Relativierung 1. objekt- np

⟨*Transformation/transformation.rel.pl*⟩+≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
                [np([Def, Per, Genus], [Num, Kas]) | NP],
                [v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2],
                [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3]
              ],
              [Tempus,Modus,VSt], Tr) :-
               (VSt = letzt ; VSt = zweit), %!,
               NP2 = [DET , N1],
               transformiere_rs([[v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                                  [np([Def, Per, Genus], [Num, Kas]) | NP],
                                  [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3]
                                 ],
                                 [Per,Num,Tempus2,Modus2],[Genus2,Num2,Kas2],VTr),
               Tr = [np([Def2, Per2, Genus2], [Num2, Kas2]), DET, N1, VTr].

transformiere_rs([[v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
```

```
                    [np([Def, Per, Genus], [Num, Kas]) | NP],
                    [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3]],
                    [Per,Num,Tempus,Modus],[GenusRP,NumRP,KasRP],RSTr)
                 :-
                  RSTr = [s([rel(GenusRP, NumRP)],[Tempus,Modus,letzt]),
                          RPTr, NPTr1,NPTr2,  VTr],
                  RPTr = [np([rel, Per, GenusRP], [NumRP, KasRP]) , RPTR],
                  VTr = [v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                  NPTr1 = [np([Def, Per, Genus], [Num, Kas]) | NP],
                  NPTr2 = [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3],
                  NOM = Kas,
                  ((AKK = KasRP,DATGEN = Kas3);(AKK = Kas3;DATGEN = KasRP)),
                  rp(GenusRP,NumRP,KasRP,RPTR).
```

(vii) Regel für bitransitive Verben, Relativierung 2. objekt- np akk

⟨*Treansformation/transformation.rel.pl*⟩≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
                 [np([Def, Per, Genus], [Num, Kas]) | NP],
                 [v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                 [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2],
                 [np([Def3, Per3, Genus3], [Num3, Kas3]) | NP3]
                ],
                 [Tempus,Modus,VSt], Tr) :-
                 (VSt = letzt ; VSt = zweit), %!,
                 NP3 = [DET , N1],
                 transformiere_rs([[v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                                  [np([Def, Per, Genus], [Num, Kas]) | NP],
                                  [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2]
                                 ],
                                 [Per,Num,Tempus,Modus],[Genus3,Num3,Kas3],VTr),
                 Tr = [np([Def3, Per3, Genus3], [Num3, Kas3]), DET, N1, VTr].

transformiere_rs([[v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                   [np([Def, Per, Genus], [Num, Kas]) | NP],
                   [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2]],
                   [Per,Num,Tempus,Modus],[GenusRP,NumRP,KasRP],RSTr)
                 :-
                  RSTr = [s([rel(GenusRP, NumRP)],[Tempus,Modus,letzt]),RPTr, NPTr1,NPTr2,  VTr],
                  RPTr = [np([rel, Per, GenusRP], [NumRP, KasRP]) , RPTR],
                  VTr = [v([[NOM,AKK,DATGEN]], [Per, Num, Tempus, Modus]),VFin],
                  NPTr1 = [np([Def, Per, Genus], [Num, Kas]) | NP],
                  NPTr2 = [np([Def2, Per2, Genus2], [Num2, Kas2]) | NP2],
                  NOM = Kas,
                  ((AKK = KasRP,DATGEN = Kas2);(AKK = Kas2;DATGEN = KasRP)),
                    rp(GenusRP,NumRP,KasRP,RPTR).
```

(viii) Relativierung von Präpositionalobjekten bei Verben `v([ [nom,Praep] ],Form)`:

⟨*Brispiel*⟩≡

```
Der Hamster handelt mit der Hamsterin

    der Hamster der mit der Hamsterin handelt.
    der Hamsterin mit der der Hamster handelt.
```

```
Analyse der Eingabe
----------

1. s([def], [praes, ind, zweit])
      np([def, 3, mask], [sg, nom])
         det([def, sw], [mask, sg, nom])
            der
         n1([mask], [-, -, sg, nom])
            n([mask], [sg, nom])
               Hamster
      v([[nom, praep(mit)]], [3, sg, praes, ind])
         handelt
      pp([def(3, sg)], [mit])
         pr([[dat]], [mit])
            mit
         np([def, 3, fem], [sg, dat])
            det([def, sw], [fem, sg, dat])
               der
            n1([fem], [-, -, sg, dat])
               n([fem], [sg, dat])
                  Hamsterin

Transformationen  der Eingabe:
---------------

1. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
         der
      n1([mask], [-, -, sg, nom])
         n([mask], [sg, nom])
            Hamster
      s([rel(mask, sg)], [praes, ind, letzt])
         np([rel, 3, mask], [sg, nom])
            rp([mask], [sg, nom])
               der
         pp([def(3, sg)], [mit])
            pr([[dat]], [mit])
               mit
            np([def, 3, fem], [sg, dat])
               det([def, sw], [fem, sg, dat])
                  der
               n1([fem], [-, -, sg, dat])
                  n([fem], [sg, dat])
                     Hamsterin
         v([[nom, praep(mit)]], [3, sg, praes, ind])
            handelt

2. np([def, 3, fem], [sg, dat])
      np([def, 3, fem], [sg, dat])
         det([def, sw], [fem, sg, dat])
            der
         n1([fem], [-, -, sg, dat])
            n([fem], [sg, dat])
```

```
            Hamsterin
      s([rel(fem, sg, praep(mit))], [praes, ind, letzt])
         np([rel, 3, fem], [sg, dat])
            pr([[dat]], [mit])
               mit
            rp([fem], [sg, dat])
               der
         np([def, 3, mask], [sg, nom])
            det([def, sw], [mask, sg, nom])
               der
            n1([mask], [-, -, sg, nom])
               n([mask], [sg, nom])
                  Hamster
          v([[nom, praep(mit)]], [3, sg, praes, ind])
            handelt
```

(ix) Relativierung der np nom für präpositional ergänzte Verben

⟨*Transformation/transformation.rel.pl*⟩+≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
               [np([Def, Per, Genus], [Num, Kas]) | NP],
               [v([[NOM,praep(Praep)]], [Per, Num, Tempus, Modus]),VFin],
               [pp([Def2],[Praep])|PP]
              ],
               [Tempus,Modus,VSt], Tr) :-
                (VSt = letzt ; VSt = zweit), %!,
                NP = [DET , N1],
                transformiere_rs([[v([[NOM,praep(Praep)]], [Per, Num,Tempus, Modus]),VFin],
                                  [pp([Def2], [Praep])|PP]
                                 ],
                                 [Per,Num,Tempus,Modus],[Genus,Num,Kas],VTr),
                Tr = [np([Def, Per, Genus], [Num, Kas]), DET, N1, VTr].

transformiere_rs([[v([[NOM,praep(Praep)]], [Per, Num, Tempus, Modus]),VFin],
                  [pp([Def2], [Praep])|PP]
                 ],
                  [Per,Num,Tempus,Modus],[GenusRP,NumRP,KasRP],RSTr)
                 :-
                 RSTr = [s([rel(GenusRP, NumRP)],[Tempus,Modus,letzt]),RPTr, PPTr,VTr],
                 RPTr = [np([rel, Per, GenusRP], [NumRP, KasRP]), RPTR],
                 VTr =  [v([[NOM,praep(Praep)]], [Per, Num, Tempus, Modus]),VFin],
                 PPTr = [pp([Def2], [Praep])|PP],
                 NOM = KasRP,
                 rp(GenusRP,NumRP,KasRP,RPTR).
```

(x) Relativierung der np der pp Fall für präpositional ergänzte Verben

⟨*Transformation/transfromation.rel.pl*⟩≡

```
transformiere([s([Def],[Tempus,Modus,VSt]),
               [np([Def, Per, Genus], [Num, Kas]) | NP],
               [v([[NOM,praep(Praep)]], [Per, Num, Tempus, Modus]),VFin],
               [pp([Def2],[Praep])|PP]
              ],
               [Tempus,Modus,VSt], Tr) :-
               (VSt = letzt ; VSt = zweit), %!,
```

```
                    PP = [PR,NPPP],
                    NPPP = [Kopf|Rest],
                    Kopf = np([def,3,GenusRP],[NumRP,KasusRP]),
                    transformiere_rs([[v([[NOM,praep(Praep)]], [Per, Num, Tempus, Modus]),VFin],
                                      [pr([[KASPP]], [Praep]),PR],
                                      [np([Def, Per, Genus], [Num,Kas]) | NP]
                                     ],
                                     [Per,Num,Tempus,Modus],[GenusRP,NumRP,KasusRP],VTr),
                    Tr = [np([Def, Per, GenusRP], [NumRP, KasusRP]), NPPP, VTr].

transformiere_rs([[v([[NOM,praep(Praep)]], [Per, Num, Tempus, Modus]),VFin],
                  [pr([[KASPP]], [Praep]),PR],
                  [np([Def, Per, Genus], [Num,Kas]) | NP]
                 ],
                  [Per,Num,Tempus,Modus],[GenusRP,NumRP,KasRP],RSTr)
                  :-
                  RSTr = [s([rel(GenusRP, NumRP, praep(Praep))],[Tempus,Modus,letzt]),
                          RPTr, NPTr,VTr],
                  RPTr = [np([rel, Per, GenusRP], [NumRP, KasRP]) , PR,RPTR],
                  NPTr = [np([Def, Per, Genus], [Num,Kas]) | NP],
                  VTr =  [v([[NOM,praep(Praep)]], [Per, Num, Tempus, Modus]),VFin],
                  rp(GenusRP,NumRP,KasRP,RPTR).
```

(xi) Relativierung eines Genitivattributs:

⟨*Beispiele*⟩+≡

```
die Hamsterin des Hamsters kommt.
       die Hamsterin des Hamsters die kommt.
       des Hamsters, dessen Hamsterin kommt.

Analyse(n) der Eingabe:
------------

1. s([def], [praes, ind, zweit])
      np([def, 3, fem], [sg, nom])
         det([def, sw], [fem, sg, nom])
            die
         n1([fem], [-, -, sg, nom])
            n([fem], [sg, nom])
               Hamsterin
            np([def, 3, mask], [sg, gen])
               det([def, sw], [mask, sg, gen])
                  des
               n1([mask], [-, -, sg, gen])
                  n([mask], [sg, gen])
                     Hamsters
      v([[nom]], [3, sg, praes, ind])
         kommt

Transformation der Eingabe:
--------------

1. np([def, 3, fem], [sg, nom])
      det([def, sw], [fem, sg, nom])
```

```
   die
n1([fem], [-, -, sg, nom])
   n([fem], [sg, nom])
      Hamsterin
   np([def, 3, mask], [sg, gen])
      det([def, sw], [mask, sg, gen])
         des
      n1([mask], [-, -, sg, gen])
         n([mask], [sg, gen])
            Hamsters
s([rel(fem, sg)], [praes, ind, letzt])
   np([rel, 3, fem], [sg, nom])
      rp([fem], [sg, nom])
         die
   v([[nom]], [3, sg, praes, ind])
      kommt
```

Für den ersten Fall wird die normale Transformationsregel für Relativierung der Nominativ-NP verwendet. Der Baum für das Genitivattribut ist in der | `NP`-Restliste. verwendet

???Noch machen??? Extraregel für Relativierung der np Genitivattribut. Relativpronomen `dessen`

9 Anhang 1: Hilfsprogramme

9.1 Transitive Hülle einer zweistelligen Relation

Wir brauchen die transitive Hülle gewisser Relationen zwischen grammatischen Kategorien, insbesondere die der Relationen

$$\{(A, B) \mid A \to B\} \quad \text{und} \quad \{(A, B) \mid \exists \alpha, \beta\, (A \to \alpha B \beta \wedge \mathit{deletable}(\alpha))\}$$

Da diese von möglichen Eingabesätzen unabhängig sind, kann man sie vor der Satzanalyse berechnen und während der Analyse auf die vorberechneten Relationen zurückgreifen.

Genauer gesagt, brauchen wir nur die transitive, nicht die transitive reflexive Hülle. Aus der transitiven Hülle `tc` sondern wir die Elemente der Form (X, X) aus, um die *irreflexive transitive Hülle* wie folgt zu definieren:

⟨*Parser/transclos.pl*⟩≡
```
:- ensure_loaded('Parser/subsumption.pl').

itc(Base,Ancestor,Descendant) :-
        tc(Base,Ancestor,Descendant),
        not(Ancestor == Descendant).
```

Wir nehmen an, daß die Ausgangsrelation als Prädikat `Base/2` gegeben ist. Die transitive Hülle von `Base/2` ist dann `tc(Base,X,Y)`. Zur Berechnung der transitiven Hülle betrachten wir verschiedene Alternativen: einerseits Varianten des Algorithmus' von Warshall und andererseits eine induktive Berechnung in Stadien.

Die folgende Fallunterscheidung dient zum Testen, bis eine brauchbare Version fertig ist. Je nach Auswahl `auswahl(tc(Warshall))` wird die transitive Hülle nach einer anderen Version berechnet. (Damit man sie parallel benutzten kann, haben die Moduln zum Teil eine leere Export-Liste.)

⟨*Parser/transclos.pl*⟩+≡
```
:- use_module('Parser/warshall.1.pl').
:- use_module('Parser/warshall.2.pl').
:- use_module('Parser/warshall.3.pl').
:- use_module('Parser/warshall.4.pl').
:- use_module('Parser/inductive.pl').

tc(Base,Ancestor,Descendant) :-
        ((tc(warshall-1) ; tc(warshall-2) ; tc(warshall-3) ; tc(warshall-4))
        -> tc_warshall(Base,Ancestor,Descendant)
        ; true),
        (tc(inductive) -> tc_inductive(Base,Ancestor,Descendant) ; true).

⟨warshall⟩
```

Ein effizienter Algorithmus zur Berechnung der transitiven Hülle einer zweistelligen Relation stammt von C.Warshall. Für eine Grundrelation von N Tupeln berechnet dieses Verfahren die transitive Hülle in $O(N^3)$-Schritten (falls die Knoten atomar sind).

⟨*warshall*⟩≡
```
tc_warshall(Base,Ancestor,Descendant) :-
        base_graph(Base,Graph),
        warshall(Graph,Closure),
        member(Ancestor-Descendants,Closure),
        member(Descendant,Descendants).
```

In tc/3 wird zuerst das Prädikat Base/2 in einen Eingabegraphen umgewandelt, der durch eine Liste von Paaren (auch mit freien Variablen) der Form Parent-Children dargestellt wird, wobei Children eine Liste der Base/2-Nachkommen des Elements Parent ist:

⟨*warshall*⟩+≡
```
base_graph(Base,Graph) :-
        Edge =.. [Base,Parent,Child],
        setof(Parent-Children,
              setof(Child,Edge,Children),
              Graph).
```

Die Ausgabe von warshall(Graph+,Closure-) ist die transitive Hülle dieses Graphen, in der gleichen Darstellung.

⟨*warshall*⟩+≡
```
warshall(Graph, Closure) :-
        setof(V,Vs^member(V-Vs,Graph),Nodes),
        warshall(Nodes,Graph,Closure).
```

Der Kern des Algorithmus ist warshall/3, wovon wir verschiedene Versionen betrachten (bis eine korrekte und schnelle fertig ist):

⟨*warshall*⟩+≡
```
warshall(Nodes,Graph,Closure) :-
        (tc(warshall-1) -> 'warshall-1':warshall(Nodes,Graph,Closure) ; true),
        (tc(warshall-2) -> 'warshall-2':warshall(Nodes,Graph,Closure) ; true),
        (tc(warshall-3) -> 'warshall-3':warshall(Nodes,Graph,Closure) ; true),
        (tc(warshall-4) -> 'warshall-4':warshall(Nodes,Graph,Closure) ; true).
```

9.1.1 Der Algorithmus von C.Warshall für Relationen zwischen Grundtermen

Wir orientierun uns an der Implementierung in Prolog aus dem Buch von Richard O'Keefe, *The Craft of Prolog*, S.171 ff. (Von den dort angegebenen vier Versionen betrachten wir nur die dritte Version.)

⟨*Parser/warshall.1.pl*⟩≡
```
:- module('warshall-1',[]).
```

Das erste Argument von warshall/3 ist die Liste der noch zu behandelnden Knoten des Eingangsgraphen, das zweite der um solche Kanten erweiterte Graph, die einen Weg durch einen schon behandelten Knoten abkürzen.

Falls keine Knoten mehr zu behandeln sind, ist der erweiterte Graph die transitive Hülle des Eingabegraphen:

⟨*Parser/warshall.1.pl*⟩+≡
```
warshall([],Closure,Closure).
```

Sonst sucht man vom nächsten zu behandelnden Knoten, V, die Nachfolger Y im erweiterten(!) Graphen E, erweitert E um Abkürzungen von Wegen durch V, und behandelt dann die restlichen Knoten:

⟨*Parser/warshall.1.pl*⟩+≡
```
warshall([V|Nodes],E,Closure) :-
        memberchk(V-Y,E),
        extend(E,V-Y,Extended),
        warshall(Nodes,Extended,Closure).
```

Ist V-Y ein Knoten V mit seinen Nachfolgern Y, so sind Wege von X durch V zu Knoten in Y durch direkte Wege von X zu diesen Knoten in Y abzukürzen:

⟨*Parser/warshall.1.pl*⟩+≡
```
extend([],_,[]).
extend([X-Neibs|G],V-Y,[X-NewNeibs|NewG]) :-
        memberchk(V,Neibs),
        !,
        union(Neibs,Y,NewNeibs),
        extend(G,V-Y,NewG).
extend([X-Neibs|G],V-Y,[X-Neibs|NewG]) :-
        extend(G,V-Y,NewG).
```

Dieses Programm zur Berechnung der transitiven Hülle kann *nicht für Relationen zwischen Termen mit freien Variablen* benutzt werden.

Beispiel 9.1 Angenommen, wir wollen die transitive Hülle der folgenden `first`-Relation berechnen:

⟨*tc: Beispiel 1*⟩≡
```
:- dynamic first/2.
first(a,b(X)).
first(b(1),c(1)).
first(b(Y),d(Y)).
```

Da freie Variable universell gelesen werden, sollten `first(a,d(Y))` und `first(a,c(1))` hinzukommen. Der erste Schritt der Berechnung ist noch in Ordnung:

⟨*tc: Beispiel 1*⟩+≡
```
287 ?- base_graph(first,Graph).
Graph = [a-[b(X)], b(Y)-[d(Y)], b(1)-[c(1)]]
```

Aber durch den anschließenden Aufruf von `warshall/2` wird der Ausgangsgraph verändert:

⟨*tc: Beispiel 1*⟩+≡
```
288 ?- base_graph(first,Graph), warshall(Graph,Closure).

Graph = [a-[b(1)], b(1)-[d(1)], b(1)-[c(1)]]
Closure = [a-[b(1), d(1)], b(1)-[d(1)], b(1)-[c(1)]]
```

Der Fehler kommt daher, daß die `memberchk/2`- und `union/3`-Aufrufe in `warshall/2` die freien Variablen im Ausgangsgraphen belegen.

9.1.2 Der Algorithmus von C. Warshall für Relationen zwischen beliebigen Termen

Wir versuchen daher, den Algorithmus von Warshall so anzupassen, daß freie Variable in den Tupeln wie ∀-quantifizierte Variable behandelt werden. Ein Tupel steht damit für die Menge seiner Instanzen, d.h. all der Tupel von Termen, die aus ihm durch Einsetzen von Termen in seine freien Variablen hervorgehen. Die transitive Hülle kann dann –in gewissen, bei den Grammatikvorberechnungen ausreichenden Fällen– durch eine endliche Liste von „allgemeinsten" Tupeln dargestellt werden, von denen keines eine Instanz eines anderen Tupels der Liste ist .

Instanzen und Belegungen werden in Abschnitt 9.1.4 über `Parser/subsumption.pl` behandelt.

Versuch 2

⟨*Parser/warshall.2.pl*⟩≡
```
:- module('warshall-2',[relation_product/3,instance_product/3,match_filter/3]).
```

Das erste Argument von `warshall/3` ist die Liste der noch zu behandelnden `Parent-Children`-Paare des Eingangsgraphen, von der man nur die `Parent`-Teile braucht, um alle Knoten des Graphen zu durchlaufen. Das zweite Argument ist eine Erweiterung des Ausgangsgraphen um die Kanten `(A,C)`, die durch Verkettung zweier Kanten `(A,B)` und `(B,C)` mit einem schon behandelten Knoten `B` des Ausgangsgraphen entstanden sind.

Falls keine Knoten mehr zu behandeln sind, ist der erweiterte Graph die transitive Hülle des Eingabegraphen:

⟨*Parser/warshall.2.pl*⟩+≡
```
warshall([],Closure,Closure).
```

Sonst sei etwa `V` der nächste zu behandelnde Knoten. Dann sucht man dessen Nachfolger, `Vs`, im erweiterten Graphen `E`, erweitert `E` um Abkürzungen von Wegen durch `V`, und behandelt dann die restlichen Knoten:

⟨*Parser/warshall.2.pl*⟩+≡
```
warshall([V|Nodes],E,Closure) :-
        descendants(V,E,Vs),
        extend(E,V-Vs,Extended),
        warshall(Nodes,Extended,Closure).
```

⟨*descendants*⟩

Um die Nachkommen von `V` in `E` zu finden, muß jeder Knoten `X` aus `E` berücksichtigt werden, von dem `V` eine Instanz ist; die Instanzen der Nachkommen der `X` werden durch `match_union/3` als Nachkommen von `V` gesammelt:

⟨*descendants*⟩≡
```
descendants(V,[X-Xs|Es],Vs) :-
        descendants(V,[X-Xs|Es],[],Vs).
descendants(V,[],[]).

descendants(V,[X-Xs|Es],Acc,Vs) :-
        !,(match(V,X,[],Env) -> substitute(Xs,Env,VXs) ; VXs = []),
        match_union(VXs,Acc,AccXs),
        descendants(V,Es,AccXs,Vs).
descendants(V,[],Acc,Acc) :- !.
```

Beim Sammeln der Nachfolger werden mit `match_union/3` aus `Parser/subsumption.pl` die Instanzen von Nachkommen ausgefiltert, da die ∀-quantifizierten freien Variablen ja die Spezialfälle enthalten sollen.

Sind `Vs` die Nachkommen von `V` im erweiterten Graphen, so müssen alle Wege nach `V` zu Wegen nach den `Vs` fortgesetzt werden. Dabei kommt man zu `V` auch durch Einsetzung in einen Nachfolger `Xi`:

⟨*Parser/warshall.2.pl*⟩+≡
```
extend([],_,[]).
extend([X-Xs|G],V-Vs,[X-NewXs|NewG]) :-
        link_check(V,Xs),
        !,
        relation_product(X-Xs,V-Vs,NewXs),
        instance_product(X-Xs,V-Vs,SubGraph),
        extend(G,V-Vs,NewGraph),
        append(SubGraph,NewGraph,NewGX),
        match_filter(X-NewXs,NewGX,NewG).
```

```
        % match_filter(X-NewXs,SubGraph,NewSubGraph),
        % match_filter(X-NewXs,NewGraph,NewGraphX),
        % append(NewSubGraph,NewGraphX,NewG).
extend([X-Xs|G],V-Vs,[X-Xs|NewG]) :-
        extend(G,V-Vs,GV),
        match_filter(X-Xs,GV,NewG). % reicht nicht, vgl Testbett,b1.
```

Wir brauchen einerseits direkte Produkte `(A,B) = (X,Xi)(V,Vj)`, wo `X` nicht instantiiert wird. Hierzu werden nach der Umbenennung der Variablen und der Gleichsetzung `Xi = V` in `link/3` die Variablen wieder zu den Variablen von `X` umbenannt:

⟨*Parser/warshall.2.pl*⟩+≡

```
relation_product(X-Xs,V-Vs,NewXs):-
        (setof(BE,Xi^Vj^A^B^Env^(member(Xi,Xs),
                                 member(Vj,Vs),
                                 link((X,Xi),(V,Vj),(A,B)),
                                 match(X,A,[],Env),
                                 substitute(B,Env,BE)),
               XVs) -> true ; XVs = []),
        match_union(XVs,Xs,NewXs).
```

Andererseits brauchen wir Produkte `(A,B) = (X,Xi)(V,Vj)`, die bei der Gleichsetzung `Xi = V` zu echten Instanzen von `X` führen:

⟨*Parser/warshall.2.pl*⟩+≡

```
instance_product(X-Xs,V-Vs,Graph) :-
        (setof(A-As,instances(X-Xs,V-Vs,A-As),GraphInst)
        -> match_filter(X-Xs,GraphInst,Graph) ; Graph = []).

instances(X-Xs,V-Vs,A-As) :-
        setof(B,Xi^Vj^(member(Xi,Xs),
                       member(Vj,Vs),
                       link((X,Xi),(V,Vj),(A,B)),
                       not(matches(X,A))),
                    Bs),
        match_compress(Bs,[],As).

match_compress([],Cs,Cs) :- !.
match_compress([B|Bs],Cs,Es) :-
        match_union([B],Cs,Ds),
        match_compress(Bs,Ds,Es).
```

Falls der Knoten `A` eine Instanz von `X` ist (unter der Belegung `Env`), so werden von seinen Nachfolgern `As` alle diejenigen entfernt, die Instanzen der `Env`-Instanzen der Nachfolger `Xs` von `X` sind:

⟨*Parser/warshall.2.pl*⟩+≡

```
match_filter(X-Xs,Graph,NewGraph) :-
        match_filter(X-Xs,Graph,[],NewGraph).

match_filter(X-Xs,[A-As|Graph],Acc,NewGraph) :-
        match(A,X,[],Env),
        !,
        substitute(Xs,Env,XsEnv),
        drop_instances(As,XsEnv,LessAs),
        (LessAs = []
```

```
        -> match_filter(X-Xs,Graph,Acc,NewGraph)
        ; match_filter(X-Xs,Graph,[A-LessAs|Acc],NewGraph)).
match_filter(X-Xs,[A-As|Graph],Acc,NewGraph) :-
        match_filter(X-Xs,Graph,[A-As|Acc],NewGraph).
match_filter(X-Xs,[],Acc,Acc).
```

⟨*link*⟩

Bem. Es wird in `match_filter/3` nicht geprüft, ob `X-Xs` auf Grund von allgemeineren Kanten im Restgraphen ausgedünnt werden könnte.

Beispiel 9.2 Wenn wir die transitive Hülle der folgenden Relation

⟨*tc: Beispiel 2*⟩≡

```
:- dynamic first/2.
first(a(X),b(X)).
first(b(1),c(1)).
first(b(Y),d(Y)).
first(c(Z),b(Z)).
```

berechnen, so erhalten wir (nach Einsetzen lesbarer Variablen):

⟨*tc: Beispiel 2*⟩+≡

```
51 ?- base_graph(first,Graph), warshall(Graph,Closure).

Graph = [a(X)-[b(X)], b(Y)-[d(Y)], b(1)-[c(1)], c(Z)-[b(Z)]]
Closure =
        [a(X)-[d(X), b(X)],
        a(1)-[c(1)],
        b(Y)-[d(Y)],
        b(1)-[b(1), c(1)],
        c(Z)-[d(Z), b(Z)],
        c(1)-[c(1)]]
```

Es werden nur die Spezialfälle aufgeführt, die nicht Instanzen allgemeinerer Fälle sind. (Dumm: durch eine Programmänderung ist das jetzt schlechter - aber ich habe die ursprüngliche Version nicht mehr. Nur Version 4 unten macht es korrekt.)

Problem: Die Implementierung von `warshall/3` ist nicht effizient genug: schon bei halbwegs brauchbaren Merkmalgrammatiken reicht der Rekursionsstack nicht aus! Man sollte die beiden Fälle `relation_product` und `instance_product` gemeinsam berechnen, sonst wird

```
member(Xi,Xs),member(Vj,Vs),link((X,Xi),(V,Vj),(A,B))
```

mit den teuren `setof`'s dauernd doppelt berechnet.

Versuch 3

Leider ist Version 2 unvollständig: es werden i.a. zu wenig Tupel berechnet. (Dann wird die First-Relation zu klein, folglich auch der statische Predictor, und bei der Analyse werden mögliche Lesarten der Eingabe nicht erkannt.) Die bei der Verkettung von Kanten durch Instanziierung neu entstehenden

Knoten(!) geben wir beim Rekursionsaufruf dazu, damit auch Wege durch diese Knoten abgekürzt werden:

⟨*Parser/warshall.3.pl*⟩≡
```
:- module('warshall-3',[]).

warshall([],Closure,Closure).
warshall([V|Nodes],E,Closure) :-
        descendants(V,E,Vs),
        extend(E,V-Vs,Extended,Untreated),
        collect_nodes(Untreated,NewNodes),
        append(NewNodes,Nodes,AllNodes),
        warshall(AllNodes,Extended,Closure).

⟨descendants⟩
collect_nodes(Graph,Nodes) :-
        collect_nodes(Graph,[],Nodes).
collect_nodes([A-As|G],Acc,Ns) :-
        collect_nodes(G,[A|Acc],Ns).
collect_nodes([],Acc,Acc).
```

Bei der Erweiterung des Graphes müssen wir uns die neu erzeugten Knoten als unbehandelte merken, da sie mit alten Knoten kombinierbar sein könnten, was später getestet wird.

⟨*Parser/warshall.3.pl*⟩+≡
```
extend([],_,[],[]).
extend([X-Xs|G],V-Vs,[X-NewXs|NewG],Untreated) :-
        link_check(V,Xs),
        !,
        relation_product(X-Xs,V-Vs,NewXs),
        instance_product(X-Xs,V-Vs,SubGraph),
        extend(G,V-Vs,NewGraph,UntreatedG),
        append(SubGraph,NewGraph,NewG),
        append(SubGraph,UntreatedG,Untreated).
extend([X-Xs|G],V-Vs,[X-Xs|NewG],Untreated) :-
        extend(G,V-Vs,NewG,Untreated).
⟨link⟩
```

Beispiel 9.3 Die folgende Relation ergibt:

⟨*tc: Beispiel 3*⟩≡
```
:- dynamic first/2.
first(a(*),d(X)).
first(b(X),a(X)).
first(c(Y),b(Y)).

26 ?- base_graph(first,Graph), warshall(Graph,Closure).

Graph = [a(*)-[d(X)], b(Y)-[a(Y)], c(Z)-[b(Z)]]
Closure = [a(*)-[d(X)], b(Y)-[a(Y)], b(*)-[d(U)],
           c(Z)-[a(Z), b(Z)], c(*)-[d(V), a(*)]]
```

Das ist noch zu groß: `(c(*),a(*))` ist schon in `(c(Z),a(Z))` enhalten!

Durch eine weitere Korrektur wurden die überflüssigen Elemente noch vermieden. Damit war der Fehler von Version 2 behoben, da die unbehandelten Knoten in der `Untreated` Liste gesammelt und in `warshall/3` durch den Rekursionsaufruf betrachtet wurden.

Bei kleinen Relationen wurde die Hülle richtig berechnet, aber bei großen war schnell die Stack-Grenze erreicht. (Beachte, daß weniger weggefiltert wird, die `match_filter`-Aufrufe fehlen, dafür hat man mehr `append`-Aufrufe.)

Versuch 4

Der nächste Versuch ist, durch Verwendung von Endrekursion die Version 3 effizienter zu machen.

⟨*Parser/warshall.4.pl*⟩≡
```
:- module('warshall-4',[]).
```

Das erste Argument von `warshall/3` ist die Liste der noch zu behandelnden Knoten des Graphen, von denen Kanten ausgehen. Das zweite Argument ist eine Erweiterung des Ausgangsgraphen um die Kanten `(A,C)`, die durch Verkettung zweier Kanten `(A,B)` und `(B,C)` mit einem schon behandelten Knoten `B` entstanden sind.

Falls keine Knoten mehr zu behandeln sind, ist der erweiterte Graph die transitive Hülle des Eingabegraphen:

⟨*Parser/warshall.4.pl*⟩+≡
```
warshall([],Closure,Closure).
```

Sonst sei etwa `V` der nächste zu behandelnde Knoten. Dann sucht man dessen Nachfolger, `Vs`, im erweiterten Graphen `E`, indem man zu allen Knoten `X` in `E`, von denen `V` eine Instanz ist, die entsprechenden Instanzen der Nachfolger `Xs` von `X` sammelt. Nun muß `E` erweitert werden, indem zu allen Knoten, unter deren Nachkommen `V` „vorkommt", die Nachkommen um die `Vs` erweitert werden. Da aber die Kanten `V-Vs` und die Kanten `Y-Ys` von `E` freie Variablen haben, muß man nicht nur `V` unter den `Ys` suchen und ggf. den Graphen um die Kanten `Y-Vs` erweitern, sondern man muß auch gemeinsame Instanzen `V'` und `Yi'` suchen und den Graphen ggf. um neue Kanten `Y'-Vs'` erweitern. Obendrein sollte der erweiterte Graph keine Kanten enthalten, die Instanzen anderer Kanten sind.

Nachdem `E` um Abkürzungen von Wegen durch `V` erweitert ist, behandelt man die restlichen Knoten; dazu müssen auch die Knoten `Y'` und `Vs'` gerechnet werden, die bei der Erweiterung neu „entstanden", da die neuen Kanten von oder zu diesen Knoten noch in die transitive Hülle eingehen.[30]

⟨*Parser/warshall.4.pl*⟩+≡
```
warshall([V|Nodes],E,Closure) :-
        descendants(V,E,Vs),
        extend(E,V-Vs,Extended),
        setof(N,Cs^member(N-Cs,E),NodesE),
        setof(N,Cs^member(N-Cs,Extended),NodesExtended),
        (setof(N,(member(N,NodesExtended),not(equiv_member(N,NodesE))),NewNodes)
        -> append(Nodes,NewNodes,ExtendedNodes)
        ; ExtendedNodes = Nodes),
        warshall(ExtendedNodes,Extended,Closure).

equiv_member(K,[N|Ns]) :-
        ((matches(K,N),matches(N,K)) -> true ; equiv_member(K,Ns)).
```

[30] Wenn wir zu den neu entstandenden Knoten nur die rechnen, die nicht zu `V` oder einem sowieso noch zu behandelnden Knoten äquivalent sind, kann das Verfahren divergieren. Muß man nicht nur die Ausgangs-, sondern auch die Zielknoten neuer Kanten als neue Knoten betrachten?

Die Nachkommen von `V` im Graphen `E` bestimmt man genau wie bei Version 2:

⟨*Parser/warshall.4.pl*⟩+≡
  ⟨*descendants*⟩

Nun muß `E` erweitert werden, indem zu allen Knoten, unter deren Nachkommen `V` „vorkommt", die Nachkommen um die `Vs` erweitert werden. Da aber die Kanten `V-Vs` und die Kanten `Y-Ys` von `E` freie Variablen haben, muß man nicht nur `V` unter den `Ys` suchen und ggf. den Graphen um die Kanten `Y-Vs` erweitern, sondern man muß auch gemeinsame Instanzen `V'` und `Yi'` suchen und den Graphen ggf. um neue Kanten `Y'-Vs'` erweitern. Obendrein sollte der erweiterte Graph keine Kanten enthalten, die Instanzen anderer Kanten sind.

Das Prädikat `extend(E+,V-Vs+,Extended-)` soll den Graphen `E` um die gegebenen Kanten `V-Vs` zu einem neuen Graphen `Extended` erweitern. Damit die Berechnung einigermaßen effizient wird, verwenden wir anfangs leere Akkumulatorlisten für Zwischenergebnisse:

⟨*Parser/warshall.4.pl*⟩+≡
```
extend(E,V-[],E) :- !.
extend(E,V-Vs,Extended) :-
        extend(E,V-Vs,[],Extended).
```

Falls in `E` keine Kanten `X-Xs` mehr sind, bildet die akkumulierte Kantenmenge das Ergebnis:

⟨*Parser/warshall.4.pl*⟩+≡
```
extend([],V-Vs,AccGraph,AccGraph).
```

Im anderen Fall wird durch `link_check(V,Xs)` getestet, ob `V` und mindestens einer der Nachfolger `Xs` von `X` gemeinsame Instanzen haben.

Falls es gemeinsame Instanzen von `V` und einem der Nachkommen `Xs` gibt, werden neue Kanten gesammelt und die restlichen Kanten mit `V-Vs` kombiniert. (Kanten von neuen Knoten werden durch den rekursiven Aufruf von `warshall/3` berücksichtigt.)

⟨*Parser/warshall.4.pl*⟩+≡
```
extend([X-Xs|G],V-Vs,AccGraph,NewGraph) :-
        link_check(V,Xs),
        !, erweitern(X,Xs,V-Vs,AccGraph,AccGraphXV),
        extend(G,V-Vs,AccGraphXV,NewGraph).
```

Falls `V` und die Nachkommen von `X` keine gemeinsamen Instanzen haben, wird der Graph der gesammelten Kanten um die Kanten `X-Xs` erweitert (und durch allgemeinste Kanten dargestellt) und die restlichen Kanten `G` mit `V-Vs` verglichen:

⟨*Parser/warshall.4.pl*⟩+≡
```
extend([X-Xs|G],V-Vs,AccGraph,NewGraph) :-
        add_edges(X-Xs,AccGraph,AccGraphX,_),
        extend(G,V-Vs,AccGraphX,NewGraph).
```

Erweitern eines Graphen um eine Kantenmenge: Eine Kantenmenge `X-Xs` soll zum Graphen `Graph` hinzugefügt und der erweiterte Graph in der Darstellung allgemeinster Kanten ausgegeben werden. Die Kanten des erweiterten Graphen und separat die neuen Kanten werden in anfangs leeren Listen gesammelt:

⟨*Parser/warshall.4.pl*⟩+≡
```
add_edges(X-Xs,Graph,Extended,NewEdges) :-
        add_edges(X-Xs,Graph,[],[],Extended,NewEdges).
```

Der Knoten `X` wird mit jedem Knoten `A` des Graphen verglichen, bis ein äquivalentes `A` gefunden ist. Falls `A` eine Instanz von `X` unter einer Belegung `Env` ist, werden von den Kanten `A-As` diejenigen weggelassen, die `Env`-Instanzen der Kanten `X-Xs` sind:

⟨*Parser/warshall.4.pl*⟩+≡
```
  add_edges(X-Xs,[A-As|Graph],Acc,AccE,NewGraph,NewEdges) :-
          match(A,X,[],Env),
          !,
          substitute(Xs,Env,XsEnv),
          drop_instances(As,XsEnv,LessAs),
          (LessAs = []
          -> add_edges(X-Xs,Graph,Acc,AccE,NewGraph,NewEdges)
```

Falls auch `X` eine Instanz von `A` ist, d.h. `A` und `X` äquivalent sind, werden von den `Xs` diejenigen weggelassen, die entsprechende Instanzen der verbliebenen `LessAs` sind, und `NewGraph` und `NewEdges` zurückgegeben[31]

⟨*Parser/warshall.4.pl*⟩+≡
```
          ; (match(X,A,[],Env2)
            -> substitute(LessAs,Env2,LessAsEnv2),
                drop_instances(Xs,LessAsEnv2,LessXs),
                append(LessXs,LessAsEnv2,NewXs),
                append([X-NewXs|Acc],Graph,NewGraph),
                NewEdges = [X-LessXs|AccE]
```

Andernfalls werden nur die Kanten `A-LessAs` zum neuen Graphen getan und der Restgraph mit `X-Xs` verglichen:

⟨*Parser/warshall.4.pl*⟩+≡
```
            ; add_edges(X-Xs,Graph,[A-LessAs|Acc],AccE,NewGraph,NewEdges))).
```

Falls `A` keine Instanz von `X` ist, wird geprüft, ob `X` eine Instanz von `A` ist. Falls das so ist, werden die Kanten von `X-Xs` weggelassen, die entsprechende Instanzen der Kanten `A-As` sind:

⟨*Parser/warshall.4.pl*⟩+≡
```
  add_edges(X-Xs,[A-As|Graph],Acc,AccE,NewGraph,NewEdges) :-
          match(X,A,[],Env),
          !,
          substitute(As,Env,AsEnv),
          drop_instances(Xs,AsEnv,LessXs),
          (LessXs = []
          -> append([A-As|Graph],Acc,NewGraph)    % nur allgemeinste Kanten?
          ; add_edges(X-LessXs,Graph,[A-As|Acc],AccE,NewGraph,NewEdges)).
```

Sonst, wenn also weder `X` eine Instanz von `A` ist noch umgekehrt `A` eine Instanz von `X`, werden die Kanten `A-As` auf dem Akkumulator gesammelt und `X-Xs` mit den restlichen Kanten des Graphen verglichen:

⟨*Parser/warshall.4.pl*⟩+≡
```
  add_edges(X-Xs,[A-As|Graph],Acc,AccE,NewGraph,NewEdges) :-
          add_edges(X-Xs,Graph,[A-As|Acc],AccE,NewGraph,NewEdges).
```

[31] Wir nehmen an, daß `[A-As|Graph]` schon kompakt dargestellt war und brauchen deshalb `X-Xs` nicht mehr mit `Graph` zu vergleichen. Die Annahme muß aber für den Eingangsgraphen noch gesichert werden.

Falls keine Kanten mehr zu vergleichen sind, ergeben die neuen Kanten `X-Xs` zusammen mit den gesammelten den erweiterten Graphen und die noch verbliebenen `X-Xs` sind die neuen Kanten:

⟨*Parser/warshall.4.pl*⟩+≡
```
add_edges(X-Xs,[],Acc,AccE,[X-Xs|Acc],[X-Xs]).
```

Beispiel 9.4 Ein paar Beispiele sollen zeigen, was bei `add_edges/3` passiert:

⟨*testbeispiele*⟩≡
```
add_edges(a(1)-[b(X),c(1)],[a(X)-[c(X),b(1)],a(1)-[c(3)]],Extended,NewEdges).
Extended = [a(1)-[c(3), b(Y)], a(Y)-[c(Y), b(1)]]
NewEdges = [a(1)-[b(Y)]]

add_edges(a(X)-[b(X),c(1)],[a(Y)-[c(Y),b(2)],a(1)-[c(3)]],Extended,NewEdges).
Extended = [a(Z)-[c(Z), b(Z)], a(1)-[c(3)]]
NewEdges = [a(Z)-[b(Z)]]

add_edges(a(X)-[b(X),c(1)],[a(X)-[c(X),b(2)],a(1)-[c(3)]],Extended,NewEdges).
Extended = [a(Z)-[c(Z), b(Z)], a(1)-[c(3)]]
NewEdges = [a(Z)-[b(Z)]]

add_edges(a(X)-[b(X),c(1)],[b(Y)-[c(2),a(Y)],a(1)-[b(2)]],Extended,NewEdges).
Extended = [a(Z)-[b(Z), c(1)], a(1)-[b(2)], b(V)-[c(2), a(V)]]
NewEdges = [a(Z)-[b(Z), c(1)]]

add_edges(a(X)-[b(X),c(1)],[b(Y)-[c(2),a(Y)],a(1)-[b(1)]],Extended,NewEdges).
Extended = [a(Z)-[b(Z), c(1)], b(V)-[c(2), a(V)]]
NewEdges = [a(Z)-[b(Z), c(1)]]
```

Erweitern des Graphen und Sammeln neuer Knoten:

Alle beim Erweitern durch Vergleich von `X-Xs` und `V-Vs` enstandenen Kanten werden erst zusammen mit `X-Xs` kompakt dargestellt und dann zum Graphen getan, da diese Kanten i.a. gemeinsame Ausgangsknoten haben:

⟨*Parser/warshall.4.pl*⟩+≡
```
erweitern(X,Xs,V-Vs,AccGraph,Graph) :-
        setof(A-Bs,Y^(member(Y,Xs),link_edges((X,Y),V-Vs,A-Bs)),LinkGraph),
        add_edges(X-Xs,LinkGraph,LGraph,_),
        add_edge_sets(LGraph,AccGraph,Graph).

add_edge_sets([X-Xs|G],AccGraph,Graph) :-
        add_edges(X-Xs,AccGraph,GraphX,_),
        add_edge_sets(G,GraphX,Graph).
add_edge_sets([],AccGraph,AccGraph).
```

⟨*link*⟩

Bemerkung 9.5 Leider ist auch diese Version nicht effizient, d.h. sie stößt schnell an Stackgrenzen. Obwohl Endrekursion ausgenutzt wird, scheint sehr viel unnötige Arbeit gemacht zu werden: wenn man einen *saturierten* Knoten `V` hat, dessen Nachfolger `Vs` schon transitiv abgeschlossen sind, und sie neben `V` in eine Nachfolgermenge einbaut, so bräuchte man diese `Vs` natürlich nicht nochmal zu ergänzen versuchen.

Man sollte daher jeweils die saturierten Knoten markieren; unter den Nachfolgern eines Knotens muss man zwischen den schon behandelten und den anderen unterscheiden, und jeweils saturierte

Nachfolger zuerst behandeln (Tip von Hans Heller). Aber das geht zunächst mal für die transitive Hülle eines festen Graphen; bei uns hier ändert sich aber die Knotenmenge!

Bemerkung 9.6 Folgende endrekursive Version, bei der die `Xs` mit `V` verglichen und die Ergebnisse einzeln zum Graphen getan wird, verbraucht ebenfalls zuviel Stack:

Falls alle Nachfolger von `X` betrachtet sind, werden die Kanten `X-Xs` zum akkumulierten Graphen geschlagen; die akkumulierten Kanten und Knoten bilden das Ergebnis: (Die `X-Xs` sind keine neuen Kanten, da sie im zu erweiternden Graphen ja schon vorkamen.)

⟨*Parser/warshall.4.pl (alt)*⟩≡

```
erweitern(X,[],V-Vs,AccGraph,AccGraphXV) :-
        add_edges(X-Xs,AccGraph,AccGraphXV,_).
```

Sonst suchen wir für den nächsten Nachfolger `Xi` von `X` die allgemeinste gemeinsame Instanz mit `V`. Falls es eine solche gibt, fügen wir entsprechende Instanzen `A-Bs` der `X-Vs` als neue Kanten zum Graphen hinzu:

⟨*Parser/warshall.4.pl (alt)*⟩+≡

```
erweitern(X,[Xi|Xs],V-Vs,AccGraph,NewAccGraph) :-
        link_edges((X,Xi),V-Vs,A-Bs),
        !, add_edges(A-Bs,AccGraph,AccGraphABs,_),
        erweitern(X,Xs,V-Vs,AccGraphABs,NewAccGraph).
```

Falls `Xi` und `V` keine gemeinsame Instanz haben, vergleichen wir `V` mit den anderen Nachkommen `Xs` von `X`:

⟨*Parser/warshall.4.pl (alt)*⟩+≡

```
erweitern(X,[Xi|Xs],V-Vs,AccGraph,NewAccGraph) :-
        erweitern(X,Xs,V-Vs,AccGraph,NewAccGraph).
```

Es bleibt zu überlegen, wo Doppelarbeit gemacht wird, die die Rekursionstiefe unnötig erhöht.

Zum Testen dienen folgende Prädikate: (vgl. `Testbett/first.beispielp3`)

⟨*Parser/transclos.pl*⟩+≡

```
tctest(File) :-
        consult(File),
        atom_concat(File,'.tc',TcFile),
        protocol(TcFile),
        base_graph(first,Graph),
        nl,write('Graph: '),write(Graph),
        nl,write('Closure: '),
        profwarshall(warshall-1,Graph,Closure1),
        profwarshall(warshall-2,Graph,Closure2),
        profwarshall(warshall-3,Graph,Closure3),
        profwarshall(warshall-4,Graph,Closure4),
        nl,noprotocol,
        nl,write(done).

profwarshall(Modul,Graph,Closure) :-
        auswahl(tc(Modul)),
        nl,nl,write(Modul),write(': '),
        fresh_copy(Graph,GraphCopy), % da warshall-1 den Graphen verdirbt
        (ein(showProfile)
        -> profile(warshall(GraphCopy,Closure),plain,6)
        ; warshall(GraphCopy,Closure)),
        nl,write(Closure).
```

Es könnte nützlich sein, die aus der Grammatik abgelesene Grundrelation `first/2` kompakt zu machen, d.h. Instanzen anderer Kanten zu beseitigen, bevor man die transitive Hülle bildet.

9.1.3 Verkettung von Tupeln und Zyklentest

Es muß vermieden werden, daß die neue Relation unendlich wird. Bei der Verknüpfung von Tupeln `(A,B)` und `(C,D)` werden nach (getrennter!) Umbenennung ihrer freien Variablen die neuen Komponenten `B1` und `C1` miteinander unifiziert und aus den anderen Komponenten ein neues Tupel `(A1,D2)` gebildet.

⟨*link*⟩≡

```
link((A,B),(C,D),(A1,D2)) :-
    fresh_copy([A,B],[A1,B1]),
    fresh_copy([C,D],[C2,D2]),
    C2=B1,
    (circularity_test(A1,D2)
    -> write('ERROR: circularity in computing transitive closure detected.'),
        nl,fail
    ; (ein(verbose) -> nl,write('new tuple:   '), write((A1,D2)) ; true)).

link_edges((A,B),C-Ds,A1-Ds2) :-
    fresh_copy([A,B],[A1,B1]),
    fresh_copy([C,Ds],[C2,Ds2]),
    C2=B1,
    (circularity_tests(A1,Ds2)
    -> write('ERROR: circularity in computing transitive closure detected.'),
        nl,fail
    ; (ein(verbose) -> nl,write('new tuples:   '), write((A1,Ds2)) ; true)).

circularity_tests(A,[D|Ds]) :-
        (circularity_test(A,D) -> true ; circularity_tests(A,Ds)).
circularity_tests(A,[]) :- fail.
```

Durch `link_check(V,Xs)` wird getestet, ob `V` und mindestens eines der Elemente von `Xs` gemeinsame Instanzen haben:

⟨*link*⟩+≡

```
link_check(V,[X|Xs]) :-
        ((fresh_copy(V,V1), fresh_copy(X,X2), X2=V1) -> true ; link_check(V,Xs)).
link_check(V,[]) :- fail.
```

Das neue Tupel `(A1,D2)` wird nur dann in die Relation aufgenommen, wenn es keine Zyklen erzeugt: wenn keinem variablen Teilterm `X` in `A1` an derselben Stelle in `D2` ein Teilterm `t(X,...)` verschieden von `X` entspricht. Andernfalls wird die Bildung der neuen Relation abgebrochen.[32]

⟨*link*⟩+≡

```
circularity_test(C,A) :-
    (circle(C,A) -> true ; circle(A,C)).

circle(C,A) :-
```

[32] Man könnte auch eine endliche Oberrelation berechnen, z.B. durch Verallgemeinerung der kritischen Tupel. Das machen wir erst, wenn die Grammatiken es erfordern.

```
    var(A),
    !,fail.
circle(C,A) :-
    atomic(A),
    !,fail.
% Damit nicht ueber match([[]],[Term|Terms]) = [(Term,[])] eine Schleife
% entsteht:
circle(C,[[]]) :-
    !,fail.
circle([Arg1|Args],[Term1|Terms]) :-
    !,(circle(Arg1,Term1) ; circle([Args],[Terms])).

circle(C,A) :-
    hasVar(C),
    A =.. [Functor|Terms],
    occurs(C,Terms).
circle(C,A) :-
    not(var(C)),
    A =.. [Functor|Terms],
    C =.. [Functor|Args],
    circle(Args,Terms).

hasVar(Term) :-
     var(Term),!.
hasVar(Term) :-
     atomic(Term),
     !,fail.
hasVar([Term|Rest]) :-
     !,(hasVar(Term) -> true ; hasVar(Rest)).
hasVar(Term) :-
     Term =.. [Functor|Arguments],
     hasVar(Arguments).

occurs(X,[Term|Rest]) :-
    subterm(X,Term),!.
occurs(X,[Term|Rest]) :-
    !,occurs(X,Rest).
occurs(X,[]) :- fail.

subterm(X,Y) :-
    X == Y,!.                     % Hier mit unifyable Umbenennungen erlauben?
subterm(X,Y) :-
    compound(Y),
    !, Y =.. [Functor|Args],
    occurs(X,Args).
```

Wir nehmen nur solche Tupel auf, die nicht schon ein Spezialfall eines vorhandenen Tupels sind, i.e. die nicht in das Muster eines vorhandenen Tupels passen.

9.1.4 Subsumption

Ein Term s *subsumiert* den Term t, wenn es eine Belegung $[s_1/x_1, \ldots, s_k/x_k]$ der freien Variablen $x_1, \ldots, x_k$ von s gibt, so daß $s[s_1/x_1, \ldots, s_k/x_k]$ der Term t ist. Man nennt dann t auch eine *Instanz* des *Musters* s, und sagt, t paßt in das Muster s:

⟨*Parser/subsumption.pl*⟩≡
```
matches(Term,Muster) :-
        match(Term,Muster,[],_).
```

Durch `match(Term+,Muster+,Belegung+,ErweiterteBelegung-)` wird festgestellt, ob `Term` in das `Muster` paßt, wenn man die vorgegebene Belegung zur Instantiierung benutzt und gegebenenfalls erweitert, wo sie nicht definiert ist:

⟨*Parser/subsumption.pl*⟩+≡
```
match(Term,Pattern,Env,Newenv) :-
     atomic(Pattern),
     !,Term == Pattern,
     Newenv = Env.

match(Term,Pattern,Env,Newenv) :-
     var(Pattern),
     unbound(Pattern,Env),
     !,Newenv = [(Pattern,Term)|Env].
match(Term,Pattern,Env,Newenv) :-
     var(Pattern),
     subst(Pattern,Env,Instance),
     Term == Instance,
     !,Newenv = Env.
match(Term,Pattern,Env,_) :-
     var(Pattern),!,fail.

match(Term,[S1|S2],Env,Newenv) :-
     !,Term = [T1|T2],
     match(T1,S1,Env,Newenv1),
     match(T2,S2,Newenv1,Newenv).

match(Term,Pattern,Env,Newenv) :-
     Pattern =.. [F|Rest],
     !, compound(Term),
     Term =.. [F|Args],
     match(Args,Rest,Env,Newenv).
```

Belegen der freien Variablen von `Term` gemäß der Belegung `Env`:

⟨*Parser/subsumption.pl*⟩+≡
```
substitute(Term,Env,Newterm) :-
     var(Term),!,
     subst(Term,Env,Newterm).
substitute(Term,Env,Newterm) :-
     atomic(Term),!,
     Newterm = Term.
substitute([Term|Terms],Env,[Newterm|Newterms]) :- !,
     substitute(Term,Env,Newterm),
     substitute(Terms,Env,Newterms).
substitute([],Env,[]) :- !.
substitute(Term,Env,Newterm) :-
     compound(Term),
     not(Term==[_]),
```

```
      Term =.. [F|Args],
      !,substitute(Args,Env,Newargs),
      Newterm =.. [F|Newargs].

  subst(X,[],Result) :-
      !, Result = X.              % X sei eine Variable
  subst(X,[(Y,Term)|_],Result) :-
      X == Y,
      !, Result = Term.           % man muß unify(f(Z),Z) vermeiden !
  subst(X,[_|Rest],Term) :-
      subst(X,Rest,Term).

  unbound(X,[]) :- !.
  unbound(X,[(Y,Term)|Rest]) :- !,
       not(X==Y),unbound(X,Rest).
```

Das Prädikat `free_variables(Term+,Liste-)` von Prolog sammelt die freien Variablen von `Term` in `Liste` (ohne Doppleinträge). Wir benutzen es, um frische Kopien und Grundinstanzen von Termen zu erzeugen.

Eine 'frische' Kopie eines Terms entsteht durch Ersetzen der Variablen des Terms durch frische, d.h. noch nicht benutzte Variablen:

⟨*Parser/subsumption.pl*⟩+≡

```
  fresh_copy(Term,NewTerm) :-
      free_variables(Term,Vars),
      rename(Vars,NewVars,Env),
      substitute(Term,Env,NewTerm).

  rename([],[],[]) :- !.
  rename([X|Rest],[Y|YRest],[(X,Y)|Env]) :-
      !,rename(Rest,YRest,Env).
```

Eine neue Grundinstanz eines Terms entsteht durch Ersetzen seiner Variablen durch noch nicht benutzte 'Konstante', d.h. Grundterme `const(N)` mit Zahl `N`:

⟨*Parser/subsumption.pl*⟩+≡

```
  :- dynamic const/1.

  newconst(C) :-
      (const(X)
          -> Y is X + 1, C = const(Y), retract(const(X)), assert(const(Y))
          ; C = const(0), assert(const(0))).

  ground_instance(Term,GroundTerm) :-
      free_variables(Term,Vars),
      vars_to_consts(Vars,Consts,Env),
      substitute(Term,Env,GroundTerm).

  vars_to_consts([],[],[]) :- !.
  vars_to_consts([X|Xs],[C|Cs],[(X,C)|Env]) :-
      !,newconst(C), vars_to_consts(Xs,Cs,Env).
```

Prolog hat ein Prädikat `numbervars/4`, mit dem man durch `numbervars(Term,const,0,N)` schneller eine Grundinstanz eines Terms bekommt, bei der die Variable von `Term` durch Konstante `const(0)` usw. ersetzt sind; allerdings wird dabei der Term `Term` durch Unifikation verändert. In manchen Fällen kann das trotzdem nützlich sein, weil man die Belegung mit doppelter Negation wieder aufheben kann:

⟨*Parser/subsumption.pl*⟩+≡
```
known_instance(Term) :- % use this only for callable terms
        \+ \+ (numbervars(Term,const,0,_), Term).
```

Zum Weglassen aller Instanzen der Muster `Xs` aus einer Liste `As` dient

⟨*Parser/subsumption.pl*⟩+≡
```
drop_instances([A|As],Xs,LessAs) :-
        match_check(A,Xs)
        -> drop_instances(As,Xs,LessAs)
        ; drop_instances(As,Xs,Bs), LessAs = [A|Bs].
drop_instances([],Xs,[]).
```

Mit `match_union(Liste1-,Liste2-,Vereinigung+)` werden die Listen so vereinigt, daß nur die allgemeinsten Terme gesammelt werden, d.h. Instanzen anderer Elemente werden weggelassen:

⟨*Parser/subsumption.pl*⟩+≡
```
match_union([], A, A) :- !.
match_union(A, [], A) :- !.
match_union([A|B], [C|Cs], D) :-
        match_check(A, [C|Cs]),
        !, match_union(B, [C|Cs], D).
match_union([A|B], [C|Cs], [A|D]) :-
        match_check(C,[A|B]),
        !, match_union(B,Cs,D).
match_union([A|B], C, [A|D]) :-
        match_union(B, C, D).

match_check(A,[C|Cs]) :-
        matches(A,C), !.
match_check(A,[C|Cs]) :-
        match_check(A,Cs).
```

Intensionale Berechnung der transitiven Hülle

Die dritte Möglichkeit ist, die transitive Hülle intensional zu berechnen. Dazu ist der Ausgangsgraph als Menge von Regeln zu beschreiben.

⟨*tc: Version 3*⟩≡
```
45 ?- setof(A-As,setof((B,Cond),clause(A --> B,Cond),As),Graph).

Graph = [a(X)-[ ([], einschraenkung2(X))],
         c(Y)-[ ((a(Y), a(Y)), true)],
         d(Z)-[ (e(Z), einschraenkung2(Z)), (h(Z), einschraenkung3(Z))],
         e(U)-[ (f(U), einschraenkung1(U))],
         s(V)-[ ((a(...), b(...)), einschraenkung1(V))]]
```

Hieraus müsste nun die transitive Hülle bestimmt werden, indem man zu jedem Tupel auch die Bedingungen kombiniert und vereinfacht. Dabei muß durch die Vereinfachung sichergestellt werden, daß die Bedingungen nicht unbeschränkt wachsen.

Alternative induktive Berechnung der transitiven Hülle

Die transitive Hülle einer Relation zwischen Prolog-Termen, die freie Variable enthalten, kann unendlich groß sein, z.B. bei der Relation $\{(X, f(X))\}$, deren transitive Hülle die Relation $\{(X, f^n(X)) \mid n \in I\!N\}$ umfaßt (wenn man freie Variable so versteht, daß dort beliebige Terme eingesetzt werden dürfen).

Es genügt aber für unsere Zwecke, eine *Oberrelation* der transitiven Hülle zu berechnen: wir wollen Tupel ignorieren, die *nicht* in der transitiven Hülle sind. Dazu berechnen wir allgemeiner den *Abschluß* $\overline{R}^h$ einer Relation R unter einer partiellen zweistelligen Operation h, d.h. die kleinste Relation $S \supseteq R$, die unter Anwendung von h abgeschlossen ist:

$$\bigcap\{S \mid R \subseteq S, \forall a, b \in S(h(a,b)\downarrow \rightarrow \ h(a,b) \in S)\}$$

Sei $S * R := \{h(s,r) \mid s \in S, r \in R\}$. Die naheliegende Idee ist, $\overline{R}^h$ durch folgende Schleife zu berechnen:

```
 begin  S := R ;
        New := (S*R)-S ;
        while New =/= empty do
              { S := S + New ;
                New := (New*R + R*New) - S }
        return S.
end.
```

Wenn eine Grammatik Kategorien mit Merkmalen hat, können die Relationen unendlich groß sein. Um auch diesen Fall zu erfassen, erlauben wir als Elemente Prolog-Terme mit freien Variablen. Die Konstruktion von `New` muß dann verallgemeinert werden zu

```
New := (S*R + R*S) - {a | for some b in S, condition(a,b) holds}.
```

Als Bedingung werden wir `matches(X,Y)`, X *ist eine Instanz von* Y, benutzen, um eine unendliche Relation kompakt darzustellen.

Eine alternative Berechnung der transitiven Hülle der First-Relation wäre dann wie folgt möglich:

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

```
:- module(inductive,[tc_inductive/3,tc/4]).

compute_first :-
    findall((X,Y),first(X,Y),BasicFirsts),
    tc(BasicFirsts,link,matches,Firsts),
    % retractall(first(_,_)),
    add_first(Firsts).

tc_inductive(Base,Ancestor,Descendant) :-
    Edge =.. [Base,X,Y],
    findall((X,Y),Edge,BasicTuples),
    tc(BasicTuples,link,matches,Tuples),
    member((Ancestor,Descendant),Tuples).
```

$\langle$*Transitive Hülle*$\rangle$

$\langle$*link*$\rangle$

⟨*Transitive Hülle*⟩≡

```
tc(Relation0,Operation,Condition,Result) :-
     compact(Relation0,Condition,Relation),      % eine unnötige Kompaktifizierung?
     apply(Operation,Relation,Relation,News),
     compact(News,Condition,NewElements),
     difference(NewElements,Relation,Condition,NewTuples),  % NewTuples = R*R - R
     close(NewTuples,Relation,Relation,Operation,Condition,Result).
```

Die `while`-Schleife wird durch `close/6` verwirklicht:

⟨*Transitive Hülle*⟩+≡

```
close([],S,_Relation,_Operation,_Condition,S).      % this realizes the while-loop
close([X|Rest],S,Relation,Operation,Condition,Result) :-
     union([X|Rest],S,Condition,NewS),
     update(Operation,[X|Rest],NewS,Relation,Condition,NewTuples),
     (ein(verbose)
     -> (nl,nl,write('New tuples for the transitive closure: '),
            write(NewTuples))
     ; true),
     close(NewTuples,NewS,Relation,Operation,Condition,Result).

update(Operation,N,S,Relation,Condition,NewTuples) :-
     % NewTuples := (N*R + R*N) - {a | for some b in S, condition(a,b) holds}.
     apply(Operation,N,Relation,New1),
     apply(Operation,Relation,N,New2),
     compact(New1,Condition,New11),
     compact(New2,Condition,New22),
     union(New11,New22,Condition,New),             % we need a weaker union +  !
     difference(New,S,Condition,NewTuples).
```

In `union(Liste1+,Liste2+,Bedingung+,Vereinigung-)` nehmen wir an, daß `Liste1` und `Liste2` schon (unter der Bedingung) kompaktifiziert sind:

⟨*Transitive Hülle*⟩+≡

```
union([],List,_Condition,List).
union([X|Rest],List,Condition,Union) :-
     member(X,List,Condition),
     !,union(Rest,List,Condition,Union).
union([X|Rest],List,Condition,Union) :-
     difference(List,[X],Condition,XList),
     union(Rest,XList,Condition,Umin),
     append([X],Umin,Union).

compact([A|Rest],Condition,NewList) :-
     compact(Rest,Condition,CompactRest),
     union([A],CompactRest,Condition,NewList).
compact([],_Condition,[]).
```

Durch `difference(N,S,Condition,Diff)` soll die Menge

$$Diff = N - \{a \mid \exists b \in S \; Condition(a,b)\}$$

berechnet werden. Wir setzen voraus, daß N und S relativ zu *Condition* schon kompaktifiziert sind.

⟨*Transitive Hülle*⟩+≡
```
difference(List,[],_Condition,List) :- !.
difference([],_List,_Condition,[]) :- !.
difference([X|Rest],List,Condition,Diff) :-
    member(X,List,Condition),!,
    difference(Rest,List,Condition,Diff).
difference([X|Rest],List,Condition,Diff) :-
    difference(Rest,List,Condition,Restlist),
    append([X],Restlist,Diff).                    % assuming X is not in Rest!
member(X,List,Condition) :-
    member(Y,List),
    Goal =.. [Condition,X,Y],
    call(Goal).
```

Zur Berechnung von $S * R$ dient `apply(*,S,R,List)`, wobei `*` eine partielle Operation sein kann:[33]

⟨*Transitive Hülle*⟩+≡
```
apply(_Operation,[],_List,[]).
apply(_Operation,[_X|_Rest],[],[]).
apply(Operation,[X|RestX],[Y|RestY],List) :-
    apply1(Operation,X,[Y|RestY],XList),
    apply(Operation,RestX,[Y|RestY],RestList),
    union(XList,RestList,List).

apply1(_Operation,_Element,[],[]).
apply1(Operation,Element,[Y|Rest],List) :-
    p_apply(Operation,Element,Y,YList),
    apply1(Operation,Element,Rest,RestList),
    union(YList,RestList,List).
p_apply(PartialOperation,X,Y,ValueList) :-
    Goal =.. [PartialOperation,X,Y,NewElement],
    call(Goal),!,
    ValueList = [NewElement].
p_apply(_PartialOperation,_X,_Y,[]) :- !.          %  if op(X,Y) is undefined
```

Nur zum Testen:

⟨*Testbett/Beispiele/tctest.pl*⟩≡
```
w1 :- ['Parser/warshall.1.pl'].
w2 :- ['Parser/warshall.2.pl'].
w3 :- ['Parser/warshall.3.pl'].
w4 :- ['Parser/warshall.4.pl'].
wc :- ['Parser/transclos.pl.cis'].

b0 :-   retractall(first(_,_)),
        assert(first(a,b(X))),
        assert(first(b(1),c(1))),
        assert(first(b(Y),d(Y))).

b1 :-   retractall(first(_,_)),
        % assert(first(a(X),b(X))),
```

[33] Man sollte das Ergebnis noch ausdünnen, da `union/3` von SWI Doppelelemente in der Liste läßt.

```
        assert(first(b(1),c(1))),
        % assert(first(b(Y),d(Y))),
        assert(first(c(Z),b(Z))).

b2 :-   retractall(first(_,_)),
        assert(first(a(*),d(X))),
        assert(first(b(Y),a(Y))),
        assert(first(c(Z),b(Z))).

cl(TC) :- base_graph(first,G),warshall(G,TC).
cl(G,TC) :- base_graph(first,G),warshall(G,TC).
```

9.2 Erstellung des Vollformenlexikons

Zur Erstellung des Vollformenlexikons benutzen wir zwei Informationsquellen, das CISLEX mit hauptsächlich morphologischer Information über die Vollformen, und eine Verblexikon von Daniel Schnorbusch, in dem Komplementrahmen zu den Vollverben des Deutschen angegeben sind.

Die Einträge dieser beiden Lexika werden in ein Format umgerechnet, das zu den Grammatikregeln der hier verwendeten Grammatik paßt. Die lexikalische Analyse des Parsers erstellt für jeden Eingabesatz ein Teillexikon `lexword/3`, das sowohl die syntaktische als auch die semantische Information –genauer gesagt, nur die Komplementrahmen der Vollverben– enthält.

Bei der Erstellung von `lexword/2` soll ein Lexikon aller bekannten Wörter, `word/3`, benutzt werden, in dem die Information in der Form `word(Vollform,Kategorie,Stammform)` dargestellt wird.

Die Information über den Stamm ist für das Parsen eigentlich nicht nötig; sie soll für folgende Zwecke benutzt werden:

(i) Alle Information zu einem Wort, die im CISLEX nicht enthalten ist, soll in einem Stammlexikon `stamm/3` abgelegt werden, damit nur dieses an Bedürfnisse des Parsens angepaßt werden muß.

(ii) Um ein Phrasenflexionsprogramm zu schreiben, braucht man ein Flexionsprogramm für Wörter, das (wie unten beschrieben wird) aus dem Vollformlexikon und den Wortstämmen gewonnen wird. (Effizienz spielt hier keine Rolle.)

Insbesondere sollte die Information über Wortrahmen zwar in `lexword/2` bei jeder Vollform stehen, aber pro Wortstamm nur einmal (bei Nomina, Adjektiven, Verbkomposita) von Hand ergänzt werden. Auch zur Korrektur ist es nützlich, wenn die falsche Information nur an einem Vorkommen korrigiert werden muß.

9.2.1 Morphologische und syntaktische Information aus dem CISLEX

Im CISLEX bildet jeder Eintrag zu einer Vollform eine Zeile der Datei. Wir übersetzen die CISLEX-Datei zeilenweise in das vom Parser verwendete Format.

Ausgabe mit gemischten Wortarten

Eine Möglichkeit besteht darin, die mit dem CISLEX erzeugten Lemmata der Wörter eines Textes nach der Umformatierung an das vom Parser benutzte Format in eine einzigen Ausgabedatei zu schreiben:

⟨*Lexika/umformatierer.pl*⟩≡
```
formatiere :-
        ensure_loaded('Tokenizer/tokenizer.pl'),
        write(user,'Eingabedatei: (im Format des CISLEX)'),
        read(Eingabedatei),
        concat_atom([Eingabedatei,'.wrd','.tmp'],Ausgabedatei),
        write(user,'Ausgabedatei: '),write(user,Ausgabedatei),
        formatiere(Eingabedatei,Ausgabedatei).
```

Das CIS-Lexikon eines Textes, d.h. eine der Dateien `Testbett/*.lem`, wird zeilenweise gelesen, mit `zerlege/6` in Bestandteile zerlegt und durch `schreibeEintraege/6` nach der Umformung auf eine Ausgabe geschrieben:

⟨*Lexika/umformatierer.pl*⟩+≡
```
formatiere(Eingabedatei,Ausgabedatei) :-
        open(Eingabedatei,read,InStream),
        open(Ausgabedatei,write,OutStream,[buffer(line),eof_action(eof_code)]),
        repeat,
        get0(InStream,Char),
        (dateiende(Char)
        -> flush_output(OutStream)
        ; readline(InStream,[Char],Line),
            zerlege(Line,Vollform,Stamm,Praefix,Kategorie,Formen),
            schreibeEintraege(OutStream,Vollform,Stamm,Praefix,Kategorie,Formen),
            write(user,'.'),flush_output(user),
            fail),
        close(InStream),
        close(OutStream),
        nl,write(user,'Done.').

readline(Stream,Seen,Line) :-
        get0(Stream,Char),
        (zeilenende(Char)
        -> reverse(Seen,Zeile),name(Line,Zeile)
        ; readline(Stream,[Char|Seen],Line)).
```

Ausgabe nach Wortarten getrennt

Zur Erleichterung der Nachkorrektur ist es sinnvoll, die Lexikoneinträge nach verschiedenen Wortarten aufzuteilen. Dazu werden entsprechende Dateien und Ströme geöffnet; da die Dateien eine Nachbearbeitung von Hand erfordern, sind sie durch `.tmp.` als temporäre Dateien gekennzeichnet:

⟨*Lexika/umformatierer.pl*⟩+≡
```
formatiere_nach_Wortarten :-
        ensure_loaded('Tokenizer/tokenizer.pl'),
        write(user,'Eingabedatei (im Format des CISLEX): '),
        read(Eingabedatei),
        concat_atom([Eingabedatei,'.<Wortart>','.tmp.wrd'],Ausgabedatei),
        write(user,'Ausgabedateien: '),write(user,Ausgabedatei),
        formatiere_nach_Wortarten(Eingabedatei).

formatiere_nach_Wortarten(Eingabedatei) :-
        atom_concat(Eingabedatei,'.a.tmp.wrd',Adjektive),
        atom_concat(Eingabedatei,'.adv.tmp.wrd',Adverbien),
        atom_concat(Eingabedatei,'.det.tmp.wrd',Determinatoren),
        atom_concat(Eingabedatei,'.n.tmp.wrd',Nomen),
        atom_concat(Eingabedatei,'.na.tmp.wrd',NominalAdjektive),
        atom_concat(Eingabedatei,'.conj.tmp.wrd',Konjunktionen),
        atom_concat(Eingabedatei,'.part.tmp.wrd',Partikel),
        atom_concat(Eingabedatei,'.pr.tmp.wrd',Praepositionen),
        atom_concat(Eingabedatei,'.pron.tmp.wrd',Pronomen),
        atom_concat(Eingabedatei,'.vpr.tmp.wrd',Verbteile),
```

```
atom_concat(Eingabedatei,'.v.tmp.wrd',Verben),
open(Eingabedatei,read,InStream),
open(Adjektive,write,AdjektivStrom),
open(Adverbien,write,AdverbStrom),
open(Determinatoren,write,DeterminatorStrom),
open(Nomen,write,NomenStrom),
open(NominalAdjektive,write,NominalAdjektivStrom),
open(Konjunktionen,write,KonjunktionStrom),
open(Partikel,write,PartikelStrom),
open(Praepositionen,write,PraepositionStrom),
open(Pronomen,write,PronomenStrom),
open(Verbteile,write,VerbteilStrom),
open(Verben,write,VerbStrom),
```

Nun werden die einzelnen Zeilen der Eingabedatei gelesen, bis man das Dateiende erreicht:

⟨*Lexika/umformatierer.pl*⟩+≡

```
repeat,
get0(InStream,Char),
(dateiende(Char)
-> true
; readline(InStream,[Char],Line),
    zerlege(Line,Vollform,Stamm,Praefix,Kategorie,Formen),
    ((Kategorie = a ; Kategorie = asuff)
    -> schreibeEintraege(AdjektivStrom,Vollform,Stamm,Praefix,a,Formen) ; true),
    (Kategorie = adv
    -> schreibeEintraege(AdverbStrom,Vollform,Stamm,Praefix,adv,Formen) ; true),
    (Kategorie = det
    -> schreibeEintraege(DeterminatorStrom,Vollform,Stamm,Praefix,det,Formen) ; true),
    (Kategorie = n
    -> schreibeEintraege(NomenStrom,Vollform,Stamm,Praefix,n,Formen) ; true),
    (Kategorie = na
    -> schreibeEintraege(NominalAdjektivStrom,Vollform,Stamm,Praefix,na,Formen) ; true),
    (Kategorie = conj
    -> schreibeEintraege(KonjunktionStrom,Vollform,Stamm,Praefix,conj,Formen) ; true),
    (Kategorie = part
    -> schreibeEintraege(PartikelStrom,Vollform,Stamm,Praefix,part,Formen) ; true),
    (Kategorie = pr
    -> schreibeEintraege(PraepositionStrom,Vollform,Stamm,Praefix,pr,Formen) ; true),
    (Kategorie = pron
    -> schreibeEintraege(PronomenStrom,Vollform,Stamm,Praefix,pron,Formen) ; true),
    (Kategorie = vpart
    -> schreibeEintraege(VerbteilStrom,Vollform,Stamm,Praefix,vpr,Formen) ; true),
    (Kategorie = v
    -> schreibeEintraege(VerbStrom,Vollform,Stamm,Praefix,v,Formen) ; true),
    (ein(verbose)
    -> schreibeEintraege(user,Vollform,Stamm,Praefix,Kategorie,Formen) ; true),
    fail),
```

Schließlich werden die geöffneten Ströme wieder geschlossen:

⟨*Lexika/umformatierer.pl*⟩+≡

```
close(AdjektivStrom),
close(AdverbStrom),
close(DeterminatorStrom),
```

```
        close(NomenStrom),
        close(NominalAdjektivStrom),
        close(KonjunktionStrom),
        close(PartikelStrom),
        close(PraepositionStrom),
        close(PronomenStrom),
        close(VerbteilStrom),
        close(VerbStrom),
        close(InStream),
        nl,write(user,'Done').
```

Die erzeugten Dateien stehen in dem Verzeichnis, in dem die Eingabedatei stand; man sollte sie von Hand korrigieren und die korrigierten Fassungen nach `Lexika/*.lem.<Kategorie>.wrd` verlagern.

Umformatieren von CISLEX-Einträgen

Im CISLEX sind die Vollformen nach einem bestimmten Schema kodiert, z.B.

```
abbau,abbau.mask(NS1,NP17)#0:aeM
durftest,dürfen.VUNR#0:2eVi
ein,ein.DET2#0:neNz:aeNz:neMz
ein,ein.VPART#0,:neNz:aeNz:neMz
```

Um daraus die für die lexikalische Analyse nötigen Einträge der Form `word(<Wortform>,<Kategorie>)` (bzw. `word(<Wortform>,<Kategorie>,_Stammform)`) zu erhalten, muß die Kodierung der morphologischen und syntaktischen Information des CISLEX in die bei den hier implementierten Analyseverfahren verwendete Kodierung umgerechnet werden.

(i) Jede Zeile wird zerlegt in die Vollform, die Stammform, die syntaktische Kategorie und in Merkmale zur Beschreibung einer Subkategorie und der morphologischen Formmöglichkeiten.

⟨*Lexika/umformatierer.pl*⟩+≡

```
zerlege(Zeile,Vollform,Stamm,Unterklasse,Kategorie,KlasseUndFormen) :-
        concat_atom([Kern|KlasseUndFormen],':',Zeile),
        concat_atom([Wort,KategorienCode],'.',Kern),
        concat_atom([Vollform,Stamm],',',Wort),
        formatiereCat(KategorienCode,Kategorie,Unterklasse).

formatiereCat(C,a,U) :-
        atom_prefix(C,'ADJ'),
        !, atom_concat('ADJ',Suff,C), concat_atom([U,N],'#',Suff).
formatiereCat(C,det,U) :-
        atom_prefix(C,'DET'),
        !, atom_concat('DET',Suff,C), concat_atom([U,N],'#',Suff).
formatiereCat(C,pr,U) :-
        atom_prefix(C,'PREP'),
        !, atom_concat('PREP',Suff,C), concat_atom([U,N],'#',Suff).
formatiereCat(C,pron,U) :-
        atom_prefix(C,'Pron'),
        !, atom_concat('Pron',Suff,C), concat_atom([U,_],'#',Suff).
```

Bei den Verben sind die Unterklassen durch ein gegebenenfalls abtrennbares Präfix gegeben:

⟨*Lexika/umformatierer.pl*⟩+≡
```
formatiereCat(C,v,U) :-
        atom_prefix(C,'VST'),
        !, atom_concat('VST',Suff,C), concat_atom([_,U],'#',Suff).
formatiereCat(C,v,U) :-
        atom_prefix(C,'VSW'),
        !, atom_concat('VSW',Suff,C), concat_atom([_,U],'#',Suff).
formatiereCat(C,v,U) :-
        atom_prefix(C,'VUNR'),
        !, atom_concat('VUNR',Suff,C), concat_atom([_,U],'#',Suff).

formatiereCat(C,adv,U) :-
        atom_prefix(C,'ADV'),
        !, atom_concat('ADV',Suff,C), concat_atom([_,U|N],'#',Suff).
formatiereCat(C,conj,U) :-
        atom_prefix(C,'KONJ'),
        !, atom_concat('KONJ',Suff,C), concat_atom([_,U],'#',Suff).
formatiereCat(C,part,U) :-
        atom_prefix(C,'PART'),
        !, atom_concat('PART',Suff,C), concat_atom([_,U],'#',Suff).
formatiereCat(C,vpr,U) :-
        atom_prefix(C,'VPART'),
        !, atom_concat('VPART',Suff,C), concat_atom([_,U],'#',Suff).

formatiereCat(C,asuff,U) :-
        atom_prefix(C,'ASUFF'),
        !, atom_concat('ASUFF',Suff,C), concat_atom([U,N],'#',Suff).
formatiereCat(C,pdet,U) :-
        atom_prefix(C,'PDET'),
        !, atom_concat('PDET',Suff,C), concat_atom([U,N],'#',Suff).

formatiereCat(C,n,'') :- atom_prefix(C,'mask'),!.
formatiereCat(C,n,'') :- atom_prefix(C,'fem'),!.
formatiereCat(C,n,'') :- atom_prefix(C,'neut'),!.
formatiereCat(C,n,'') :- atom_prefix(C,'no_gen'),!.

formatiereCat(C,nn,'') :- atom_prefix(C,'EN(N)'),!.
formatiereCat(C,vn,'') :- atom_prefix(C,'EN(V)'),!.
formatiereCat(C,na,'') :- atom_prefix(C,'NA'),!.
```

Bem. Die Kategorie `asuff` steht im CISLEX für Suffixadjektive wie `eckig`, `auffällig`, mit denen Adjektive wie `viereckig` oder `unauffällig` gebildet werden. Wörter dieser Art werden im Parser noch nicht benutzt.

(ii) Aus der Vollform und der syntakischen Kategorie wird aus jedem `Code` der Subkategorie und der morphologischen Form ein Eintrag `word(<Vollform>,<Kategorie>(<Subkategorie>,<Form>),Stamm)` erzeugt und in eine entsprechende Datei `Testbett/*.tmp.wrd` geschrieben.

Die einzelnen Kodierungen `Code` werden in ihre Buchstaben zerlegt und diese dann übersetzt.

⟨*Lexika/umformatierer.pl*⟩+≡
```
schreibeEintraege(Strom,Vollform,Stamm,Praefix,Kategorie,[C|Codes]) :-
        uebersetzeCode(Vollform,Stamm,Praefix,Kategorie,C,Eintrag),
```

```
        writeq(Strom,Eintrag),write(Strom,'.\n'),
        fail.
schreibeEintraege(Strom,Vollform,Stamm,Praefix,Kategorie,[C|Codes]) :-
        schreibeEintraege(Strom,Vollform,Stamm,Praefix,Kategorie,Codes).
schreibeEintraege(Strom,Vollform,Stamm,Praefix,Kategorie,[]).
```

Da allerdings manche Teile nicht die Form, sondern die Unterklasse bestimmen (z.B. Genus bei Nomina im Unterschied zum Genus bei Adjektiven), werden die Merkmale je nach der Kategorie mit `formatiereChars/4` auf Unterklassen- und Formmerkmale verteilt und mit `ordneForm/3` ggf. anders angeordnet:

⟨*Lexika/umformatierer.pl*⟩+≡

```
uebersetzeCode(Vollform,Stamm,Unterklasse,Kategorie,Code,Eintrag) :-
        atom_chars(Code,Chars),
        formatiereChars(Kategorie,Chars,Klasse,Form),
        ordneForm(Kategorie,Form,OrdForm),
        !,
```

Außerdem sind bei Präfixverben Vollformen zu ergänzen: alle finiten Formen mit abgespaltenem Präfix, sowohl die für die Verwendung in der Verbzweitstellung als auch die Imperativformen[34], und die infinite Form mit nach dem Präfix eingeschobenem `zu`: (Die Komplementrahmen werden aus einem Stammlexikon ergänzt.)

⟨*Lexika/umformatierer.pl*⟩+≡

```
        ((Kategorie = v,
          number_chars(Praefixlaenge,[Unterklasse]),
          sub_atom(Stamm,0,Praefixlaenge,_,Praefix))
        -> (Praefixlaenge = 0
           -> KlasseNeu = Klasse, FormNeu = OrdForm, VollformNeu = Vollform
           ; ((KlasseNeu = Klasse, \+ (OrdForm = [_,imp]),
                   FormNeu = OrdForm, VollformNeu = Vollform)
             ; (KlasseNeu = [Praefix],
                   \+ (OrdForm = [_,part2]; OrdForm = [_,part]; OrdForm = [inf]),
                   FormNeu = OrdForm,
                   atom_concat(Praefix,VollformNeu,Vollform))
             ; (KlasseNeu = Klasse, OrdForm = [inf],
                   FormNeu = [infzu],
                   zu_einfuegung(Praefix,Vollform,VollformNeu))
                   ))
```

Bei Präpositionen erzeugen wir einen Eintrag in `word/2`, in dem die Präposition selbst (mit dem regierten Kasus, falls der nicht eindeutig ist) als Formmerkmal dient:

⟨*Lexika/umformatierer.pl*⟩+≡

```
        ; (Kategorie = pr
          ->  ( (Unterklasse = '1', Kas = akk, FormNeu = [Vollform])
              ; (Unterklasse = '2', Kas = gen, FormNeu = [Vollform])
              ; (Unterklasse = '3', Kas = dat, FormNeu = [Vollform])
              ; (((Unterklasse = '4', (Kas = dat ; Kas = akk))
                 ; (Unterklasse = '0', (Kas = nom ; Kas = gen ; Kas = dat ; Kas = akk))),
                  StammNeu =.. [Vollform,Kas],
                  FormNeu = [StammNeu])
              ),
              KlasseNeu = [[Kas]],
              VollformNeu = Vollform
```

[34] bei denen auch die zusammengeschriebenen Formen des CISLEX zu unterdrücken sind

Bem. Diese Einträge werden in `<Beispieltext>.p.tmp.wrd` geschrieben, und nur benutzt, um zu kontrollieren, ob die ausführlicheren Einträge aus Abschnitt 6.2.4 für die jeweilige Präposition einen Eintrag hatten.

Die Nomina, die Eigennamen und die nominalisierten Adjektive (sowie die –fehlenden– nominalisierten Verben) erfordern Großschreibung:

⟨*Lexika/umformatierer.pl*⟩+≡
```
             ; (KlasseNeu = Klasse,
                   FormNeu = OrdForm,
                   (member(Kategorie,[n,na,vn,en,nn])
                   -> großschreibung(Vollform,VollformNeu)
                   ; VollformNeu = Vollform)))),
          Cat =.. [Kategorie,KlasseNeu,FormNeu],
```

Bei den Wörtern mit verschiedenen Vollformen wird ein Eintrag in `word/3` mit einer Stammform erzeugt, sonst ein Eintrag in `word/2` ohne Stammform:

⟨*Lexika/umformatierer.pl*⟩+≡
```
          (member(Kategorie, [v,n,na,nv,a])
          -> Eintrag =.. [word, VollformNeu, Cat, Stamm]
          ; Eintrag =.. [word, VollformNeu, Cat]).
```

Es folgen die Hilfsprogramme der Übersetzung der CISLEX-Codes:

⟨*Lexika/umformatierer.pl*⟩+≡
```
  zu_einfuegung(Praefix,Vollform,VollformNeu) :-
          atom_concat(Praefix,Rest,Vollform),
          ergaenzeKlammern(Praefix,Rest,zu,VollformNeu).

  großschreibung(Wort,WortNeu) :-
          name(Wort,[C|Chars]),
          (name(ä,[C]) -> name(Ä,[CNeu]) ; true),
          (name(ö,[C]) -> name(Ö,[CNeu]) ; true),
          (name(ü,[C]) -> name(Ü,[CNeu]) ; true),
          ((97 =< C, C =< 122) -> CNeu is C-32 ; true),
          !, name(WortNeu,[CNeu|Chars]).
  großschreibung(Wort,Wort).
```

Bei der Formatierung des Codes werden Merkmale auf Art- und Formparameter verteilt. Der Sonderfall der Suffixadjektive wird vorher abgefangen: sie werden wie Adjektive behandelt:

⟨*Lexika/umformatierer.pl*⟩+≡
```
  formatiereChars(asuff,Chars,Klasse,Form) :-
          !,formatiereChars(a,Chars,Klasse,Form).

  formatiereChars(Kategorie,[Char|Cs],Unterklasse,Form) :-
          formatiereChar(Char,M),
          formatiereChars(Kategorie,Cs,Us,Fs),
          (subkatMerkmal(Kategorie,M)
          -> Unterklasse = [M|Us], Form = Fs
          ; Unterklasse = Us, Form = [M|Fs]).
  formatiereChars(Kategorie,[],[],[]).

  subkatMerkmal(n,mask).
  subkatMerkmal(n,fem).
  subkatMerkmal(n,neut).
```

```
subkatMerkmal(na,mask).
subkatMerkmal(na,fem).
subkatMerkmal(na,neut).
subkatMerkmal(na,unsp).
subkatMerkmal(det,st).
subkatMerkmal(det,sw).
subkatMerkmal(det,ge).
```

Die relative Anordnung der Formmerkmale wird an den Gebrauch in früheren Parsern angepaßt (der Sonderfall der Suffixadjektive `asuff` ist nachträglich eingebaut; beim Infinitiv von Verben wird das Genus `unsp` weggelassen):

⟨*Lexika/umformatierer.pl*⟩+≡

```
ordneForm(n,[Kas,Num],[Num,Kas]) :- !.
ordneForm(v,[unsp,inf],[inf]) :- !.
% ordneForm(na,[Kas,Num],[Num,Kas]) :- !.
ordneForm(na,[Kas,Num,Adekl,Komp],[Komp,Adekl,Num,Kas]) :- !.
% ordneForm(na,[Komp,Adekl,Kas,Num],[Komp,Adekl,Num,Kas]) :- !.
ordneForm(a,[Praed,Komp],[Komp,Praed]) :- !.
ordneForm(a,[Kas,Num,Gen,Adekl,Komp],[Komp,Adekl,Gen,Num,Kas]) :- !.
ordneForm(det,[Kas,Num,Gen],[Gen,Num,Kas]).
ordneForm(pron,[Kas,Pers,Num,Gen],[Gen,Num,Kas,Pers]). % ??

ordneForm(asuff,X,Y) :-
        !, ordneForm(a,X,Y).
ordneForm(Cat,Form,[]) :-
        member(Cat,[adv,conj,part]),  % Fehler im CIS-Lex
        !.
% ordneForm(pr,_,_) :- fail.            % besser in Lexika/pp.wrd, von Hand gemacht
ordneForm(Cat,Form,Form).
```

Jeder Buchstabe der Formkodierung hat eine eigene Bedeutung, die in der lexikalischen und syntaktischen Anlayse wie folgt bezeichnet wird:

⟨*Lexika/umformatierer.pl*⟩+≡

```
formatiereChar(n,nom).
formatiereChar(a,akk).
formatiereChar(d,dat).
formatiereChar(g,gen).
formatiereChar('1',1).
formatiereChar('2',2).
formatiereChar('3',3).
formatiereChar('0',unsp).
formatiereChar(e,sg).
formatiereChar(m,pl).
formatiereChar('F',fem).
formatiereChar('M',mask).
formatiereChar('N',neut).
formatiereChar('U',unsp).
formatiereChar(x,st).  % stark
formatiereChar(y,sw).  % schwach
formatiereChar(z,ge).  % gemischt
formatiereChar(u,praedik). % undekliniert
formatiereChar(p,pos).
formatiereChar(k,komp).
```

```
formatiereChar(s,sup).
formatiereChar('G',praes).
formatiereChar('V',praet).
formatiereChar('O',unsp).
formatiereChar(i,ind).
formatiereChar(c,konj).
formatiereChar(b,imp).
formatiereChar('E',part).
formatiereChar('Z',part2).
formatiereChar('I',inf).
```

In die Artmerkmale werden keine Komplementrahmen eingesetzt; das erfolgt später mit Hilfe eines Stammlexikons, so daß man die Komplementrahmen nur dort (und daher nur einmal) angeben und *von Hand korrigieren* muß.

Bemerkung 9.7 Folgende Fehler der erzeugten Vollformeinträge sind bekannt:

(i) Fehlenden Klassenmerkmale, die ersten Argumente des Kategorienausdrucks, müssen von Hand nachgetragen werden! Aus den CISLEX-Einträgen wird nur das Genus bei Nomina abgelesen.

(ii) Die Formmerkmale im CISLEX (zumindest in der benutzten Version) sind FALSCH bei Präpositionen, Partikeln, Konjunktionen und abtrennbaren Verbpräfixen und Adverbien:

⟨*CISLEX-Einträge*⟩≡

```
an,an.PREP4#01:neN
ins,in.PDET#0:aeN:neN
überhaupt,überhaupt.ADV#00:aeNy
aber,aber.KONJ#0:1mGc:3mGc
```

In der neueren Version lauten die Einträge:

⟨*CISLEX-Einträge*⟩+≡

```
an,an.PREP4#0,:g
an,an.PREP4#01:g
an,an.PREP4#02:g
an,an.PREP4#0:g
ins,in.PDET#0:aeN:neN  % wieso Nominativ?
aber,aber.KONJ#00:g
aber,aber.KONJ#0:g
aber,aber.KONJ#0S:g
aber,aber.PART#00:g
aber,aber.PART#0:g
aber,aber.PART#0S:g
```

Diese Einträge sind ebenfalls auf Programmfehler zurückzuführen, weshalb für Präpositionen, Determinatoren und andere Wortklassen die in Abschnitt 6.2 angegebenen Einträge benutzt werden.

(iii) Die Merkmale sind nicht gründlich überlegt bei Partizipien, zu überprüfen bei `det` und `pron`.

(iv) Die Komparation muß bei Partikeln (nicht: Adj) ein Klassenmerkmal werden.

(v) Die CISLEX-Einträge bei Eigennamen enthalten kein Genus und *nur* die Stammform. Außerdem wird nicht genau zwischen Vor- und Nachnamen unterschieden:

⟨*CISLEX-Einträge*⟩+≡

```
darwin,darwin.EN(V)#0:g
darwins,darwins.EN(N)#0:g
jürgen,jürgen.EN(V)#0:g
```

Daher sind in `word/2` bei den Vor- und Nachnahmen sind die Klassen- und Formmerkmale von Hand zu korrigieren; statt dem von der `NP`-Grammatik erwarteten `word('Sigmund', vn([mask], [sg, nom]))` bekommt man nur `word('Sigmund', vn([], [gen]))`.

(vi) Bei der Anordnung der Formmerkmale ist der letzte Fall in `ordneForm` bloß ein Default.

(vii) Die Adjektiveinträge der Form `angetriebenst,angetrieben.ADJ0#0:AFs` werden nicht übersetzt, da `formatiereChar('A',X)` fehlt. Mit `AFs` wird im CISLEX kodiert, daß die entsprechende Form als Präfix zur Bildung zusammengesetzter Adjektive verwendbar ist, wie `best` in `bestmöglich` (P.Maier-Meyer, S.70).

Testen der Umformung von Einträgen des CISLEX

⟨*Lexika/umformatierer.pl*⟩+≡

```
formtest(Line) :-
        zerlege(Line,Vollform,Stamm,Praefix,Kategorie,Formen),
        nl,schreibeEintraege(user,Vollform,Stamm,Praefix,Kategorie,Formen).
```

Beispielsweise erhält man aus den Einträgen

⟨*CISLEX-Einträge*⟩+≡

```
überwältigungen,überwältigung.fem(NS0,NP3)#0:nmF:gmF:amF
unbewusste,unbewusste.NA#0:aeFxp:aeFyp:aeFzp:aeNyp:amUxp:neFxp:neFzp:neUyp:nmUxp
anschaut,anschauen.VSW1#2:2mGi:3eGi:mb
```

folgende Einträge in `word/3`, noch ohne Komplementrahmen:

⟨*CISLEX-Einträge umgerechnet*⟩≡

```
word(Überwältigungen, n([fem], [pl, nom]), überwältigung).
word(Überwältigungen, n([fem], [pl, gen]), überwältigung).
word(Überwältigungen, n([fem], [pl, akk]), überwältigung).

word('Unbewusste', na([fem], [pos, st, sg, akk]), unbewusste).
word('Unbewusste', na([fem], [pos, sw, sg, akk]), unbewusste).
word('Unbewusste', na([fem], [pos, ge, sg, akk]), unbewusste).
word('Unbewusste', na([neut], [pos, sw, sg, akk]), unbewusste).
word('Unbewusste', na([unsp], [pos, st, pl, akk]), unbewusste).
word('Unbewusste', na([fem], [pos, st, sg, nom]), unbewusste).
word('Unbewusste', na([fem], [pos, ge, sg, nom]), unbewusste).
word('Unbewusste', na([unsp], [pos, sw, sg, nom]), unbewusste).
word('Unbewusste', na([unsp], [pos, st, pl, nom]), unbewusste).

word(anschaut, v([], [2, pl, praes, ind]), anschauen).
word(schaut, v([an], [2, pl, praes, ind]), anschauen).
word(anschaut, v([], [3, sg, praes, ind]), anschauen).
word(schaut, v([an], [3, sg, praes, ind]), anschauen).
word(schaut, v([an], [pl, imp]), anschauen).
```

Der Stamm ist nützlich, um z.B. einer Stammformdatei die möglichen Komplemente zu entnehmen und in die Vollformen zu übertragen. (Vgl. Abschnitt 9.2.3.)

9.2.2 Komplementrahmen bei Vollverben

Noch nicht ins CISLEX integriert ist CISLEX-V, ein Verblexikon von Daniel Schnorbusch, das zu einfachen Vollverben des Deutschen die Komplementrahmen der Verben mit semantischen Einschränkungen an die Argumente enthält.

Wir benutzen dieses Verblexikon (in `erzeuge_staemme/0` in Abschnitt 9.2.3), um Komplementrahmen der einfachen Vollverben eines Beipieltexts in eine Stammdatei `<Beispieltext>.staemme.pl` zu übertragen. Für zusammengesetzte Verben oder Präfixverben muß man dort Komplementrahmen von Hand ergänzen.

Dazu wir zuerst jeder Eintrag des Verblexikons in die im Parser verwendete Notation umgerechnet und überflüssige Information beseitigt. Da das Verblexikon nicht frei verfügbar ist, kann *die hier beschriebene Umrechnung mit* `verbrahmen/0` *vom normalen Benutzer nicht verwendet werden.* Falls das Verblexikon nicht vorhanden ist, muß man in `<Beispieltext>.staemme.pl` *alle* Verbrahmen von Hand eintragen.

Die ursprünglichen Einträge im Verblexikon bestehen aus Zeilen der Form

⟨*CISLEX-V-Format*⟩≡
```
Infinitiv;<CPL:Komplementrahmen>;<sel:Semantische Einschränkungen>;
```

Die Komplementrahmen unterscheiden zwischen `X`, `Y`, `Z` und `W`-Argumenten und erlauben für jedes Argument verschiedene alternative syntaktische Formen. Die semantischen Einschränkungen beziehen sich auf diese Argumentnamen. Konkrete Einträge sind beispielsweise:

⟨*CISLEX-V-Einträge*⟩≡
```
ädern;<CPL:[Xnom_Yakk]>;<sel:X{hm}_Y{ub}>;
ändern;<CPL:[Xnom_Yakk_Z{Prp:rsp}]>;<sel:X{hm}_Y{th}_Z{ql}>;
ängstigen;<CPL:[Xnom_Yakk_Z{kor:mit$snt:d|w}|Z{Prp:mit}]>;<sel:X{hm}_Y{fn}_Z{sv}>;
```

Für den Parser ignorieren wir die semantischen Einschränkungen und rechnen nur die Komplementrahmen in die Form um, die der Parser erwartet. Es wird eine Hilfstabelle `verbrahmen/2` erstellt, die zur Bildung der Stammeinträge (vgl. Abschnitt 9.2.3) benutzt wird, auf die wiederum das Lexikon `word/3` des Parsers aufbaut.

⟨*Lexika/umformatierer.pl*⟩+≡
```
verbrahmen :-
        ensure_loaded('Tokenizer/tokenizer.pl'),
        write(user,'Eingabedatei (im Format des CISLEX-V): '),
        read(Eingabedatei),
        concat_atom([Eingabedatei,'.rahmen','.tmp'],Ausgabedatei),
        write(user,'Ausgabedatei: '),write(user,Ausgabedatei),write(user,'\n'),
        erzeuge_verbrahmen(Eingabedatei,Ausgabedatei).
```

Das Verblexikon eines Textes wird zeilenweise gelesen, mit `zerlege/3` in Bestandteile zerlegt und durch `schreibe_verbrahmen/2` nach der Umformung auf eine Ausgabe geschrieben:

⟨*Lexika/umformatierer.pl*⟩+≡
```
erzeuge_verbrahmen(Eingabedatei,Ausgabedatei) :-
        open(Eingabedatei,read,InStream),
        open(Ausgabedatei,write,OutStream),
        repeat,
        get0(InStream,Char),
        (dateiende(Char) -> true
        ; readline(InStream,[Char],Zeile),
```

```
            (zerlege(Zeile,Stamm,Rahmen)
            -> schreibe_verbrahmen(OutStream,Stamm,Rahmen)
            ; write(user,'Unzerlegbar: '),write(user,Zeile),write(user,'\n'),
                write(OutStream,'% '),write(OutStream,Zeile),write(OutStream,'\n')),
            fail),
        close(InStream),
        close(OutStream),
        write(user,'Datei mit übersetzten Verbrahmen erstellt.').
```

⟨*Lexika/umformatierer.pl*⟩+≡

```
zerlege(Zeile,Stamm,RahmenAlternativen) :-
        (concat_atom([Stamm,Komplementrahmen|_],';',Zeile)
        -> entferneKlammern('<CPL:[',']>',Komplementrahmen,Komplemente),
            concat_atom(Liste,'_',Komplemente),
            !,formatiereRahmen(Liste,Rahmen),
            alternativRahmen(Rahmen,RahmenAlternativen)
        ; fail).

entferneKlammern(Links,Rechts,Atom,Gekuerzt) :-
        concat(Links,Suffix,Atom),
        concat(Gekuerzt,Rechts,Suffix).
ergaenzeKlammern(Links,Rechts,Atom,Geklammert) :-
        concat(Links,Atom,Praefix),
        concat(Praefix,Rechts,Geklammert).

formatiereRahmen([Komplement|Rest],[Alternativen|RestFmt]) :-
        zerlege_in_Alternativen(Komplement,Alternativen),
        formatiereRahmen(Rest,RestFmt).
formatiereRahmen([],[]).
```

Im Verblexikon wird bei Objekten, die durch Reflexivpronomen realisiert werden müssen, nicht zwischen Akkusativ- und Dativobjekten unterschieden. Wir übersetzen die Angabe stets in ein akkusativisches Reflexivobjekt; falsche Einträge[35] `akk(refl)` müssen von Hand zu `dat(refl)` korrigiert werden:

⟨*Lexika/umformatierer.pl*⟩+≡

```
zerlege_in_Alternativen('Refl',[akk(refl)]) :- !.
zerlege_in_Alternativen('Adv',[adv]) :- !.
zerlege_in_Alternativen('E',[es]) :- !.

zerlege_in_Alternativen(Atom,Liste) :-
        member(Char,['X','Y','Z','W']),
        concat(Char,Rest,Atom),
        concat_atom(Liste1,Char,Rest),
        entferneMids(Liste1,Liste2),
        vereinfacheAlternativen(Liste2,Liste).

entferneMids([Atom|Atome],[AtomE|AtomeE]) :-
        (concat(AtomE,'|',Atom) -> true ; AtomE = Atom),
        entferneMids(Atome,AtomeE).
entferneMids([],[]).
```

[35] `Mach dich[akk] auf die Socken, aber mach dir[dat] nicht in die Hose!`

Falls das Verblexikon für ein Argument die Alternativen erlaubt, es als Korrelat eines Satzes *und* als Präpositionalobjekt darzustellen, ignorieren wir die erste Möglichkeit[36]:

⟨*Lexika/umformatierer.pl*⟩+≡
```
vereinfacheAlternativen([A,B|Cs],Es) :-
        atom_prefix(A,'{kor:'),
        atom_prefix(B,'{Prp:'),
        !, vereinfacheAlternativen([B|Cs],Es).
```

Falls das Verblexikon ein Präpositionalobjekt oder einen Objektsatz erlaubt, zerlegen wir die Alternative möglicher Präpositionen oder Satzformen:

⟨*Lexika/umformatierer.pl*⟩+≡
```
vereinfacheAlternativen([B|Cs],Es) :-
        atom_prefix(B,'{Prp:'),
        !, zerlegePraepositionen(B,Bs),
        vereinfacheAlternativen(Cs,Ds),
        append(Bs,Ds,Es).
vereinfacheAlternativen([B|Cs],Es) :-
        atom_prefix(B,'{snt:'),
        !, zerlegeSatzkompl(B,Bs),
        vereinfacheAlternativen(Cs,Ds),
        append(Bs,Ds,Es).
```

Sonst stellen wir das Objekt im Format des Parsers dar und behandeln die restlichen Objekte:

⟨*Lexika/umformatierer.pl*⟩+≡
```
vereinfacheAlternativen([A|Bs],[AFmt|Cs]) :-
        formatiereKompl(A,AFmt),
        vereinfacheAlternativen(Bs,Cs).
vereinfacheAlternativen([],[]).
```

Wo das Verblexikon alternative Präpositionen zum Anschluß des Präpositionalobjekts erlaubt, stellen wir das Komplement durch die Liste der Präpositionen dar:

⟨*Lexika/umformatierer.pl*⟩+≡
```
zerlegePraepositionen(Atom,Praeps) :-
        entferneKlammern('{Prp:','}',Atom,Kern),
        concat_atom(Praeps,'|',Kern).
```

Wo das Verblexikon Objekt- oder Subjektsätze erlaubt, vereinfachen wir die Alternativen zu den Fällen des definiten und des interrogativen Komplementsatzes, die durch `daß` und `qu` repräsentiert werden:[37]

⟨*Lexika/umformatierer.pl*⟩+≡
```
zerlegeSatzkompl(Atom,Kompls) :-
        entferneKlammern('{snt:','}',Atom,Kern),
        concat_atom(Komplemente,'|',Kern),
        ((member(d,Komplemente) , Kompls = [daß]) ;
         (Komplemente = [w,i] , Kompls = [qu]) ;
            (Komplemente = [i] , Kompls = [qu])).
```

[36] Anscheinend ist an Verben wie `glauben an[akk]` und `glauben (daran, daß S)` gedacht, wo man eher das Präpositionalobjekt semantisch einschränken müßte: es sind nur Nominalobjekte wie `den Niedergang des Kapitalismus` oder `den lieben Gott` erlaubt, die Nominalisierungen oder elliptische Formen von Aussagen sind.

[37] Mir scheint die feinere Unterscheidung des Verblexikons redundant zu sein; jedenfalls nutzt der Parser diese Unterschiede nicht aus.

Das Format der verbliebenen Komplementdarstellung wird nun in das des Parsers übersetzt. Definite Komplementsätze werden immer durch `daß`, interrogative durch `qu` und infinite durch `infzu` abgekürzt; dabei ignorieren wir, ob der Komplementsatz laut Verblexikon mit einem oder durch ein Korrelat formuliert werden kann:

⟨*Lexika/umformatierer.pl*⟩+≡

```
formatiereKompl(X,daß) :-
        (X = '{snt:d}'
        ; atom_prefix(X,'{kor:e$snt:d')
        ; atom_prefix(X,'{kor:*$snt:d')
        ; atom_prefix(X,'{kor:*|e$snt:d')
        ), !.

formatiereKompl(X,K) :-
        X = '{kor:*|e$snt:o|w|i}',
        !, (K = qu ; K = infzu).

formatiereKompl(X,qu) :-
        (X = '{kor:e$snt:w}'
        ; X = '{kor:*$snt:o|w}'
        ; X = '{kor:*|e$snt:o|w}'
        ), !.

formatiereKompl(X,infzu) :-
        (X = '{kor:*$snt:i}'
        ; X = '{kor:e$snt:i}'
        ; X = '{kor:*|e$snt:i}'
        ), !.
```

Gelegentlich, insbesondere bei den Modalverben, muß man die `infzu`-Einträge zu `inf`-Einträgen korrigieren, da das Verblexikon keinen Unterschied zwischen infiniten Komplementsätzen macht. Die übrigen Fälle müssen von Hand nachbearbeitet werden:

⟨*Lexika/umformatierer.pl*⟩+≡

```
formatiereKompl(X,X).          % Noch von Hand behandeln!

schreibe_verbrahmen(Strom,Stamm,[R|Rahmen]) :-
        Eintrag =.. [verbrahmen,Stamm,R],
        writeq(Strom,Eintrag),
        write(Strom,'.\n'),
        schreibe_verbrahmen(Strom,Stamm,Rahmen).
schreibe_verbrahmen(Strom,Stamm,[]).
```

Im `Rahmen` des Verblexikons sind in jedem Argument mehrere Alternativen erlaubt; durch Ausmultiplizieren machen wir daraus eine Liste von alternativen Rahmen mit eindeutigen Argumenten.

Falls nur ein Argument vorhanden ist, werden aus seinen Alternativen die Rahmen mit je einer dieser Alternativen gebildet:

⟨*Lexika/umformatierer.pl*⟩+≡

```
alternativRahmen([Arg1],Altn) :-
        singletons(Arg1,Altn).
singletons([A|As],[[A]|Bs]) :-
        singletons(As,Bs).
singletons([],[]).
```

Falls noch mehr als ein Argument erlaubt ist, wird zuerst aus den restlichen Argumenten eine Liste von eindeutigen Rahmen gebildet und dann die einzelnen Möglichkeiten für das erste Argument zu jedem dieser Rahmen hinzugefügt:

⟨*Lexika/umformatierer.pl*⟩+≡
```
alternativRahmen([Arg1|Args],Altn) :-
        alternativRahmen(Args,ArgsAltn),
        addArg1(Arg1,ArgsAltn,Altn).

addArg1([A|As],Altn,Bs) :-
        addArg1(As,Altn,AsAltn),
        addArg(A,Altn,AAltn),
        append(AAltn,AsAltn,Bs).
addArg1([],_,[]).

addArg(A,[B],[[A|B]]) :- !.
addArg(A,[B|Bs],[[A|B]|Cs]) :-
        addArg(A,Bs,Cs).
addArg(A,[],[A]).
```

Testen der Umformung von Einträgen des Verblexikons

Nur zum Testen dient:

⟨*Lexika/umformatierer.pl*⟩+≡
```
formtest_ds(Zeile) :-
        zerlege(Zeile,Stamm,Rahmen),
        nl,schreibe_verbrahmen(user,Stamm,Rahmen).
```

Beispielsweise erhält man aus dem ursprünglichen Einträgen

⟨*CISLEX-V-Einträge*⟩+≡
```
ärgern;<CPL:[Xnom_Refl_Y{kor:*|über$snt:d|w|i}|Y{Prp:über}]>;<sel:X{hm}_Y{th|sv}>;
ärgern;<CPL:[Xnom_Yakk_Z{kor:mit$snt:d|i}|Z{Prp:durch|mit}]>;<sel:X{hm}_Y{fn}_Z{sv}>;
ärgern;<CPL:[X{snt:d|w|i}|Xnom_Yakk]>;<sel:X{sv}_Y{hm}>;
```

mit `formtest_ds/1` die Einträge:

⟨*Verblex-Einträge umgerechnet*⟩≡
```
verbrahmen(ärgern, [nom, akk(refl), über]).
verbrahmen(ärgern, [nom, akk, durch]).
verbrahmen(ärgern, [nom, akk, mit]).
verbrahmen(ärgern, [daß, akk]).
verbrahmen(ärgern, [nom, akk]).
```

Es gibt keine Unterscheidung zwischen optionalen und obligatorischen Komplementen, außer durch verschiedene Komplementrahmen.

Bem.: Gewisse Abkürzungen der Komplemente werden aus dem Verblexikon übernommen, obwohl der Parser damit noch nichts anfangen kann:

(i) die durch `#nom` angegebenen Objektnominalphrasen, die im Plural stehen müssen,

(ii) die durch Abkürzungen von Präpositionengruppen angegebenen Objektpräpositionalphrasen, wobei `loc` für lokale (wie `in,an,auf` mit Dativrektion), `dir` für direktionale, `kon` für kontaktive (wie `an`, `auf`, `gegen`), `tmp` für temporale, `pol` für polare (wie `für`, `gegen`), `istr` für instrumentale Präpositionen steht.

9.2.3 Erzeugung von Stammlexika

Damit man auch andere Texte, die vom CISLEX lemmatisiert wurden, mit dem Parser bearbeiten kann, ist es zweckmäßig, wenn alle im CISLEX fehlenden (und möglichst auch alle falschen) Informationen der Lexikoneinträge auf einfache Weise ergänzt werden können.

Im CISLEX fehlt insbesondere die Information über die möglichen Komplemente eines Verbs, Adjektivs oder Nomens. Andere syntaktische Information, z.B. ob ein Verb ein betontes Präfix hat, ist zwar kodiert, aber es werden nicht alle Konsequenzen daraus gezogen: z.B. enthält das Lexikon weder die finiten Formen, bei denen das Präfix abgetrennt wird (für die Verbzweitstellung), noch die `infzu`-Form, bei der das `zu` nach dem Präfix eingefügt wird.

Damit man nicht alle Zusatzinformation in jeder Vollform nachtragen muß, wird eine eigene Datei `<Beispieltext>.stamm.tmp` angelegt, in der zu jeder Stammform die benötigte Zusatzinformation vermerkt wird. Wir machen das erstmal nur für die Komplementrahmen (und ergänzen vielleicht später Unterklassenmerkmale):

⟨*Lexika/umformatierer.pl*⟩+≡
```
:- dynamic cis_stamm/2.

erzeuge_default_staemme :-
        ensure_loaded('Tokenizer/tokenizer.pl'),
        (access_file('../Lexika/cis-verben.schnorbusch.rahmen.pl',read) ->
            consult('../Lexika/cis-verben.schnorbusch.rahmen.pl')
        ; true),
        write(user,'Eingabedatei (im CISLEX-Format): '),
        read(Eingabedatei),
        concat_atom([Eingabedatei,'.stamm.tmp'],Ausgabedatei),
        concat_atom([Eingabedatei,'.stamm.pl'],Editieren),
        (access_file(Editieren,read)
        -> write(user,'Korrigierte Stammdatei: '),
            write(user,Editieren),
            write(user,' existiert.\n Trotzdem erzeugen? (y.|n.):\n '),
            read(user,Antwort),
            (Antwort = n
            -> true
            ; erzeuge_stammeintraege(Eingabedatei,Ausgabedatei),
                write(user,'\nAusgabedatei: '),write(user,Ausgabedatei))
        ; erzeuge_stammeintraege(Eingabedatei,Ausgabedatei),
            write(user,'\nAusgabedatei: '),write(user,Ausgabedatei)).

erzeuge_stammeintraege(Eingabedatei,Ausgabedatei) :-
        open(Eingabedatei,read,InStream),
        open(Ausgabedatei,write,OutStream),
        default_stamm(InStream,OutStream),
        close(InStream),
        close(OutStream),
        nl,write(user,'Default-Stammeinträge erzeugt.').

default_stamm(InStream,OutStream) :-
        retractall(cis_stamm(_,_)),
        !, repeat,
        get0(InStream,Char),
        (Char = -1 % dateiende(Char)
        -> flush_output(OutStream), true
```

```
        ; readline(InStream,[Char],Line),
          zerlege(Line,_Vollform,Stamm,_Unterklasse,Kategorie,_Formen),
          (cis_stamm(Stamm,Kategorie)
          -> true
          ;   schreibe_stamm(OutStream,Stamm,Kategorie),
              assert(cis_stamm(Stamm,Kategorie))),
          fail).
```

Das Format der Stammeinträge kann sich ändern, wenn weitere Information in den Stammeintrag aufgenommen werden muß. In `verbrahmen/2` sind die Komplementrahmen des nach Abschnitt 9.2.2 umformatierten Verblexikons aufgeführt.

⟨*Lexika/umformatierer.pl*⟩+≡

```
schreibe_stamm(Strom,Stamm,v) :-
        verbrahmen(Stamm,_),
        !, verbrahmen(Stamm,Rahmen),
        Eintrag = stamm(v,Stamm,Rahmen),
        writeq(Strom,Eintrag),write(Strom,'.\n').
schreibe_stamm(Strom,Stamm,v) :-
        !, Eintrag = stamm(v,Stamm,[nom,akk]),
        write(user,'\nOhne Komplementrahmen: '),
        write(user,Stamm),
        writeq(Strom,Eintrag),write(Strom,'. '),
        writeq(Strom,'% Default!'),
        write(Strom,'\n').

schreibe_stamm(Strom,Stamm,Kat) :-
        ((Kat = n ; Kat = a ; Kat = na ; Kat = nv)
        -> Eintrag = stamm(Kat,Stamm,[]),
           writeq(Strom,Eintrag),write(Strom,'.\n')
        ; true).
```

Zugriffsprädikate auf das Stammlexikon

Zum Parsen mit einer größeren Grammatik braucht man ein größeres Vollformenlexikon. Dafür ist es sinnvoll, syntaktische Information zu den Wörtern bei ihren Stammformen anzugeben und nicht bei jeder Vollform zu wiederholen, sodaß die Korrektur dieser Information einfach ist.

Die zusätzliche syntaktische Information, z.B. die Angabe der Komplementrahmen, wird in einem getrennten Stammlexikon `stamm(Wortkategorie, Stamm, Information)` formuliert.

Um den Parser in einfacher Weise an Änderungen der syntaktischen Information der Stammeinträge anpassen zu können, wird auf dieses Stammlexikon nur mit speziellen Pädikaten wie

⟨*Lexika/word.pl*⟩≡

```
komplementrahmen(Kat,Stamm,Rahmen) :-
        stamm(Kat,Stamm,Rahmen).
```

zugegriffen, so daß man gegebenenfalls nur diese Prädikate anpassen muß. Das Format der Stammeinträge enthält vorläufig nur die Komplementrahmen; sobald es geändert wird, müssen alle Zugriffsprädikate angepaßt werden.

Das vom Parser erwartete Vollformenlexikon `word(Vollform,Kategorie)` konstruiert man, indem man das umformatierte CIS-Lexikon `word(Vollform,Pseudokategorie,Stamm)` mit der Information des Stammlexikons kombiniert.

Die aus dem CISLEX abgeleitete Pseudokategorie wird zur Zeit nur um einen Komplementrahmen erweitert, später aber vielleicht auch um andere Unterklassenparameter:

⟨*Lexika/word.pl*⟩+≡

```
word(W,Cat) :-
        word(W,CatCIS,Stamm),
        CatCIS =.. [Kat,Art,Form],
        (komplementrahmen(Kat,Stamm,_Rahmen)
        -> komplementrahmen(Kat,Stamm,Rahmen),
            ((Kat = v, Cat =.. [Kat,[Rahmen|Art],Form])
            ; (Kat = a, Cat =.. [Kat,Rahmen,Form])
            ; ((Kat = n; Kat = na),
                  (Rahmen = []
                  -> ArtNeu = Art ; append(Art,[Rahmen],ArtNeu)),
                  Cat =.. [Kat,ArtNeu,Form]))
        ; Cat = CatCIS).
```

Bem. Bei den Adjektiven besteht (nach der Grammatik `ap.grm`) die Art aus dem Komplementrahmen; es sollte aber der Komplementrahmen nur eines von mehreren Artmerkmalen sein, d.h. es fehlt eine Klammerebene.

Damit zur Parsezeit das Vollformenlexion `word/2`, das auf `word/3` zurückgeführt wird, benutzt werden kann, muß auch `<Beispieltext>.stamm.pl` geladen sein.

9.3 Flexionsprogramm für Wörter

Mit dem Stamm (ohne Komplemente und Klassenparameter) können wir aus dem Vollformenlexikon, das natürlich aus einem Stammformenlexikon durch ein Flexionsprogramm erzeugt wurde, ein Flexionsprogramm zurückgewinnen:

⟨*Lexika/word.pl*⟩+≡

```
flektiere(Vollform1, Kategorie1, Form2, Vollform2) :-
        word(Vollform1, Kategorie1, Stamm),
        Kategorie1 =.. [Kat,Art,Form1],
        Kategorie2 =.. [Kat,Art,Form2],
        word(Vollform2, Kategorie2, Stamm).
```

Das wird später zur Transformation von analysierten Phrasen benutzt, um die darin vorkommenden Wörter umzuwandeln.

10 Anhang 2: Lösungen der Aufgaben

Lösung zu Aufgabe 3.1:

Wenn Kommentare (mit `%`) beginnen, ändert man das Lesen, indem man `skipComment(Strom+,Kommentar-)` aufruft:

⟨*Ignoriere Kommentare*⟩≡

```
read_sentence(Stream,Sentence,Char,Seen) :-
        name('%',[Char]),
        !,skipComment(Stream,[Char,10]),        % 10=Zeilenende
        read_sentence(Stream,Sentence,Seen).
```

Die Zeichen des Stroms werden gesammelt und später am Bildschirm ausgegeben (beides kann man auch sein lassen):

⟨*skipComment*⟩≡

```
skipComment(Stream,CommentSeen) :-
        get0(Stream,Char),
        skipComment(Stream,Char,CommentSeen).

skipComment(_Stream,Char,CommentSeen) :-
        zeilenende(Char),
        !,reverse([Char|CommentSeen],Comment),
        name(Atom,Comment),
        write(user,Atom).

skipComment(Stream,Char,CommentSeen) :-
        skipComment(Stream,[Char|CommentSeen]).
```

Man muß nur noch `<Ignoriere Kommentare>` am Anfang von `read_sentence/4` einfügen und `<skipComment>` laden. △

Lösung zu Aufgabe 3.2: Weil Dateizugriffe teuer sind, werden beim Schreiben auf einen Strom die zu schreibenden Zeichen in einem Puffer einer bestimmten Größe, z.B. einer Zeilenlänge oder einer Eingabedatei, zwischengespeichert; alle im Puffer gesammelten Zeichen werden dann in einem einzigen Schritt auf den Strom geschrieben. Mit `flush_output(Strom)` erzwingt man, daß der Puffer auf den `Strom` geleert wird.

Um den Text einer Datei in einzelne Sätze zu trennen, öffnet man die Eingabedatei zum Lesen und die Ausgabedatei zum Schreiben:

⟨*Tokenizer/satztrenner.pl*⟩≡

```
satztrennen(Eingabedatei,Ausgabedatei) :-
        open(Eingabedatei,read,Eingabestrom),
        open(Ausgabedatei,write,Ausgabestrom),
        einzelsaetze(Eingabestrom,Ausgabestrom),
        close(Ausgabestrom),
        close(Eingabestrom).
```

⟨*einzelsaetze*⟩

Nun liest man wiederholt vom Eingabestrom einen Satz und schreibt diesen sowie anschließend mit `nl(Ausgabestrom)` zwei zusätzliche Zeilenumbrüche auf den Ausgabestrom. Falls das Ende des Eingabestroms erreicht ist, leert man den Puffer der zu schreibenden Zeichen mit `flush_out(Ausgabestrom)`:

⟨*einzelsaetze*⟩≡

```
einzelsaetze(Eingabestrom,Ausgabestrom) :-
        get0(Eingabestrom,Char),
        (dateiende(Char)
        -> flush_output(Ausgabestrom)
        ; (read_sentence(Eingabestrom,Satz,Char,[]),
              entferneUmbruch(Satz,[],Zeile),
              name(Atom,Zeile),
              write(Ausgabestrom,Atom),
              nl(Ausgabestrom), nl(Ausgabestrom),
              einzelsaetze(Eingabestrom,Ausgabestrom))
        ).
```

Ohne das Leeren des Puffers würde in der Ausgabedatei nichts erscheinen; sie wird also erst beschrieben, wenn man alle Sätze der Eingabedatei gelesen hat!

Zeilenumbrüche und doppelte Leerzeichen innerhalb eines Satzes werden wie folgt beseitigt (vgl. Tokenisierung):

(i) Leerzeichen und Umbruch nach einer Worttrennung werden entfernt:

⟨*einzelsaetze*⟩+≡

```
entferneUmbruch([T,N,L|String],Cs,Zeile) :-
        worttrennung([T,N]),(leerZeichen(L);zeilenende(L)),
        !,entferneUmbruch([T,N|String],Cs,Zeile).
```

(ii) Leerzeichen nach einem Leerzeichen, einem Umbruch, oder einem Trennzeichen werden entfert:

⟨*einzelsaetze*⟩+≡

```
entferneUmbruch([C,L|String],Cs,Zeile) :-
        (leerZeichen(C);zeilenende(C);trennZeichen(C)),
        leerZeichen(L),
        !,entferneUmbruch([L|String],Cs,Zeile).
```

(iii) Leerzeichen vor einem Umbruch werden entfernt:

⟨*einzelsaetze*⟩+≡

```
entferneUmbruch([L,N|String],Cs,Zeile) :-
        leerZeichen(L),zeilenende(N),
        !,entferneUmbruch([N|String],Cs,Zeile).
```

(iv) Worttrennungen werden beseitigt (ohne daß Leerstellen entstehen):

⟨*einzelsaetze*⟩+≡

```
entferneUmbruch([T,N|String],Cs,Zeile) :-
        worttrennung([T,N]),
        !,entferneUmbruch(String,Cs,Zeile).
```

(v) Doppelte Umbrüche werden verschmolzen:

⟨*einzelsaetze*⟩+≡

```
entferneUmbruch([N,N|String],Cs,Zeile) :-
        zeilenende(N),
        !,entferneUmbruch([N|String],Cs,Zeile).
```

(vi) Verbliebene Umbrüche werden durch ein Leerzeichen ersetzt:

⟨*einzelsaetze*⟩+≡
```
entferneUmbruch([N|String],Cs,Zeile) :-
        zeilenende(N),
        !,leerZeichen(L),entferneUmbruch(String,[L|Cs],Zeile).
```

(vii) Andere Zeichen werden überlesen:

⟨*einzelsaetze*⟩+≡
```
entferneUmbruch([C|String],Cs,Zeile) :-
        entferneUmbruch(String,[C|Cs],Zeile).
```

(viii) Schließlich wird die invers akkumulierte Zeichenfolge umgekehrt:

⟨*einzelsaetze*⟩+≡
```
entferneUmbruch([],Cs,Zeile) :-
        reverse(Cs,Zeile).
```

Paragraphenanfänge des Eingangsstrings werden durch ein Leerzeichen am Zeilenanfang gekennzeichnet.

Bemerkung: Es gibt eine Option (vgl. `open/4` von SWI-Prolog), bei der das Schreiben ohne Zwischenpuffer erfolgt. Das sollte man nur verwenden, wenn man z.B. auf den Bildschirm schreibt, wo der Leser auf die Ausgabezeichen wartet.

Hier ist eine Variante, mit der man die Sätze gleichzeitig numerieren kann:

⟨*einzelsaetze*⟩+≡
```
satznumerieren(Eingabedatei,Ausgabedatei) :-
        open(Eingabedatei,read,Eingabestrom),
        open(Ausgabedatei,write,Ausgabestrom),
        einzelsaetze_numeriert(Eingabestrom,Ausgabestrom,0),
        close(Ausgabestrom),
        close(Eingabestrom).

einzelsaetze_numeriert(Eingabestrom,Ausgabestrom,N) :-
        get0(Eingabestrom,Char),
        (dateiende(Char)
        -> flush_output(Ausgabestrom)
        ; (read_sentence(Eingabestrom,Satz,Char,[]),
              entferneUmbruch(Satz,[],Zeile),
              write(Ausgabestrom,N), write(Ausgabestrom,') '),
              name(Atom,Zeile), write(Ausgabestrom,Atom),
              nl(Ausgabestrom), nl(Ausgabestrom),
              M is N + 1,
              einzelsaetze_numeriert(Eingabestrom,Ausgabestrom,M))
        ).
```

△

Lösung zu Aufgabe 5.1: Das ist inzwischen im Abschnitt 5.3.5 ausgeführt. △

Lösung zu Aufgabe 5.2: Wir schreiben die Bäume an den Bildschirm, aber gleichzeitig auch in eine Datei `baum.tmp.txt`, damit man sich die Analyse aufheben kann:

⟨*Parser/PrintTree/showtree.pl*⟩≡

```
showASCIItrees(Trees) :-
   showAsciiTrees(Trees),
   tell('baum.tmp.txt'),
   showAsciiTrees(Trees),
   told.
```

Mit `showAsciiTrees(Baumfolge+,Einrückung+,Nummer+)` werden die Bäume der `Baumfolge`, beginnend bei `Nummer` und mit `Einrückung` vielen Leerzeichen am Zeilenanfang, untereinander geschrieben. Wir beginnen mit Nummer 1 und Einrückungstiefe 1:

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
showAsciiTrees(Trees) :-
        showAsciiTrees(Trees,1,1).
showAsciiTrees([Tree|Trees],Indent,N) :-
        nl,(N > 0 -> write(N),write('.'); tab(2)),
        showAsciiTree(Tree,Indent),
        (N > 0 -> M is N+1; M=N),
        showAsciiTrees(Trees,Indent,M).
showAsciiTrees([],_Indent,_N).
```

Mit `showAsciiTree(Baum+,Einrückung+)` wird ein einzelner `Baum`, mit `Einrückung` vielen Leerzeichen am Zeilenanfang, ausgegeben. Teilbäume werden 3 Leerzeichen weiter eingerückt:

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
showAsciiTree([Parent|Children],Indent) :-
        tab(Indent),write(Parent),
        IndentC is Indent + 3,
        showAsciiTrees(Children,IndentC,0).
showAsciiTree(Leaf,Indent) :-
        tab(Indent),write(Leaf).
```

△

Es ist etwas komplizierter, die Bäume mit LaTeX/TreeTeX darzustellen. (Man muß TreeTeX allerdings so ändern, daß es nicht auf binär verzweigende Bäume beschränkt bleibt, was sehr mühsam ist!).

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
showTeXtrees(Trees) :-
        nl,nl,
        showtrees1(Trees,'baum.tmp.tex'),
        shell('latex baum.tmp.tex'),
        shell('xdvi baum.tmp.dvi -expert -s 4 -geometry 780x740+300+5 -topmargin 10mm'),
        % shell('xdvi baum.tmp.dvi -expert -s 6 -geometry 980x900+300+0 -topmargin 15mm'),
        shell('rm baum.tmp.aux'),
        shell('rm baum.tmp.log').

showtrees([W1|Ws],Filename) :-
   trees([W1|Ws],Trees),
   showtrees1(Trees,Filename).
showtrees1(Trees,Filename) :-
   tell(Filename),
   write('\\documentclass{article}'),nl,
   write('\\usepackage[german]{babel}'),nl,
```

```
    write('\\usepackage[T1]{fontenc}'),nl,
    write('\\usepackage[isolatin]{inputenc}'),nl,
    % write(`\\usepackage[latin1]{inputenc}'),nl, % besser?
    write('\\topmargin0pt \\oddsidemargin0pt \\textheight23cm \\textwidth16cm'),nl,
    write('\\begin{document} \\input{TreeTeX}\\input{arity}'),nl,
    write('\\def\\Node#1#2{\\node{\\type{dot}\\arity{#2}\\rght{#1}}}'),nl,
    teXtrees(Trees),
    write('\\end{document}'),
    told.

 teXtrees([Tree|Trees]) :-
    teXtree(Tree),
    teXtrees(Trees).
 teXtrees([]).

 teXtree(Baum) :-
    nl,write('\\begin{Tree}'),nl,
    treedraw(Baum),
    write('\\end{Tree}'),nl,
    write('\\begin{center} \\usebox{\\TeXTree} \\end{center}'),nl,
    nl,write('\\vskip1cm').

 treedraw([Mother|Children]) :-
    treesdraw(Children),
    length(Children,L),
    drawnode(Mother, L).
 treedraw(Child) :-
    (atom(Child);integer(Child)), % integer: für Kardinalzahlen
    drawnode(Child,0).

 treesdraw([Child]) :-
    treedraw(Child).
 treesdraw([Brother | Sisters]) :-
    treedraw(Brother),
    treesdraw(Sisters).
 treesdraw([]).
```

Beim Darstellen der Knotenmarkierungen muß man die Umlaute richtig in die LaTeX-Datei der Baumdarstellung schreiben. Das wird mit `writeTexUmlaute/1` gemacht und ist etwas mühsam (vielleicht geht es einfacher!). Umlaute treten bei den Blättern und bei Merkmalen auf.

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
 drawnode(Node,L) :-
    (atom(Node);integer(Node)),
    write('\\Node{'),writeTexUmlaute(Node),write('}{'),write(L),write('}'),nl.
```

Kategorien werden hier normalerweise durch zweistellige Prolog-Terme `Kategorie(Unterklasse,Form)` angegeben, wobei `Unterklasse` und `Form` jeweils eine *Liste* von Merkmalwerten ist. Bei der Darstellung der Kategorien werden Klassenmerkmale als untere, Formmerkmale als obere Indizes an den Kategoriennamen notiert. Wir unterscheiden drei Fälle:

(i) Kategorien, die durch zweistellige Prolog-Terme `Kategorie(Unterklasse,Form)` angegeben werden:

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
drawnode(Node,L) :-
   Node =.. [Kat,Klasse,Form],
   write('\\Node{${\\rm '),
   write(Kat),write('}^{\\rm '),writeTexMatUmlaute(Form),write('}'),
               write('_{\\rm '),writeTexMatUmlaute(Klasse),write('}$'),
   write('}{'),write(L),write('}'),nl.
```

(ii) Kategorien, die durch einstellige Prolog-Terme `Kategorie(Unterklasse)` angegeben werden, deren Argument die Unterklassenmerkmale angibt:

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
drawnode(Node,L) :-
   Node =.. [Kat,Klasse],
   member(Kat,[comp,mod,p,subj,grad]),
   !,    % anfällig gegen Grammatikänderungen!
   write('\\Node{${\\rm '),
   write(Kat),write('}_{\\rm '),writeTexMatUmlaute(Klasse),write('}$'),
   write('}{'),write(L),write('}'),nl.
```

(iii) Kategorien, die durch einstellige Prolog-Terme `Kategorie(Form)` angegeben werden, deren Argument die Formmerkmale angibt:

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
drawnode(Node,L) :-
   Node =.. [Kat,Form],
   write('\\Node{${\\rm '),
   write(Kat),write('}^{\\rm '),writeTexMatUmlaute(Form),write('}$'),
   write('}{'),write(L),write('}'),nl.
```

Die Umlaute müssen (??) leider anders dargestellt werden als man erwartet (vielleicht ist das mit der LaTeX-Option der Eingabekodierung zu vereinfachen):

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
writeTexUmlaute(Atom) :-       % 'Bär' |-> 'B\"ar'
        texUmlaute(Atom,UW),
        write(UW).

texUmlaute(W,UW) :-
        concat(ä,R,W),!,
        texUmlaute(R,UR),
        concat('\\"a',UR,UW).
texUmlaute(W,UW) :-
        concat(ö,R,W),!,
        texUmlaute(R,UR),
        concat('\\"o',UR,UW).
texUmlaute(W,UW) :-
        concat(ü,R,W),!,
        texUmlaute(R,UR),
        concat('\\"u',UR,UW).
texUmlaute(W,UW) :-
        concat(Ä,R,W),!,
        texUmlaute(R,UR),
        concat('\\"A',UR,UW).
texUmlaute(W,UW) :-
        concat(Ö,R,W),!,
```

```
        texUmlaute(R,UR),
        concat('\\"O',UR,UW).
texUmlaute(W,UW) :-
        concat(Ü,R,W),!,
        texUmlaute(R,UR),
        concat('\\"U',UR,UW).
texUmlaute(W,UW) :-
        concat(ß,R,W),!,
        texUmlaute(R,UR),
        concat('{\\ss}',UR,UW).
texUmlaute('','') :- !.
texUmlaute(W,UW) :-
        name(W,[H|L]),
        name(AL,L),texUmlaute(AL,UL),
        name(C,[H]), concat(C,UL,UW).
```

Im Mathematik-Modus von LaTeX, der für die Hoch- und Tiefstellung der Indizes benutzt wird, braucht man (leider) wir eine andere Angabe der Umlaute. Die Form- und Klassenmerkmale werden im Kategorienausdruck zwar als Listen angegeben, aber im Baum schreiben wir die Außenklammern der Listen nicht:

⟨*Parser/PrintTree/showtree.pl*⟩+≡

```
writeTexMatUmlaute([W|Ws]) :-
        !, ersetzeUmlaute([W|Ws],NWs),
        writelist(NWs).

writeTexMatUmlaute(Ws) :-       % Ws ist Term
        atom(Ws),!,
        texMatUmlaute(Ws,UWs),   % z.B. bei s(daß,_)
        write(UWs).
writeTexMatUmlaute(Ws) :-        % Ws ist Term, z.B. refl(3,sg)
        Ws =.. [F|Args],
        ersetzeUmlaute([F|Args],[UF|UArgs]),
        UWs =.. [UF|UArgs],
        write(UWs).
writeTexMatUmlaute([]) :-        % wird vermutlich nie benutzt.
        !, write([]).

writelist([W1,W2|Ws]) :-
        !,write(W1),write(','),writelist([W2|Ws]).
writelist([W|Ws]) :-
        write(W), writelist(Ws).
writelist([]).

ersetzeUmlaute([W|Ws],[NW|NWs]) :-
        texMatUmlaute(W,NW),
        ersetzeUmlaute(Ws,NWs).
ersetzeUmlaute([],[]).

texMatUmlaute(W,UW) :-
        W =.. [Fun,Arg|Args],
        !, ersetzeUmlaute([Fun,Arg|Args],[UFun,UArg|UArgs]),
        UW =.. [UFun,UArg|UArgs].
texMatUmlaute(W,UW) :-
```

```
        concat(ä,R,W),!,
        texMatUmlaute(R,UR),
        concat('"a',UR,UW).
texMatUmlaute(W,UW) :-
        concat(ö,R,W),!,
        texMatUmlaute(R,UR),
        concat('"o',UR,UW).
texMatUmlaute(W,UW) :-
        concat(ü,R,W),!,
        texMatUmlaute(R,UR),
        concat('"u',UR,UW).
texMatUmlaute(W,UW) :-
        concat(Ä,R,W),!,
        texMatUmlaute(R,UR),
        concat('"A',UR,UW).
texMatUmlaute(W,UW) :-
        concat(Ö,R,W),!,
        texMatUmlaute(R,UR),
        concat('"O',UR,UW).
texMatUmlaute(W,UW) :-
        concat(Ü,R,W),!,
        texMatUmlaute(R,UR),
        concat('"U',UR,UW).
texMatUmlaute(W,UW) :-
        concat(ß,R,W),!,
        texMatUmlaute(R,UR),
        concat('"s',UR,UW).   % reicht nicht bei Klassen s(daß,...) !
texMatUmlaute('','') :- !.    % Stringende
texMatUmlaute(W,UW) :-
        name(W,[H|L]),
        name(AL,L),texMatUmlaute(AL,UL),
        name(C,[H]), concat(C,UL,UW).
```

Lösung zu Aufgabe 5.3 und 5.4: In den Earley-Dörre-Parser bauen wir zur Lösung von Aufgabe 5.3 die Option `omitEmptyTrees` ein, die dafür sorgt, daß gelöschte `B`-Konstituenten in der Analyse ganz weggelassen werden. Dazu muß sowohl `closure/6` als auch `complete/4` geändert werden.

Zur Lösung von 5.4 bauen wir zusätzlich eine Option `boundedChains` ein, die dafür sorgt, daß auf Kettenableitungen keine Kategorie zweimal benutzt wird.

⟨*Earley-Dörre, Parser*⟩≡

```
:- dynamic kante/6.
:- ensure_loaded('Parser/subsumption').

parse(_,_) :-
        retractall(kante(_I,_J,_A,_alpha,_beta)),
        retractall(kante(_I,_J,_A,_alpha,_beta,_trees)),
        start_symbol(S),
        (S --> Alpha),
        closure(0,0,S,[],Alpha,[]),
        fail.
parse(Wortliste,Baum) :-
        parse(0,Wortliste,Baum).
parse(I,[Wort|Rest],Baum) :-
```

```
        J is I + 1,
        shift(I,J,Wort),
        parse(J,Rest,Baum).
parse(I,[],Baum) :-
        start_symbol(S),
        kante(0,I,S,_Alpha,[],Baeume),
        Baum = [S|Baeume].

shift(I,J,Wort) :-
        lexword(I,Wort,Kategorie),
        closure(I,J,Kategorie,[Wort],[],[[Wort]]),
        fail.
shift(_,_,_).

closure(I,J,Kopf,Geschlossen,[],Baumliste) :-
        neue_kante(I,J,Kopf,Geschlossen,[],Baumliste),
        complete(I,J,Kopf,Baumliste).

closure(I,J,Kopf,Geschlossen,[B|Bs],Baumliste) :-
        deletable(B),
        (ein(omitEmptyTrees)
        -> Baumliste1 = Baumliste,
           Geschlossen1 = Geschlossen
        ; append(Geschlossen,[B],Geschlossen1),
           append(Baumliste,[[B,[]]],Baumliste1)),
        closure(I,J,Kopf,Geschlossen1,Bs,Baumliste1).
```

Hier ergänzen wir zwei Fälle, mit denen optionale und alternative Kategorien auf der rechten Regelseite behandelt werden:

⟨*Earley-Dörre, Parser*⟩+≡

```
closure(I,J,Kopf,Geschlossen,[{B}|Bs],Baumliste) :-
        !,closure(I,J,Kopf,Geschlossen,[B|Bs],Baumliste).

closure(I,J,Kopf,Geschlossen,[(B;C)|Bs],Baumliste) :-
        !,closure(I,J,Kopf,Geschlossen,[B|Bs],Baumliste),
        closure(I,J,Kopf,Geschlossen,[C|Bs],Baumliste).
```

⟨*Earley-Dörre, Parser*⟩+≡

```
closure(I,J,Kopf,Geschlossen,[B|Bs],Baumliste) :-
        neue_kante(I,J,Kopf,Geschlossen,[B|Bs],Baumliste),
        expand(J,B).
```

Eine Kante ist nicht neu, wenn sie die Instanz einer schon vorhandenen Kante ist. Um das zu testen, benutzen wir `known_instance/2` aus `Parser/subsumption.pl`:

⟨*Earley-Dörre, Parser*⟩+≡

```
neue_kante(I,J,Kopf,Geschlossen,Offen,Baumliste) :-
        known_instance(kante(I,J,Kopf,Geschlossen,Offen,Baumliste)),
        !,fail.
neue_kante(I,J,Kopf,Geschlossen,Offen,Baumliste) :-
        (ein(verbose)
        -> nl,nl,write_kante(I,J,Kopf,Geschlossen,Offen,Baumliste)
        ; true),
        assertz(kante(I,J,Kopf,Geschlossen,Offen,Baumliste)).
```

```
complete(I,J,B,Teilbaeume) :-
        kante(K,I,A,Alpha,[B|Beta],Baumliste),
        ((ein(omitEmptyTrees), Teilbaeume = [])
        -> AlphaB = Alpha,
           BaumlisteB = Baumliste
        ; append(Alpha,[B],AlphaB),
           append(Baumliste,[[B|Teilbaeume]],BaumlisteB)),
        ((ein(boundedChains), Beta = [],
          ((I = J, Baumliste = [Baum], stamm(A,Baum))
          ; (K = I, stamm(A,[B|Teilbaeume]))))
        -> true
        ; closure(K,J,A,AlphaB,Beta,BaumlisteB)),
        fail.
complete(_,_,_,_).

expand(J,A) :-
        (A --> Alpha),
        closure(J,J,A,[],Alpha,[]),
        fail.
expand(_,_).

write_kante(I,J,Kopf,Offen,Geschlossen,Baumliste) :-
        tab(3),write(I),write(','),
        write(J),write(': ('),
        write(Kopf),write('-->'),
        write(Offen),write('.'),
        write(Geschlossen),write(') '),
        nl,tab(8),write(Baumliste).
```

Auf dem Stamm eines Baums liegt die Wurzel und alle Knoten, die von der Wurzel über nicht-verzweigende Knoten erreichbar sind:

⟨*Earley-Dörre, Parser*⟩+≡

```
stamm(A,[A|Teilbaueme]) :- !.
stamm(A,[B,Teilbaum]) :-
        stamm(A,Teilbaum).
```

△

Warnung: Der Test, ob eine Kante schon vorhanden ist, kann nicht mit Hilfe der Unifikation vorgenommen werden, wenn in den Grammatikregeln unbeschränkte freie Variablen vorkommen. Sucht man beim Parsen die Regeln mit `(A --> Alpha)` und kommen verschiedene Regeln mit gleicher Form vor (z.B. weil es Disjunktionen in den Anwendungsbedingungen der Regeln gibt), so wird bei der ersten Lösung eine Instanz `(A' --> Alpha')` in die Tabelle eingetragen, die beim Vergleich mit einer zweiten Lösung durch Unifikation dann ggf. dazu führt, daß die zweite Lösung nicht in die Tabelle eingetragen wird, weil sie mit der ersten unifizierbar ist. Sie darf aber nur dann ignoriert werden, wenn sie eine Instanz der ersten Lösung ist. (Wir könnten natürlich auch testen, ob die vorhandene eine Instanz der neuen Lösung ist und ggf. die vorhandene durch die neue ersetzen.)

Bemerkung 10.1 zu Aufgabe 5.4: Ich habe nicht gründlich überlegt, ob man (a) alle divergierenden Fälle beseitigt und (b) keine wesentliche Analyse verliert.

Schaltet man nur die Option `boundedChains` ein, so erhält man (bei der Grammatik `cfg.1.grm.XX.pl` ohne Merkmale) für den Satz `der arme Hamster und der arme Hamster pfeift ein Lied und` `ein Lied` insgesamt 18 Analysen in 0.06 Sekunden (gegenüber dem divergierenden Earley-Doerre-Parser ohne die Option); mit der zusätzlichen Option `omitEmptyTrees` erhält man nur noch 1 Analyse in 0.01 Sekunden. Ohne die Adjektive hat man 162 Analysen in 0.41 Sekunden bzw. 1 Analyse in 0.05 Sekunden.

Der Parser nach Earley-Pratt-Leiß erzeugt ohne Option `omitEmptyTrees` nur 2 Analysen in 0.03 Sekunden. Mit dieser Option erhält man 1 Analyse in 0.03 Sekunden, also etwas langsamer als der Earley-Dörre-Parser mit beiden Optionen. Ohne die Adjektive hat man hier 2 Analysen in 0.02 Sekunden bzw. 1 Analyse in 0.01 Sekunden.

Lösung zu Aufgabe 6.1: Die Aufgabe ist inzwischen durch das Prädikat `erkenneWortarten/0` gelöst, das in `parse/0` und `erkenne/0` eingebaut wurde. $\triangle$

11 Anhang 3: Programm-Lücken, -Änderungen, -Erweiterungen

11.1 Änderungen

Änderungen des bisherigen Programmcodes, die dazu führen könnten, daß manche bisherigen Aufrufe nicht mehr genau so gehen wie früher:

(i) Die Datei `auswahl.muster.pl` ist etwas überarbeitet, damit man einfacher von Prolog aus die Voreinstellungen ändern kann: mit `grammatiken` werden die für einen Parser vorberechneten Grammatiken geladen. Wenn man die Vorberechnung erst erstellen will, muss man `vorberechnungen` laden, das lädt auch `originalgrammatiken`.

(ii) Die Namen der Grammatiken mit Vorberechungen sind zu `<Grammatik>.grm.ed.pl` bzw. zu `<Grammatik>.grm.epl.pl` geändert, damit man leichter die zu `<Grammatik>.grm` gehörenden Versionen erkennen kann.

(iii) Die in der Datei festgelegte Auswahl muß explizit mit `auswaehlen/0` aktiviert werden; das Prädikat `laden/0` ruft nicht mehr `auswaehlen/0` auf, sodaß es auch nach dem interaktiven Ändern der Auswahl nützlich ist, um Tokenizer, Lexer, Lexika, Grammatiken und den Parser zu laden.

(iv) 20.4.02 Bei der Vorberechnung der First-Relation der `.epl`-Parser kann man (mit der Option `base(restrict)` im Menü unter `Vorberechnungen > Auswahl:First`) bestimmte Merkmale der Kategorien bei der Berechnung der Basisrelation von `first/2` ignorieren. Dadurch erfolgt die Vorberechnung schneller und liefert eine kleinere Hilfstabelle `first/2`. (Diese enthält allgemeinere Einträge, filtert also beim Parsen weniger Grammatikregeln als unbrauchbar weg. Durch Experimentieren muß man feststellen, welche Merkmale ignoriert werden müssen, damit die Vorberechung mit dem Stack auskommt und der Parser hinreichend schnell wird.)

Dazu müssen die Datei `Lexika/wortarten.cat.pl` und die Dateien `<Grammatik>.grm` durch `restrict(Kategorie+,Vereinfacht-)` festlegen, welche Merkmale der Kategorie in ihrer vereinfachten Form ignoriert werden.

Das ist erst bei den Grammatiken `np.grm` und `pp.grm` gemacht und noch nicht gründlich getestet; dem Eindruck nach geht es aber damit ganz gut. Für die anderen Grammatiken muß mand bei allen (nicht-lexikalischen) Kategorien noch festlegen, welche Merkmale der Restrictor ausblenden soll!

(v) 25.4.02: Die Umformatierung der CISLEX-Einträge erzeugt statt `word(Wortform,Kategorie)` jetzt neue Einträge der Form `word(Wortform,Kategorie,Stammform)`. Mit Hilfe des Stamms wird durch `flektiere/2` aus `Lexika/umformatieren.pl` ein Wortflexionsprogramm simuliert (wenn alle Vollformen geladen sind).

Das neue Lexikon `Testbett/wissenschaftsreligion.lem.trneu` wird außer zum Testen noch nicht verwendet. (Eine Zurückführung von `word/2` nach `word/3` ist in `Lexika/umformatieren.pl` enthalten, soll aber später woanders hin.)

(vi) 29.6.02: Bei den Transformationsregeln (besser: Flexionsregeln) für Nominalphrasen werden jetzt auch die Nominalphrasen mit Relativsätzen behandelt, außer daß die Relativsätze nicht mit einer relativierenden Postpositionalphrase wie `weswegen`, `dessenthalber` usw. beginnnen dürfen. (Unvollständig sind allerdings auch die Vertauschungen von Konstituenten in Verbletztsätzen, z.B. bei dreistelligen Verben.)

Dazu wurden die Relativpronomina mit einem Stamm versehen, so daß eine Flexion möglich ist, und bei den Formmerkmalen wurde die Person entfernt, da die Relativpronomen `der,die,das` auch bei erster und zweiter Person verwendet werden. (Freie Relativsätze mit `wer,was` sind nicht erfaßt.)

11.2 Erweiterungen

11.2.1 Beispieltexte und Lexika

(i) Ein paar Texte habe ich in `Testbett` hinzugefügt: `wissenschaftsreligion.txt`, den ich als Referenztext für Beispielausdrücke vorschlage, und `selbstgespraech.txt`, als ein m.M. komplizierterer Text. Fragen satt Aussagen enthält `nachgefragt.txt`, ein Interview.

(ii) CIS-Lexika für die Beispieltexte liegen in `Testbett/*.lem`. Aus ihnen sind vorläufige Lexika in dem Format, das die Parser verwenden, berechnet worden und als `Testbett/*.lem.tmp.wrd` (und z.T. nach Wortarten gegliedert in `Testbett/*.lem.<Wortart>.tmp.wrd`) gespeichert. Sie enthalten aber noch Fehler und fehlende Merkmale. Man muß sie selbst an Prolog anpassen!

Nur wenige sind von Hand korrigiert und unter `Lexkia/*.lem.<Wortart>.wrd` gespeichert.

Nominalisierte Adjektive werden in eine eigene Datei geschrieben, allerdings nur in Kleinschreibung; sie haben Adjektivparameter.

(iii) Mit `erkenneWortarten/0` könnte man alle Lexikon-Einträge darauf überprüfen, ob sie das von der (vorberechneten) Grammatik gewünschte Format haben.

11.2.2 Tokenizer und Lexikalische Analyse

(i) Die Liste der Abkürzungszeichenreihen wird jetzt aus einem Abkürzungslexikon abgeleitet, das für die syntaktische Analyse nützlich ist. Bisher sind nur in der Nominalphrasengrammatik die Regeln so angepasst, daß man diese Abkürzungen auch verwenden kann.

Abküzungen mit einer *syntaktische* Kategorie, wie `z.B.` und `a.a.O.`, erfordern keine Regeländerungen.

11.2.3 Grammatikformat

(i) Die Grammatiken *müssen* jetzt ein Prädikat `auxiliary_predicates([start_symbol/1 | Preds])` enthalten: die Vorberechnung überträgt diese Prädikate in die vorberechnete Grammatikdatei.

(ii) Die Grammatikregeln erlauben auf den rechten Seiten jetzt Disjunktionen von Kategorien, etwa `(np;pp)`, und die Angabe optionaler Konstituenten, etwa `{np}`. Zusammen mit `omitEmptyTrees` kan man mit `{Cat}` optionale von obligatorischen Konstituenten/Vorkommen der Kategorie `Cat` unterscheiden. In den erzeugten Syntaxbäumen wird nicht angezeigt, ob eine Konstituente eine optionale Kategorie einer Regel verwirklicht oder eine obligatorische.

Die Vorberechnung für den Earley-Pratt-Parser ist so angepaßt, daß in den Relationen `chain/2` und `first/2` nur *einfache* Kategorien benutzt werden! Der Umgang mit disjunktiven und optionalen Kategorien erfolgt im Earley-Pratt-Parser durch einen Subsumierbarkeitstest `instance/2` und im Earley-Dörre-Parser durch Zusatzklauseln in `closure/6`.

11.2.4 Vorberechnungen und Grammatiktransformationen

(i) Man könnte bei der First-Relation den Teil mit lexikalischer erster Komponente (die eigentliche First-Relation) vom Rest (den anderen Tupeln der *Leftcorner*-Relation) abtrennen.

Der Earley-Pratt-Leiß-Parser braucht nur an einer Stelle (für den Predictor) die Leftcorner-Relation, also beide Komponenten nicht lexikalisch, aber an den anderen Stellen nur die eigentliche First-Relation! Die Trennung sollte also die Laufzeit beim Parsen verbessern.

(ii) Die Vorberechnungen für den Earley-Pratt-Parser sind wie folgt verändert worden:

(a) Die Berechnung der irreflexiven transitiven Hülle einer Relation zwischen Termen erlaubt jetzt auch Terme mit freien Variablen.

(b) Die Vorberechnung des Predictors scheint zu divergieren, bei der Regel `np -> np, conj, np` mit Parametern.

(iii) Die Berechnung der transitiven Hülle einer Relation für die Ketten- und First-Relationen scheint schnell genug (für `pp1.grm`).

(iv) Die Berechnung der lexikalischen Kategorien ist jetzt nicht mehr mit einer Schleife, sondern durch `setof` und geändertes `add_LexCats` geschrieben. (Ein Fehler in `lexCat` verkleinerte den statischen Predictor.)

11.2.5 Parser

(i) Das Prädikat `deletable/1` im Parser *Earley-Dörre* werde ich noch in `del/1` umbenennen, damit man `deletable/1` für rechte Regelseiten im DCG-Format haben kann (wie z.T. in den Vorberechnungen). Nachtrag 16.11.: Einfachheitshalber habe ich `deletable/1` nach der Vorberechnung in die erzeugte Grammatik übertragen.

(ii) Earley-Parser mit Vorberechnungen.

21.11.01: Darin einen Fehler beseitigt: Bei leerer Eingabe lieferte `setof(C,parse([],C),Cs)` die Antwort `Cs = [_X]`, die dann durch `baeume([],Ts),showASCIItrees(Ts)` zu einer Endlosschleife führte. Es fehlte `parse([],_) :- !,fail.`

(iii) Lösung von Aufgabe 5.3 zur Option `omitEmptyTrees` angegeben. Aufgabe 5.4 zum dynamischen Verhindern von zu langen Kettenableitungen im Earley-Doerre-Parser gestellt.

(iv) 21.11.01: `parse/0` und `erkenne/0` rufen einen Test, `erkenneWortarten/0`, auf, der nachprüft, ob die von der lexikalischen Analyse gefundenen Wortarten unter den Terminalkategorien der Grammatik vorkommen.

Dazu wird bei der Vorberechnung der Hilfstabellen ein Prädikat `lexCat/1` berechnet und in die umgeformte Grammatik `<Grammatik>.wrd.ed|epl.pl` übertragen. (Wenn man die Grammatik nicht neu vorberechnen will, kann man einfach `lexCat(_).` in die Grammatikdatei aufnehmen, damit `erkenneWortarten/0` nicht scheitert.)

11.2.6 Baumdarstellung

Die in der Vorlesung schon kurz benutzte graphische Baumdarstellung ist bei der Lösung zu Aufgabe 5.2 als `showTeXtrees(Baumliste+)` hinzugefügt und wird nach `Parser/PrintTree/showtree.pl` übertragen.

Damit sie benutzt werden kann, muß

(i) LaTeX installiert sein,

(ii) LaTeX den Pfadnamen zum neu hinzugefügten Verzeichnis `TeX` kennen.

(iii) `xdvi` installiert sein,

(iv) vor dem Parsen die Option `einschalten(showTrees)` gesetzt werden.

Das neue Verzeichnis `TeX` enthält die Quelldateien zu `TreeTeX` und eine von mir geschriebenen Änderungsdatei `arity.tex`, damit man mehrfach verzweigende Bäume darstellen kann. (Leider habe ich es nicht geschaft, die Beschriftungen anständig ins Zentrum der Baumknoten zu schreiben.)

Damit LaTeX den Pfadnamen kennt, kann man unter Linux

(i) mit `echo $TEXINPUTS` nachsehen, welche Verzeichnisse LaTeX durchsucht; falls darunter `~/.TeX` vorkommt, kann man einfach die Dateien aus `TeX` in das Verzeichnis `~/.TeX` übertragen.

(ii) die Liste der Pfadnamen unter `$TEXINPUTS` mit `setenv` (bzw. `set`) um `:~/<Verzeichnis der Grammatikimplementierung>/TeX` erweitern.

Die graphische Darstellung von Bäumen hinterläßt eine Datei `baum.tmp.tex`, die von der nächsten Anzeige überschrieben wird.

Warnung: Die Baumdarstellung durch `TreeTeX` hat einen Fehler: die Größe der darstellbaren Bäume ist beschränkt durch die 256 `box`-Register von `TeX`. Ich habe nicht gesehen, wie man Boxen in TeX wiederverwenden kann; daher werden alle Register auch dann verbraucht, wenn zwar die Analysen klein sind, aber die Anzahl der Analysen einer Eingabe groß ist. In diesem Fall gerät `LaTeX` in eine Schleife, *die man zwar anhalten, aber nicht abbrechen kann. Man muß dann den Prolog-Prozeß, der* `TreeTeX` *aufruft, abschießen.*

11.2.7 Hilfsprogramme

(i) In der Lösung zu Aufgabe 3.2 ist eine Funktion `satznumerieren(Eingabedatei+,Ausgabedatei+)` hinzugefügt, damit wir uns bei Beispielen leichter auf die einzelnen Sätze eines Textes beziehen können.

(ii) Berechnung der transitiven Hülle einer zweistelligen Relation. Der Algorithmus von Warshall für die Berechnung der transitiven Hülle ist auf Terme mit freien Variable erweitert, aber das ist noch weiter zu prüfen.

(iii) Zum Testen der Grammatiken sind in Abschnitt 5.2.2 die Programme `erkennePhrasen` und `analysierePhrasen` eingefügt worden, mit denen man eine Datei von Phrasen auf ihre grammatische Korrektheit überprüfen kann. Die Antworten werden in eine Ergebnisdatei geschrieben, die zur Unterscheidung mit einem Parser- und Datumssuffix versehen ist.

Es fehlt noch ein Programm, mit dem man *Teil*phrasen von Sätzen erkennen kann.

11.3 Fehler

11.3.1 Lexikon

Es gibt (abgesehen von den fehlenden Klassen- und Formparametern, die man selbst ergänzen muß) Fehler im Lexikon: manche Paradigmen sind unvollständig oder haben falsche Einträge.

(i) Bei Adjektiven fehlt die Form `[Komp,st,Genus,pl,gen]`, die man z.B. für `wegen wissenschaftlicher Gesetze` bräuchte:

⟨*lexikonfehler*⟩≡
```
lexword(0, wissenschaftlicher, a([], [pos, ge, mask, sg, nom])).
lexword(0, wissenschaftlicher, a([], [pos, sw, mask, sg, nom])).
lexword(0, wissenschaftlicher, a([], [pos, st, fem, sg, gen])).
lexword(0, wissenschaftlicher, a([], [pos, st, fem, sg, dat])).
lexword(0, wissenschaftlicher, a([], [pos, st, unsp, pl, dat])).
lexword(0, wissenschaftlicher, a([], [komp, praedik])).
```

Außerdem fehlen die Formen `[Komp,sw,Genus,pl,gen]` und `[Komp,ge,Genus,pl,gen]`, die man z.B. bei `der wissenschaftlichen Gesetze` und `keiner wissenschaftlichen Gesetze` braucht. Die einzigen Pluralformen sind

⟨*lexikonfehler*⟩+≡
```
lexword(0, wissenschaftliche,  a([], [pos, st, unsp, pl, nom])).
lexword(0, wissenschaftlichen, a([], [pos, sw, unsp, pl, nom])).
lexword(0, wissenschaftlichen, a([], [pos, ge, unsp, pl, nom])).
lexword(0, wissenschaftlichen, a([], [pos, st, unsp, pl, gen])).
lexword(0, wissenschaftlicher, a([], [pos, st, unsp, pl, dat])).
lexword(0, wissenschaftliche,  a([], [pos, st, unsp, pl, akk])).
lexword(0, wissenschaftlichen, a([], [pos, sw, unsp, pl, akk])).
lexword(0, wissenschaftlichen, a([], [pos, ge, unsp, pl, akk])).
```

Es fehlen also auch die Formen `[Komp,ge|sw,unsp,pl,dat]`, so daß z.B. `keinen|den wissenschaftlichen Gesetzen` nicht erkannt werden.

Das scheint bei allen Adjektiven so zu sein!

(ii) Bei manchen Einträgen hat man doppelte Lesarten, z.B. erhält man für `die` die folgenden Einträge:

⟨*lexikonfehler*⟩+≡
```
lexword(0, die, det([def, sw], [mask, pl, nom])).
lexword(0, die, det([def, sw], [mask, pl, akk])).
lexword(0, die, det([def, sw], [unsp, pl, nom])).
lexword(0, die, det([def, sw], [unsp, pl, akk])).
lexword(0, die, det([def, sw], [fem, sg, akk])).
lexword(0, die, det([def, sw], [fem, sg, nom])).
```

Hier stört die doppelte Lesart im Plural. Allerdings liefert das CISLEX korrekterweise *nur* die `unsp`-Lesart; die `mask`-Versionen müssen von einer Datei der Seminarteilnehmer oder aus `Lexikon/praepositionen.wrd` kommen! (Ja, sollte behoben sein.)

(iii) Bei den Personalpronominal hat das CISLEX anscheinend fehlende oder falsche Einträge:

⟨*lexikonfehler*⟩+≡
```
% CISLEX: word(ihrer, pron([def,pers], [unsp, pl, gen, 2])).
% CISLEX: word(euch, pron([def,pers], [unsp, pl, gen, 2])).
% CISLEX: word(man, pron([indef,pers], [unsp, sg, akk, 3])).
% CISLEX: word(man, pron([indef,pers], [unsp, sg, dat, 3])).
% CISLEX: word(man, pron([indef,pers], [unsp, sg, gen, 3])).
% CISLEX fehlte: word(sie, pron([def,pers], [unsp, pl, akk, 3])).
```

(iv) Ebenfalls falsch ist wohl die Adjektivdeklination bei `alle`, die `sw` sein muß:

⟨*lexikonfehler*⟩+≡
```
% CISLEX: word(alle, det([quant,st], [unsp, pl, nom])).
```

(v) Gelegentlich stimmt im CISLEX die Längenangabe des abtrennbaren Verbpräfixes nicht, die Zahl nach `#` in der Kodierung. Statt

⟨*lexikonfehler*⟩+≡
```
abschaff,abschaffen.VSW1#0:eb
```

muß es z.B. `abschaff,abschaffen.VSW1#2:eb` heißen.

(vi) Bei der Erzeugung der Wortformen mit dem CISLEX werden alle Nomina mit kleinen Anfangsbuchstaben geschrieben. Die Übersetzung des erzeugten Lexikons hatte nur die Nomina in Großschreibung übertragen, die nicht mit einem Umlaut anfingen. Das ist jetzt korrigiert, aber nicht getestet. (Es wird vorausgesetzt, daß das groß zu schreibende Wort mit einem Kleinbuchstaben beginnt.)

In `Lexika/n.ddm.wrd` habe ich die entsprechenden Einträge von Hand korrigiert.

(vii) `Erkenntnis` wird einmal als Femininum und dann als Neutrum angegeben.

(viii) Das Abkürzungslexikon `abkuerzung/2` enthält noch Abkürzungen, bei denen die syntaktische Kategorie fehlt.

11.3.2 Grammatik

(i) Grammatik `pp1`: Die Kongruenzbedingungen bei der Nominalphrase bekomme ich wegen der Lexikoneinträge nicht so hin, wie sie sein sollten: die Plural-Lesart von `der Wissenschaftler` ist ambig, da `der` einmal mit unspezifischem Genus und einmal im Maskulinum mit dem Nomen kongruiert. Die erste Lesart kann ich nicht unterdrücken.

(ii) Grammatik `pp1.epl`: zwar wird `bis auf den Mann` erkannt, aber nicht `bis zu dem Mann`. Das könnte an den Lexikoneinträgen

⟨*Fehler PP*⟩≡
```
word(zu, pr([[dat]], [zu])).
word(zu, pr([[dat]], [zu])).
word(auf, pr([[dat]], [auf(dat)])).
word(auf, pr([[akk]], [auf(akk)])).
```

liegen, oder an den PP-Regeln? Am ehesten liegt es an den Einträgen

⟨*Fehler PP*⟩+≡
```
word(bis, pr([[dir(hin)]], [bis])).
word(bis, pr([[temp]], [bis])).
word(bis, pr([[kop]], [bis])).
word(bis, pr([[auf(akk)]], [bis])).
word(bis, pr([[zu(dat)]], [bis])).
```

(iii) Grammatik `vp`: Präpositionalobjekte gehen nicht bei allen Verbformen: Es geht zwar

⟨*Fehler PP-Objekte*⟩≡
```
Er sieht das Gesetz für Gewohnheitsverbrecher vor.
Er hat das Gesetz für Gewohnheitsverbrecher vorgesehen.
Er schreitet zu einer Maßnahme.
```

aber die folgenden Versionen werden nicht erkannt:

⟨*Fehler PP-Objekte*⟩+≡
```
das Gesetz ist für Gewohnheitsverbrecher vorgesehen.
Er ist zu einer Maßnahme geschritten.
```

Mit manchen Präpositionen (und entsprechenden Verbeinträgen) klappt es doch:

⟨*Fehler PP-Objekte*⟩+≡
```
Es ist in Gewohnheitsverbrecher vorgesehen.
Er ist in einer Maßnahme geschritten.
```

Es kann also an den Präpositionalphrasen und den Verbrahmen (aus den Lexika der `vp`-Grammatik) liegen. Es kann aber auch sein, daß das Ganze ein Folgeproblem der fehlenden Disjunktionen (s.u. J.Stiehler) ist.

(iv) Grammatik `np`: Bei den Titeln und Vornamen werden keine Wiederholungen erlaubt, also `Prof. Dr. K.U. Schulz` ausgeschlossen.

11.3.3 Parser

(i) Vorberechnungen: Die Vorberechnung des Predictors scheint bei der Grammatik `np.grm` mit der Koordinationsregel `np --> np,conj,np` und der Kettenregel `np --> pron` sowie entsprechenden Merkmalen in eine Schleife zu geraten. z.B. bei `predict_static(np([def,3,unsp],[Num,Kas]), pron([],[unsp,pl,Y,3]))`.

Da das Problem ohne Merkmale nicht auftritt, könnte es an impliziten Zyklen in der `First`-Relation liegen.

Die Schleife tritt zwar nach Änderung von `add/2` nicht mehr auf, aber mit dieser Regel endet die Vorberechung in `No`.

(ii) Vorberechnungen: Ein Prädikat wird bei der Vorberechung für die `.epl`-Parser nicht exportiert. Manchmal beschwert sich der Parser, daß ihm bei `chain/2` ein `genusErw` fehlt. Wenn man die Vorberechnung neu macht, ist das Prädikat vorhanden; aber um welches geht es genau und welche `auxiliary_predicates`-Deklaration fehlt?

Es scheint, daß eine der Hilfsdateien neu geladen werden müßte, aber nur ein `ensure_loaded` benutzt wird. Am besten macht man die Vorberechnung mit direkt am Anfang eines Neustarts von Prolog.

(iii) Beim Einbau der Eigennamenregeln kann er zwar `die Entdeckung Sigmund Freuds` richtig analysieren, aber bei `Sigmund Freud(s)` ist der Baum nicht korrekt (im Vergleich zu `die Entdeckung`) – die LaTeX-Darstellung bekommt und erzeugt weitere Probleme (offene Datei?).

(iv) Beim Earley-Doerre-Parser scheinen die Konstanten, die zum Matchbarkeitstest benutzt werden, in der Datenbank zu bleiben. Jedenfalls bekam ich beim Parsen von `Sie offenbart ihm ihn.` mit der Grammatik `s.grm.ed` Zwischenergebnisse der Form

⟨*Fehler*⟩≡

```
       [21] kante(0, 1, s([def], [const(3576195), const(3576196), zweit]),
                   [np([def, 3, unsp], [pl, nom])],
                   [v([[nom, gen, akk]], [3, pl, const(3576195), const(3576196)]),
                    np([def, const(3576197), const(3576198)], [const(3576199), akk]),
                    np([def, const(3576200), const(3576201)], [const(3576202), gen])],
                   [[np([def, 3, unsp], [pl, nom]), [pron([def, pers], [unsp, pl, nom|...]), [sie]]]])
```

und die haben ja wohl nichts in der Tabelle verloren! Na ja, sie stehen nicht in der Tabelle, sondern es sind Aufrufe, mit denen festgestellt wird, ob eine Kante eine Instanz einer vorhandenen Kante ist.

11.3.4 Behobene Fehler

(i) Mit `pp1.epl` wird `auf wissenschaftliche Gesetze` nicht erkannt, nur `auf die wissenschaftlichen Gesetze`. Warum?

```
1. pp([def(3, pl)], [auf(akk)])
      pr([ [akk] ], [auf(akk)])
         auf
      np([def, 3, neut], [pl, akk])
         det([def, sw], [unsp, pl, akk])
            die
         n1([neut], [pos, sw, pl, akk])
            a([], [pos, sw, unsp, pl, akk])
               wissenschaftlichen
            n([neut], [pl, akk])
               Gesetze
```

```
1. n1([neut], [pos, st, pl, akk])
      a([], [pos, st, unsp, pl, akk])
         wissenschaftliche
      n([neut], [pl, akk])
         Gesetze
2. n1([neut], [pos, st, pl, nom])
      a([], [pos, st, unsp, pl, nom])
         wissenschaftliche
      n([neut], [pl, nom])
         Gesetze
```

Der Grund war ein zu kleiner Predictor, der zu `pp --> praep,np.`, `np --> n1.` und `n1 --> a,n` in `prediceted(np,a)` die Regel `n1 --> a,n.` nicht aufnahm, wegen einer fehlerhaften Beschleunigung des ursprünglichen statischen Predictors.

(ii) Fehlende Lesarten: Trotz der beiden Lesarten

```
1. n1([neut], [pos, st, pl, akk])
      a([], [pos, st, unsp, pl, akk])
         naturwissenschaftliche
      n([neut], [pl, akk])
         Gesetze
2. n1([neut], [pos, st, pl, nom])
      a([], [pos, st, unsp, pl, nom])
         naturwissenschaftliche
      n([neut], [pl, nom])
         Gesetze
```

und der Lexikoneinträge

```
lexword(0, als, pr([ [nom] ], [als])).
lexword(0, als, pr([ [gen] ], [als])).
lexword(0, als, pr([ [dat] ], [als])).
lexword(0, als, pr([ [akk] ], [als])).
```

bekommt man nur die eine Lesart

```
1. pp([indef(3, pl)], [auf(akk)])
      pr([ [akk] ], [auf(akk)])
         auf
      np([indef, 3, neut], [pl, akk])
         n1([neut], [pos, st, pl, akk])
            a([], [pos, st, unsp, pl, akk])
               naturwissenschaftliche
            n([neut], [pl, akk])
               Gesetze
```

während die analoge Lesart mit dem Nominativ fehlt. Durch die Anpassung des Warshall-Algorithmus ist das beseitigt: vermutlich war die First-Relation fehlerhaft.

(iii) Grammatik `np` bzw. `pp1`: Genitiv-Attribute kann man nur an Nomina im Singular anschließen. Was schließt den Anschluß beim Plural aus?

Der Grund war, daß `lexCat(n([A], [sg, B]))` der einzige Eintrag zu Nomen war, und daher der statische Predictor zu klein wurde: die Regeln `n1 --> n, {npgen ; pp}` fehlten für den Plural. Woher das kommt, bleibt unklar. Es ist dadurch behoben, daß `lexcat/1` mit `setof` statt einer Schleife programmiert ist.

(iv) Beim EPL-Parser werden nicht alle Lesarten ausgegeben: da die Tabelle nur kanonische Bäume aufbaut, fehlen die Lesarten, die von einer Startkategorie nur mit Kettenregeln erreichbar sind, z.B. Personalpronomina als Nominalphrasen. Der EPL-Parser sollte also noch Kettenableitungen von Startsymbolen zu den erkannten Phrasen erlauben.

Das ist durch Änderung von `extract(I,J,Baum)` korrigiert worden. (Allerdings wird angenommen, daß die Startkategorien keine disjunktiven Kategorien `(Cat1;Cat2)` oder optionalen Kategorien `{Cat}` sind.)

(v) Es gibt ein Problem mit disjunktiven Bedingungen (in der Grammatik `np.0` von Johannes Stiehler), das ich nicht rekonstruieren kann. Je nachdem, welche Bedingung man zuerst angibt, kommt eine andere Lösung als die einzige.

Es kann sein, daß es mit dem Problem bei Postpositionen auftritt, die nur im Neutrum funktionieren, wenn die Grammatikregeln die Prädikate zur Definition der Merkmalbereiche nicht enthalten. Bei Christoph Ringlstetter trat es etwa so auf:

⟨*Testbett/Fehler/disjunktionen.grm*⟩≡

```
auxiliary_predicates([start_symbol/1,word/2,arg/1]).

start_symbol(rels).

(rels --> relpro(Kas1), np(Kas2), v(Arg1,Arg2)) :-
       % arg(Arg1),
       % arg(Kas1), arg(Kas2),
       (Kas1 = Arg1 ; Kas2 = Arg1).

loeschbar --> [].

word(der, relpro(nom)).
word(den, relpro(akk)).
word(er, np(nom)).
word(ihn, np(akk)).
word(sah, v(nom,akk)).

arg(nom).
arg(akk).
```

Ohne das Bereichsprädikat `arg(Arg1)` in der Bedingung der Regel erkennt der Parser (sowohl Earley-Doerre wie Earley-Pratt) nur den Satz `den er sah`, aber nicht `der ihn sah`.

Wenn die Bedingung `(Kas2 = Arg1 ; Kas1 = Arg1)` lautet, wird umgekehrt nur der zweite Satz erkannt. Entsprechend verhält es sich, wenn man statt der Disjunktion in der Bedingung zwei Regeln schreibt.

⟨*fehler.disjunktion.grm*⟩≡

```
86 ?- parse. der ihn sah.
   0,0: (rels-->[].[relpro(_G479), np(_G484), v(_G479, _G490)])
        []
   0,1: (relpro(nom)-->[der].[])
        [[der]]
   0,1: (rels-->[relpro(nom)].[np(_G490), v(nom, _G496)])
```

```
        [[relpro(nom), [der]]]
    1,2: (np(akk)-->[ihn].[])
        [[ihn]]
    0,2: (rels-->[relpro(nom), np(akk)].[v(nom, _G499)])
        [[relpro(nom), [der]], [np(akk), [ihn]]]
    2,3: (v(nom, akk)-->[sah].[])
        [[sah]]
    0,3: (rels-->[relpro(nom), np(akk), v(nom, akk)].[])
        [[relpro(nom), [der]], [np(akk), [ihn]], [v(nom, akk), [sah]]]
1. rels
      relpro(nom)
         der
      np(akk)
         ihn
      v(nom, akk)
         sah
```

Dagegen lautet die Tabelle bei der anderen Eingabe:

⟨*fehler.disjunktion.grm*⟩+≡

```
87 ?- parse. den er sah.
    0,0: (rels-->[].[relpro(_G464), np(_G469), v(_G464, _G475)])
        []
    0,1: (relpro(akk)-->[den].[])
        [[den]]
    0,1: (rels-->[relpro(akk)].[np(_G475), v(akk, _G481)])
        [[relpro(akk), [den]]]
    1,2: (np(nom)-->[er].[])
        [[er]]
    0,2: (rels-->[relpro(akk), np(nom)].[v(akk, _G484)])
        [[relpro(akk), [den]], [np(nom), [er]]]
    2,3: (v(nom, akk)-->[sah].[])
        [[sah]]
Keine syntaktische Analyse gefunden.
```

Wenn man die Bedingung `arg(Arg1)` aktiviert, werden mit dem Earley-Pratt-Parser *beide* Sätze erkannt. Wenn man zusätzlich auch `arg(Kas1)` oder `arg(Kas2)` aktiviert, erkennt auch der Earley-Doerre-Parser beide Sätze.

Vermutlich wird *bei der Suche nach einer Grammatikregel* durch `(A --> Alpha)` (und gewisse `!` im Parser?) ohne die Bereichsprädikate nur *ein* Beweis der Regel, und damit nur *eine* Wahl der Variablen, gefunden.

Hier sind die Tabellen des Earley-Pratt-Parsers:

⟨*fehler.disjunktion.grm*⟩+≡

```
17 ?- erkenne. der ihn sah.

ptrees(0, 0, rels, [], [relpro(nom), np(A), v(nom, B)]).
ptrees(0, 0, rels, [], [relpro(nom), np(akk), v(akk, A)]).
ptrees(0, 1, rels, [relpro(nom)], [np(akk), v(nom, A)]).
ptrees(0, 1, rels, [relpro(nom)], [np(akk), v(akk, A)]).
ptrees(0, 2, rels, [relpro(nom), np(akk)], [v(nom, akk)]).

ctrees(0, 1, relpro(nom)).
ctrees(1, 2, np(akk)).
ctrees(2, 3, v(nom, akk)).
```

```
ctrees(0, 3, rels).

18 ?- erkenne. den er sah.

ptrees(0, 0, rels, [], [relpro(akk), np(A), v(akk, B)]).
ptrees(0, 0, rels, [], [relpro(akk), np(nom), v(nom, A)]).
ptrees(0, 1, rels, [relpro(akk)], [np(nom), v(akk, A)]).
ptrees(0, 1, rels, [relpro(akk)], [np(nom), v(nom, A)]).
ptrees(0, 2, rels, [relpro(akk), np(nom)], [v(nom, akk)]).

ctrees(0, 1, relpro(akk)).
ctrees(1, 2, np(nom)).
ctrees(2, 3, v(nom, akk)).
ctrees(0, 3, rels).
```

Ein analoges Problem kann auftreten, wenn man Disjunktionen in der Regel hat. In der Grammatik `np.grm` tritt die Regel

⟨*Testbett/Fehler/disjunktion-2.grm*⟩≡

```
n1([Gen],[pos,-,Num,Kas]) -->
        n([Gen],[Num,Kas]),
        {np([Def2,3,Gen2],[Num2,gen]) ; pp([DefPP],[Praep])}
        :-
        % kasus(Kas),
        % genus(Gen),
        numerus(Num).   % nötig für Plural !!
```

auf. Ohne die Bereichsbeschränkung `numerus(Num)` erhält man nur die `sg`-Fälle, z.B. `Gesetz der Naturwissenschaft`, aber nicht die `pl`-Fälle, z.B. `Gesetze der Naturwissenschaft`. (Bei der Regel mit dem Adjektiv beim Kopfnomen ist `numerus(Num)` nicht nötig! Das ist noch ziemlich dubios ...)

Das Problem war, –wie vermutet– daß beim Testen, ob (neue?) Einträge in den Tabellen schon vorhanden sind, durch einen Versuch, den Neueintrag zu beweisen, durch Unifikation die Gleichheit mit einem Vorhandenen Eintrag festgestellt und dann kein Eintrag vorgenommen wurde. Das ist durch Änderung von `add/2` behoben. (Allerdings für die Ergänzung Fakten, nicht für die von Regeln.)

(vi) Vorberechungen: Mit dem ursprünglichen Warshall-Agorithmus ist die Vorberechnung nicht korrekt, wenn freie Variablen in den Regeln nicht durch Bereichsprädikate beschränkt werden.

Das Problem liegt darin, daß man bei den Vorberechnungen mit `(A-->Alpha)` eine Regel sucht und dabei schon die Bedingungen löst. Falls die Bereichsprädikate fehlen, erhält man Regeln oder Tupel mit freien Variablen. Die ursprüngliche Version des Algorithmus von Warshall spezialisiert die Grundrelation durch Belegen freier Variable!

Die Anpassung des Warshall-Algorithmus' auf Relationen zwischen Termen mit freien Variablen hat das behoben. Aber die Vorberechung mit freien Variablen enthält einen neuen Fehler in der Berechnung der First-Relation: neue Tupel, deren Komponenten echte Instanzen vorhandener Knoten sind, werden nicht korrekt mit den ursprünglichen Tupeln kombiniert:

⟨*Testbett/fehler.first-2.ap.grm*⟩≡

```
first(ap(*), n1(A)).
first(ap2(A), ap(A)).
first(a(A), ap2(A)).

116 ?- compute_first.
```

```
117 ?- listing(first).

first(ap(*), n1(A)).
first(ap2(A), ap(A)).
first(a(A), ap2(A)).
first(a(A), ap(A)).
first(ap2(*), n1(A)).

118 ?- compute_first.
119 ?- listing(first).

first(ap(*), n1(A)).
first(ap2(A), ap(A)).
first(a(A), ap2(A)).
first(a(A), ap(A)).
first(ap2(*), n1(A)).
first(a(*), n1(A)).
```

Die ursprüngliche Version von `itc` hat also nicht die ganze irreflexive transitive Hülle berechnet. Das ist zwar durch eine Änderung von `extend/4` zu `extend/5` behoben, aber jetzt platzt wieder der Stack! Man MUSS die Vorberechnung mit Abspeichern von Zwischenergebnissen machen!!

(vii) Der Erkenner zu `app.grm.epl` erkennt weniger als der Parser - woran liegt das? (Geändert ist nur der Umgang mit Ordinal- und Kardinalzahlen, und wegen der Kardinalzahlen der Begriff der `LexCat`, ggf. unnötigerweise.

Die Definition von `compute_lexCat` war falsch und im Erkenner fehlte noch ein `instance(BI,B)` beim Completer. (vgl. `Testbett/fehler.predicted.ap.grm`).

(viii) Für Vergleichskonstruktionen müssen die Präpositionen `wie` und `als` besonders behandelt werden: sie regieren nicht nur den Kasus der eingebetteten Nominalphrase nicht (was implementiert ist), sondern der Kasus muß auch an der Form der Präposition ablesbar sein. Erst dann kann die Kongruenz in `np(Art1,[Kas,..]) (als|wie np(Art2,[Kas,...])) : pp(Art,als(Kas))` überprüft werden.

Die Lexikoneinträge der Präpositionen `wie` und `als` sind so geändert, daß diese den Kasus nicht bestimmen, er aber als Formmerkmal der `pp` erscheint. Damit hat man sozusagen jeweils 4 verschiedene Präpositionen.

(ix) Um bei den Vorberechnungen nicht so schnell an Speichergrenzen zu stoßen, kann man SWI-Prolog beim Starten durch `pl -L40m -G15m` mit lokalem Stack von hier 40 MB und globalem Stack von 15 MB starten. Um das auch im Emacs zu haben, muß man mit `M-x customize-variable` die Variable `prolog-program-switches` mit `((swi (L40m" G15m")))` belegen.

(x) Vorberechnungen: Es muß noch einen Fehler geben, da der `.epl`-Parser Sätze wie `Von wem schwärmt er?`, die mit `qu.grm.ed` erkannt werden, mit `qu.grm.epl` nicht erkannt werden, wenn man mit Option `basefirst(restrict)` und `tc(warshall-4)` vorberechnet. Die Tabelle ist überschaubar genug, um den Fehler zu finden.

Der Fehler lag in `predict_static/2`: falls die linke Ecke einer Regel eine Disjunktion war, hatte der statische Prediktor nicht genügend vorhergesagt. Durch Einfügen von `instance(B,BI)` etc. behoben.

(xi) Optionen im Menü: die Auswahl `pruefeLexWord` bei der Auswahl der Parser schaltet die Option nicht ein.

Der Fehler lag in `Emacs/.emacs.grammar.el`. Es wurde stattdessen die Option `filterLexWord` ein- und ausgeschaltet.

11.4 Noch zu machen:

11.4.1 Lexika und Beispieltexte

(i) Der Aufbau der Lexika sollte bei den Verben, Nomina und Adjektiven während der Übersetzung aus dem CISLEX den Benutzer nach den gewünschten Komplementrahmen fragen. Die Antwort sollte dann in alle Formen eingebaut werden, während die Vollformen an das im Parser benutzte Format angepaßt werden.

Bem.: Stattdessen ist jetzt ein Stammlexikon für Verben, Adejktive und Nomen angelegt, in das man die Komplementrahmen einfügen kann, sofern sie nicht vom CISLEX-V übernommen werden. Dazu muß aber das Vollformenlexikon bei allen Wörtern, zu denen es Formenvarianten gibt, die Stammform enthalten.

(ii) Die Einträge `Seminarbeitraege/HasselLiTonelli/konjunktionen.wrd` kontrollieren und als `Lexika/konjunktionen.wrd` übernehmen, auch in der Dokumentation. Ebenso die Adjektive aus `Seminarbeitraege/LenkLeonardi/adjektive.wrd` und andere Lexika.

(iii) Sollte man in den Komplementrahmen auch optionale Komplemente notieren, z.B. `word(lesen,v([ [nom,{akk}] ],Form))`, `word(begründet,adj([{von}],Form))`? Das würde ggf. gegenüber den ausmultiplizierten alternativen Lexikoneinträgen später Parsezeit sparen.

11.4.2 Lexikalische Analyse

(i) Um interaktiv irrelevante Lexikoneinträge aus dem Sublexikon zu entfernen oder sich auf eine gewünschte Lesart pro Wort festzulegen, kann man die Option `filterLexAnalyse` benutzen.

Man sollte aber auch eine automatische Wegfilterung irrelevanter Lexikoneinträge implementieren. Sie sollte z.B. umfassen:

(a) entferne Lesarten von `der|die|das` als Relativpronomen am Anfang der Eingabe,

(b) entferne Lesarten als Teil einer zweigliedrigen Präposition wie `(von, an)`, wenn der andere Teil der Präposition in der Eingabe gar nicht vorkommt.

(ii) Die lexikalische Analyse muß korrigiert werden, wenn man Satzzeichen nicht aus der Eingabe entfernt: in diesem Fall wird z.Zt. das Satzzeichen als seine Kategorie gelesen, es sollte aber in `word/2` nachgesehen werden, ob es eine lexikalische Kategorie dafür gibt. (Problem bei `Seminarbeitraege/Hick/synHick.grm`).

(iii) Was macht man mit Mehrwortlexemen, Redewendungen etc. im Lexikon und im Parser??

(iv) Vielleicht sollte man die Lexika gar nicht in Prolog laden, sondern externe Dateien `<Lexikon>.wrd` durchsuchen und nur die Treffer als `lexword/2`-Fakten speichern. Die Lexika werden nämlich beim Laden um einen Faktor 3 größer. (Nachteil: man muß im Lexikon schematische Einträge in Einzelfälle auflösen.)

11.4.3 Vorberechnungen im Parsergenerator

(i) Für den modularen Aufbau einer größeren Grammatik durch Teilgrammatiken müssen die Vorberechnungen angepasst werden: man sollte (vielleicht) für jede einzelne Grammatik eine Vorberechnung machen und dann z.B. daraus die Vorberechnungen der kombinierten Grammatiken erzeugen: d.h. man sollte die transitive Hülle der Vereinigung mehrerer transitiver Relationen erzeugen, z.B. der Relationen `first/np,first/pp` usw.

(ii) Restriktor: Die Restriktion der Merkmale (zur Berechnung der First-Relation) muß sinnvoller eingeschränkt werden. Zur Zeit werden Satzregeln für definite und interrogative Sätze vorgeschlagen, wenn die Eingabe mit einer definiten Nominalphrase beginnt. Es wäre sinnvoller, wenn die vorgeschlagenen Regeln die Definitheit des Satzes offen ließen.

   Den Restriktor nicht nur bei `basefirst`, sondern auch bei der Iteration (und dazu in `link/2`, wie ursprünglich in `tc/2` vorgesehen) einbauen und mit unterschiedlichen Restriktoren testen.

   Man muß das Nachladen von `wortarten.cat.pl` korrigieren; wenn man darin das `restrict/2`-Prädikat ändert, bleibt die First-Relation unverändert, wenn man vor der Vorberechnung nicht `wortarten.cat.pl` neu geladen hat! (Ich denke, das habe ich korrrigiert: `precomputations/1` löscht alte Einträge von `restrict/2` und die Grammatik `np.grm` enthält eine Deklaration `:- mulitifile restrict/2.`, oder `np.cat.pl`.)

   In *allen* Grammatikdateien (mit Merkmalen) ein `restrict/2` einbauen; das ist bisher nur für `np.grm`, `pp.grm` und `ap(p).grm` gemacht.

(iii) Predictor: Um die Parsezeit zu senken, ist es wichtig, daß der Predictor möglichst wenig Regeln vorschlägt. Auch die eingebaute 1-Vorausschau löst das nicht, sie schlägt noch zu viele unbrauchbare Regeln als möglicherweise brauchbar vor, zum Beispiel:

   (a) es werden Koordinationsregeln vorgeschlagen, auch wenn gar keine Koordinationspartikel im Rest der Eingabe vorkommen,

   (b) es werden *alle* PP-Regeln vorgeschlagen, wenn das nächste Eingabesymbol ein Personalpronomen ist: es könnte mit einer Postpositionen eine Präpositionalphrase gebildet werden - was im Deutschen unmöglich, da ist (`seinetwegen` statt `seiner wegen`),

   Eine bessere Filterung könnte man erreichen, wenn man alle in der lexikalischen Analyse (des Rests) der Eingabe auftretenden lexikalischen Kategorien sammelt und alle Regeln ausfiltert, die dazu nicht passen. Z.B. würde dann beim Lesen eines Personalpronomens keine PP-Regel vorgeschlagen, wenn es keine Postpostion in der Eingabe gibt. Dieser Filter ist jetzt in den `.epl`-Parser (als Option `ein(omitRulesWithUselessLexCat)` in `predict(J)` eingebaut, aber noch nicht gründlich getestet.

(iv) Sollte man die Parsetabellen kompakter mit Regeln statt instantiierten Spezialfällen darstellen, oder wird der Verarbeitungsaufwand dadurch sogar schlechter?

11.4.4 Testumgebung

(i) Effizienzvergleiche zwischen den Parsern und zwischen Varianten der Regeln:

   (a) Bringen optionale Konstituenten bzw. linksfaktorisierte Regeln viel? Lohnt der Aufwand im Vergleich dazu, daß man die Merkmalberechnung vorsichtiger machen muß?

   (b) Der Earley-Pratt-Version ist schneller als die Earley-Doerre-Version, aber die Vorberechnung braucht eine zu große Stacktiefe.

   Vermutlich wäre es ausreichend, wenn man nicht die `leftcorner`-Relation und den Predictor vorberechnete, sondern nur die `first`-Relation, und damit eine 1-Vorausschau in den Predictor einbaut (d.h. `scan` vor `predict` macht und dann beim Voraussagen von Regeln diejenigen wegfiltert, die das nächste Eingabesymbol nicht am Phrasenanfang erzeugen können).

(ii) Erweitere `analysierePhrasen/0`, sodaß es Parserergebnisse aus Beispieldateien mit den Ergebnissen früherer Parser- und Grammatikversionen vergleicht und festhält. Die Analysen der Beispiele jeweils mit den verbesserten(?) Grammatiken und Parsern wiederholen! (Dazu ist der Parser noch zu langsam.)

(iii) Man braucht unbedingt ein Programm `pruefe_grammatik`, mit dem man

(a) alle in der Grammatik verwendeten grammatikschen Kategorien mit den deklarierten grammatischen und den deklarierten Wortkategorien vergleicht,

(b) prüft, ob für alle Wortkategorien und für alle grammatikschen Kategorien Einträge in `restrict/2` vorhanden sind,

(c) gegebenenfalls eine Warnung an den Benutzer ausgibt.

Das sind die Stellen, wo der Grammatikschreiber bisher von Hand für Konsistenz sorgen muß und leicht Fehler macht!

Beachte: die Relation `first/2` hängt von `restrict/2` ab; wenn sie zu klein ist, werden leicht Analysen übersehen. (Angefangen in `wortarten.pl`.)

11.4.5 Grammatikregeln

(i) Nominalphrasen: Die Kategorien `ordinalzahl` und `kardinalzahl` werden bisher nur in der Adjektivgrammatik benutzt, müssen aber vielleicht auch bei den Determinatoren in Nominalphrasen berücksichtigt werden. (?)

(ii) Präpositionalphrasen: die Klassen- und die Formparameter müssen vereinheitlicht werden. Zur Zeit werden z.B. interrogative (und relativierende?) Präpositionalphrasen unterschiedlich dargestellt: `womit : pp([qu], [mit(dat)])`, `mit welchem Gesetz : pp([qu(3, neut, sg)], [mit])`, `mit wem : pp([qu(3, unsp, sg)], [mit])`, dazu noch `pp(qu(_),Form)`, wofür ich das Beispiel vergessen habe (Adverbiale?).

Das Klassenmerkmal sollte neben der Definitheit soviel Information enthalten, wie man zur Überprüfung der syntaktischen Korrektheit der Antwort braucht. Dann müßte man für Adverbialfragen die entsprechende adverbiale Funktion angeben, etwa durch `qu(AdvF)`, und das geht nur bei besserem Lexikon. Sollten dann die Ausdrücke `mit welchem Mittel` und `mit welchen Mitteln` dieselben Klassenparameter haben, oder eher `qu(3,neut,sg),instr` und `qu(3,neut,pl),instr` ? Die adverbiale Funktion bei den freien Präpostionalphrasen wäre zumindest im definiten, interrogativen und relativierenden Fall einheitlich möglich.

(iii) Die Satzgrammatik in Teilgrammatiken zerlegen, z.B. `sfin.grm`, `sinf.grm`, `smod.grm`. Optionale Adverbialphrasen einbauen. Ist nur angefangen ...

(iv) Sätze: Die Beschreibung der Reflexivpronomina und der reflexiven Verben sollte wie in der Vorlesung besprochen nachgezogen werden (vgl. email von E.Torres). Im Verbrahmen wollten wir `dat(refl)` bzw. `akk(refl)` schreiben, und dazu nur Pronomina als entsprechende Komplemente zulassen.

Vgl. die Email auch für die besprochene Beschreibung der Kontrollverben.

(v) Die Datpei `Lexika/wortarten.cat.pl` einlesen und mit dem darin definierten `wortkategorie/1` ein Kontrollprädikat schreiben, das bei der Vorberechnung prüft, ob alle Wortkategorien der Grammatik die Eigenschaft `wortkategorie/1` erfüllen.

Diese Datei sollte vielleicht auch die Deklaration `:- multifile auxiliary_categories/1` enthalten, die z.Zt. in `np.grm` steht, was nur ausreicht, wenn `np.grm` die erste Teilgrammatik ist. Das sollte aber davon abhängen, ob alle (Merkmal-)Grammatiken die `wortarten.cat.pl` brauchen.

Wann muß sie gelesen werden? Wenn der Parser erzeugt wird, also beim Vorberechnen. Die entsprechenden Laufzeitprädikate sollten dann in der erzeugten Datei `.epl` (und ggf. `.ed`) stehen. Komisch: sie soll laut `np.grm` geladen werden, aber es fehlten Teile von `first/2` (wegen `restrict/2`?).

11.4.6 Dokumentation

(i) Den Parser von Sebastian Hick einbauen!

(ii) Parser `earley_doerre_0` beschreiben: nur für geschlossene Kategorien, hat daher kein Matching, sondern nur Unifikation beim Vergleich, ob eine Kante in der Tabelle ist.

(iii) Die Datei `auswahl.pl` muß in der Dokumentation der benutzten `auswahl.hl.pl` angepasst werden!

(iv) Die Überarbeitung der Auswahldatei ist noch nicht fertig, insbesondere ist `grammatiken/0` durch die Sonderfälle noch unleserlich. Bei diesen Fällen, bei denen mehrere Dateien zu laden sind oder der Dateiname nicht aus dem der Originalgrammatik und dem Parsersuffix hervorgeht, sind noch auf dieses Schema zu bringen, indem man eine Hauptdatei mit dem passenden Namen anlegt, die die anderen Dateien lädt.

(v) Sollte `restrict/2` nicht in `auswahldatei.pl` deklariert und im ersten Kapitel beschrieben werden?)

Seminarbeiträge:

(i) Nominalphrasen: (a) Nominalphrasen mit (nicht extrahiertem) Relativsatz, (b) Genitivattribute und ihre Kodierung in den Merkmalen (d.h. weitere Parameter bei `n1` oder neue Kategorie `n2`?) (c) Nominalisierte Verben (und Adjektive) mit Komplementrahmen erlauben.

Wie vermeidet man doppelte Lesarten im Plural, wo die Determinatoren im Lexikon mit Genus `unsp` vermerkt sind? Das Kongruenzprädikat muß mit einem *entweder-oder* statt einem *oder* formuliert werden.

(ii) Infinitive: Der Klassenparameter `s([infzu(Person,Numerus)],Form)` sollte anzeigen, welche Person und Numerus die Reflexivpronomina des Infinitivsatzes haben. Das braucht man beim Einbau des Infinitivsatzes an Kontrollverben.

(iii) Modalverben: Können die nicht einfach als Vollverben mit Komplementrahmen `inf` beschrieben werden?

(iv) Fragesätze: Die Verbeinträge für Verben mit Interrogativkomplement sollten dieses im Komplementrahmen nicht so spezifisch wie durch `conj(ob)` angeben, sondern allgemeiner durch `qu`. Entsprechend sind die Grammatikregeln zu verallgemeinern. Man kann m.E. stets nicht nur Satzfragen, sondern auch Komplementfragen anschließen: `Er fragte, (ob ...) | (wer ...) | (wann ...) | (mit wessen N ...) | wen ?`

(v) Prädikativsätze: Einschränkung auf die entsprechenden Hilfsverben `sein,haben,werden`, teilweise `bekommen`.

(vi) Relativsätze: Regeln beschreiben und einbauen (lassen).

(vii) Modalverben und Infinitive anpassen.

(viii) Reflexive Verben mit eigenen Satzregeln (pronominalisiertes NP-Objekt), oder lieber ein `refl`-Merkmal bei den Nominalphrasen?

(ix) Possesiva sind bisher z.T. Determinatoren, z.T. Adjektive (glaube ich)

(x) Klasseneinteilung bei den PP's vereinfachen?

(xi) Abkürzungslexikon vervollständigen: teilweise sind noch keine Kategorien verbeben worden.

12 Anhang 4: Beispieltext

Der Referenztext in `Testbett/wissenschaftsreligion.txt` ist ein Artikel aus DIE ZEIT, Nr. 27, 2000. Die dazu erstellten Lexika finden sich (unbearbeitet) nach Wortarten gegliedert in `Testbett`, die überprüften und teilweise geänderten Einträge im Verzeichnis `Lexika`.

> Heiliges Erschauern
>
> Über die Heraufkunft einer neuen Wissenschaftsreligion und ihre Gefahren
>
> von Klaus Günther
>
> Am Ende der Weimarer Republik bereitete die sozialdemokratische Reichstagsfraktion einen Gesetzentwurf vor, der für rückfällige Straftäter, so genannte „Gewohnheitsverbrecher", die freiwillige Sterilisation vorsah. Als einige Zeit später die Nazis ein Gesetz über die zwangsweise Sterilisation von Gewohnheitsverbrechern verabschiedeten, wurde dies von der Auslands-SPD vorbehaltlos als fortschrittliche Maßnahme im Kampf gegen das Verbrechen begrüßt. Dass Kriminalität erblich sei, galt als gesicherte wissenschaftliche Erkenntnis.
>
> Trotz vereinzelter Bemühungen ist es bisher nicht gelungen, ein „Kriminalitätsgen" ausfindig zu machen. Heute würde man vermutlich auch nicht zu einer so einschneidenden Maßnahme wie einer Sterilisation schreiten, sondern vielleicht einen genchirurgischen Eingriff versuchen. Indes war die Vermutung, Kriminalität sei erblich, schon damals nicht so sicher bestätigt, wie diejenigen glaubten, die daraus politische Schlussfolgerungen für die Strafgesetzgebung zogen. Doch die Autorität der Naturwissenschaften stand außer Zweifel, und ihre eindrucksvollen Fortschritte, vor allem in der von Darwin begründeten Evolutionstheorie, ließen der Politik scheinbar keine andere Wahl.
>
> Eine noch viel stärkere Überwältigung der Öffentlichkeit durch die Wissenschaft scheint gegenwärtig stattzufinden. Wiederum sind es neue, in rascher Folge gewonnene Erkenntnisse über die Natur des Menschen und Techniken ihrer Veränderung, welche ein heiliges Erschauern hervorrufen. Neben Humangenetik und Hirnforschung, Soziobiologie und Evolutionspsychologie treten die beeindruckenden Leistungen der Computertechnik, der Nanotechnologie und jener Wissenschafts- und Technikzweige, welche beide Entwicklungen miteinander verknüpfen, also Biophysik und -informatik, die Entwicklung intelligenter Software oder die Robotik. Deren Möglichkeiten erscheinen zugleich als Bedrohung und als glückliche Chance.
>
> Wiederum sind es vor allem die auf wissenschaftliche Autorität gestützten Schlussfolgerungen, welche die Zukunft bedrohlich erscheinen lassen. Wenn die Soziobiologie Vergewaltigungen von Frauen durch Männer mit einer genetischen Verhaltensprogrammierung erklärt, ist dann nicht jeder Vergewaltiger unschuldig? Wenn Menschen ihr Erbgut durch technische Eingriffe verändern oder gar im Wege der Klonierung verdoppeln können, muss man sich dann nicht vor den Missbrauchsgefahren fürchten und Kontrollen einführen? Sind solche Regeln nicht eigentlich Züchtungsregeln für Menschen? Wenn ein Psychotiker wie der „Unabomber" die Nanotechnik in die Hände bekommt - droht dann der Welt nicht eine Katastrophe, gegen die sich die Pläne der notorisch Bösen in jedem James-Bond-Film geradezu idyllisch ausnehmen?
>
> 'Wir sind leichtgläubig gegenüber der Autorität der Wissenschaft'
>
> Gewiss wäre es falsch, solche Gefahren herunterzuspielen oder gar zu leugnen, wie umgekehrt die Augen vor den neuen Möglichkeiten zu verschließen. Irritierend sind jedoch die Leichtgläubigkeit und Leichtfertigkeit, mit welchen Wissenschaft und Technik Autorität zugeschrieben werden. Das gilt auch noch für jene radikalen Gegner, die hinter jeder soziobiologischen Forschungshypothese gleich Faschismus wittern und daher die Soziobiologie am liebsten ganz verbieten möchten. Die Kardinaltugend der Wissenschaft ist jedoch die

Skepsis, denn nur der beständige Zweifel motiviert zur Überprüfung scheinbar gesicherter Erkenntnisse, bis sie sich als falsch herausstellen.

Leider wird diese Tugend oftmals auch von der Wissenschaft selbst nicht beachtet, wenn sie euphorisch oder furchtsam von den ungeahnten Möglichkeiten der Zukunft schwärmt. Lässt dieses Staunen, anders als bei Platon, die Kritik verstummen? Die wissenschaftlichen und technischen Fortschritte treffen jedenfalls auf eine auffallende Bereitschaft, die Strapazen der Kritik nicht mehr auf sich zu nehmen oder Bedenken als wissenschafts- und technikfeindlich zu verdammen. „Wissenschaft“, so war kürzlich zu lesen, „ist spannend wie ein Thriller, spannender als viele Romane oder Filme.“ Ja, Wissenschaft kann sehr spannend sein - aber warum soll die Selbstauslegung des Menschen in Literatur und Film weniger spannend sein? Weil Kunst dem Betrachter die Fragwürdigkeit seiner eigenen Selbst- und Weltdeutung offenbart und Gewissheiten relativiert? Die gläubige Bejahung der Wissenschaft räumt dagegen mit Selbstzweifeln auf. Warum soll man sich durch Kunst ins Leiden an der Endlichkeit treiben lassen, wenn man sich doch klonen kann?

Das geht freilich nur, wenn dabei die Wissenschaft selbst sakralisiert und zu einer säkularen Religion erhoben wird. Dann werden einfach ihre kulturellen und politischen Relativierungen abgestreift, und die konstruktivistische Natur der Wissenschaft als eine menschliche Problemlösungspraxis wird unsichtbar gemacht. Ulrich Raulff hat in der ‘FAZ’ angesichts der Berliner Ausstellung ‘Sieben Hügel’ die nötigen Stichworte geliefert. Endlich würden die Wissenschaftsobjekte dieser Ausstellung kontextfrei präsentiert - „zur Freude all derer, die sich an den pädagogischen und sozialhistorischen Ausstellungen satt gesehen haben ... Zugleich erscheint sie als eine ideale Form, um eine Wissenschaftsreligion zu zelebrieren, die endlich die kulturellen Ketten des neunzehnten und die politischen Ideologien des zwanzigsten Jahrhunderts abgeschüttelt hat und im Begriff ist, eine reine Religion zu werden: die stärkste Form der Bindung und Bündelung geistiger Energie.“

Das ist die Zumutung einer Paradoxie. Die Wissenschaft, die vom methodischen Zweifel lebt, die sich nur der Bemühung um Falsifikation verdankt, soll fraglos hingenommen und von jeglichem Zweifel, der ja nur ein törichter wäre, gereinigt werden. Die Vernunft soll ihre Ansprüche aufgeben; die Gläubigen sollen demütig ihr „Amen“ beten. Demgegenüber scheint die kulturelle, politische und sozialhistorische Reflexion der Wissenschaft langweilig. Sie macht aus dem Thriller eine trockene Abhandlung und aus dem Neuen Testament eine historische Untersuchung über das Leben Jesu. So wird im Zeitalter der Globalisierung Wissenschaft zur neuen Religion, die unser Selbstverständnis angreift, um es zu verändern.

Dabei könnte schon eine einfache Überlegung davon überzeugen, dass es die reine Naturwissenschaft ohne sozialhistorischen Kontext nicht gibt: Einer der wichtigsten Paradigmenwechsel in der Evolutionstheorie steht in einem auffälligen Zusammenhang mit sozialhistorischen Veränderungen. Es dürfte nämlich kein Zufall sein, dass die Evolutionstheorie die Erhaltung der Art zum wichtigsten Selektionskriterium zu einer Zeit erhob, als der Nationalismus den Vorrang des Kollektivs vor dem Einzelnen forderte. Und wenn heute demgegenüber behauptet wird, die treibende Kraft der menschlichen Evolution sei allein die Weitergabe der individuellen Gene, so dürfte ein Zusammenhang mit dem gegenwärtig dominierenden Individualismus der Ökonomie nicht ganz fern liegen.

Man denke nur an den Ende des 19. Jahrhunderts ausgetragenen Streit zwischen Virchow und Haeckel über den „demokratischen“ oder „autoritären“ Aufbau der Zelle. An dieser Kontroverse lässt sich nachvollziehen, wie sehr die Naturwissenschaften bis in die Wahl ihrer Metaphern den kulturellen und sozialhistorischen Kontexten verhaftet sind. Auch bieten die Verflechtungen zwischen Naturwissenschaften, technischen Wissenschaften und Ökonomie genügend Anlass, kontextorientiert über die Wissenschaft zu denken. Gewiss löst sich Wissenschaft dabei nicht in Sozialhistorie auf, und der Nachweis eines genetischen Zusammenhangs zwischen sozialen Veränderungen und wissenschaftlicher Erkenntnis berührt deren Wahrheitsgehalt nicht. Aber allein schon der Streit zwischen Experten und

Gutachtern über die politisch relevanten Fragen der Wissenschaft sollte uns vor einer religiösen Einstellung bewahren.

Möglicherweise birgt die Wissenschaftsreligion größere Gefahren als die einzelnen Ergebnisse der Wissenschaft selbst. In der Geschichte sind es nämlich stets die Wissenschaftsgläubigen gewesen, die Popularisierer und großen Vereinfacher, die sich zu leichtfertigen Schlussfolgerungen hinreißen ließen. Das eindrucksvollste und zugleich schrecklichste Beispiel ist die simple Übertragung der Darwinschen Evolutionstheorie auf Ethnien, Gesellschaften und Kulturen am Anfang des 20. Jahrhunderts. Vor allem zwei Probleme werden bei solchen popularisierenden Schlussfolgerungen leicht übersehen: der so genannte „naturalistische Fehlschluss" und der Reduktionismus. Der Philosoph David Hume und mit ihm die Aufklärung hatten auf den logischen Fehler aufmerksam gemacht, den begeht, wer aus einem Sein ein Sollen, aus der Natur verbindliche Verhaltensnormen herleiten will. Ob eine Verhaltensweise oder ein biologischer Vorgang „normal" ist, ist eine Frage der Kultur und nicht der Natur. Oder mit einem Beispiel aus der Soziobiologie: Selbst wenn die Behauptung zutreffen sollte, dass Männer in der Evolution ein Verhaltensprogramm ausdifferenziert haben, das eine Disposition zur Vergewaltigung von Frauen erzeugt, folgt daraus nicht, dass es „normal" ist, wenn Männer Frauen vergewaltigen. Dem voraus geht die Setzung einer Norm, die sich an kulturell vermittelten Kriterien des Nützlichen und Schädlichen, Gerechten und Ungerechten orientiert.

Die Suggestionskraft wissenschaftlicher Entdeckungen verführt oft dazu, mit ihnen alles und jedes zu erklären. Das war mit Darwins Evolutionstheorie nicht anders als mit Freuds Entdeckung des Unbewussten. Wenn die Evolutionspsychologie bestimmte Verhaltensweisen der Menschen auf ihre Funktion im Wettbewerb um die günstigsten Fortpflanzungschancen zurückführt, bedeutet das weder, dass diese Verhaltensweisen ausschließlich damit erklärt werden können, noch, dass sämtliche menschliche Verhaltensweisen allein darin ihre Ursache haben. Es mag ja sein, dass dieser Wettbewerb eine größere Rolle in der Evolution von Kulturen spielt, als bislang angenommen wurde. Der Variantenreichtum menschlicher Kulturäußerungen wird jedoch schlicht verkannt, wenn von ihm behauptet wird, er bestehe letztlich in nichts anderem als verschiedenen Versuchen zur Optimierung von Fortpflanzungschancen.

Vielleicht wurde ein Physiker zur nobelpreiswürdigen Lösung eines Problems der Quantentheorie durch den Prestigegewinn angetrieben, den er sich davon im Kampf der Männer um die Frauen mit den „besten" Genen erhoffte. Doch damit ist weder erklärt, warum dieser Mann die Quantenphysik und nicht die Evolutionspsychologie als Arena für sein Balzverhalten wählte; noch lässt sich entscheiden, ob seine quantentheoretischen Lösungen wahr oder falsch sind. Und alle Versuche einer monokausalen Erklärung scheitern spätestens dann, wenn es um singuläre Ereignisse menschlichen Verhaltens geht. Die Evolutionspsychologie vermag wichtige Einsichten zur Erklärung kriegerischen Verhaltens liefern. Aber damit lässt sich nicht vollständig erklären, warum 1618 der Dreißigjährige Krieg ausbrach.

'Was der Mensch ist, lässt sich nicht von außen bestimmen'

Ein ähnlicher Reduktionismus liegt auch den Befürchtungen über eine Entmachtung des Menschen durch Roboter oder sich selbst reproduzierende Maschinen zugrunde. Es wäre falsch, die Gefahren dieser technologischen Entwicklungen zu leugnen. Aber die Behauptung, Maschinen könnten Menschen ersetzen, setzt voraus, dass man vorher festgelegt hat, wodurch der Mensch bestimmt sei. Gewisse mathematische Fähigkeiten des Menschen lassen sich schon jetzt durch Maschinen ersetzen und sogar bei weitem übertreffen. Aber deshalb ist noch niemand auf den Gedanken gekommen, menschliche Intelligenz auf diese mathematischen Fähigkeiten zu reduzieren, um dann anschließend zu behaupten, der ganze Mensch würde in Zukunft durch Maschinen ersetzt werden.

Was den Menschen ausmacht, lässt sich nicht allein von außen bestimmen. Menschen leben nämlich immer in zwei Welten zugleich - als biologische Wesen in der natürlichen

Welt und als Kulturwesen in einer Welt symbolischer Bedeutungen, die vor allem durch die Sprache repräsentiert und durch Kommunikation vermittelt wird. Zu dieser Welt gehören auch die Selbstdeutungen der Menschen, mit denen sie bestimmen, wer sie sind und wer sie sein wollen. Die Frage, wie sich diese zwei Welten zueinander verhalten, ist eines der hartnäckigen und vielleicht auch unlösbaren Probleme der Philosophie, die gegenwärtig vor allem zwischen Hirnforschern, Psychologen und Neurophilosophen diskutiert werden.

Gewiss lassen sich kulturelle Symbolsysteme ihrerseits wiederum „biologisch erklären" - vielleicht als ein erfolgreiches Mittel der Evolution, durch das menschliche Gehirne sich wechselseitig so beeinflussen, dass sie die höherstufigen Strukturen erwerben, die ihnen ein Überleben in einer hochkomplexen Umwelt ermöglichen. Aber von einer solchen Erklärung führt kein Weg zu dem kulturellen Bedeutungsgehalt selbst. Was die Betroffenen jeweils unter einem bestimmten kulturellen Symbol verstehen, von welchem sie in ihren sozialen Beziehungen Gebrauch machen, erschließt sich durch eine solche Erklärung nicht. Wir brauchen sie auch nicht, wenn wir verstehen wollen, wie Menschen sich selbst und die Welt, in der sie leben, deuten. Ein Mann, der einer Frau seine Liebe erklären will, scheitert, wenn er sich dabei ohne Ironie des soziobiologischen Vokabulars bedient. Einer Frau, die vor der Entscheidung steht, ob sie ihre Schwangerschaft abbrechen will oder nicht, ist nicht damit geholfen, dass ein Verhaltensbiologe sie über die „wahren" Ursachen ihrer Entscheidung aufklärt und ein Hirnforscher nach Untersuchung ihrer Gehirnstrukturen das Verhalten voraussagt. Nun mag Freiheit aus der Perspektive eines naturwissenschaftlichen Beobachters nichts als eine Illusion sein, und es ist müßig, darüber zu spekulieren, ob es so etwas wie einen Anfang ohne Ursache, eine Wirksamkeit aus sich selbst in der Natur geben kann oder nicht. Nach allem, was wir wissen, gibt es sie nicht. Sobald Menschen aber miteinander kommunizieren, einander ein Versprechen geben, eines Fehlverhaltens beschuldigen oder einander verzeihen, können sie nicht umhin, an diese Illusion zu glauben und sich wechselseitig als Personen zu deuten und zu behandeln, die anders hätten handeln können und daher auch fähig und willens sind, einander Rechenschaft über ihr Handeln in der Weise zu geben, als ob es ihr jeweiliges Handeln wäre - und nicht das Resultat von Naturprozessen.

Auch wenn sich die Kultur der Selbst- und Weltdeutung nicht vollständig auf Natur reduzieren lässt, so bleibt es doch zumindest eine offene Frage, wie sich diese Kultur unter dem Einfluss einer Wissenschaftsreligion verändern wird. Zwei gegenläufige Entwicklungen zeichnen sich ab: Auf der einen Seite nehmen die Verfügungsmöglichkeiten des Menschen über seine eigene Natur und die seiner Nachkommen erheblich zu. Zu unserer gegenwärtigen Kultur der Welt- und Selbstdeutung gehört es nun aber, dass wir uns wechselseitig als Personen deuten, die über Wahlmöglichkeiten verfügen und - bei Abwesenheit von Zwang - auch frei sind. Wer sich in dieser Situation für die eine Handlung entscheidet, hätte sich auch für die andere entscheiden, also anders handeln können. Weil wir einander wechselseitig diese Freiheit zuschreiben, können wir uns auch verantwortlich machen.

In dem Maße aber, wie es uns möglich wird, menschliche Gene zu verändern, hat die genetische Ausstattung eines Menschen für uns nicht mehr die Bedeutung eines hinzunehmenden Schicksals. Wenn wir sie verändern können, dann müssen wir auch die Verantwortung dafür übernehmen, 'ob' wir sie verändern oder nicht. Auf dieses Symbolsystem, auf die Deutung des Menschen als freie, verantwortliche Person, beziehen sich auch jene Wissenschaftler, die wie Bill Joy nach einer Verantwortung von Wissenschaft und Technologie rufen. Mit der Ausdehnung unserer Verfügungsmöglichkeiten über die innere und äußere Natur wächst also auch die Zurechnung der Folgen zur individuellen Verantwortung - und dann auch die Haftung für zurechenbare schädliche Folgen - ins Unermessliche.

Es könnte jedoch sein, dass diese Kultur der Verantwortung durch denselben Prozess untergraben wird, der zu der gewaltigen Steigerung unserer Möglichkeiten führt. Es geht nicht nur um das Problem, ob es der Menschheit gelingen wird, die Folgen von Wissenschaft und Technik zu beherrschen. Fraglich ist auch, ob wir unsere heute praktizierte

Kultur der Verantwortung auch künftig noch fortsetzen können, weil wir inzwischen unser Selbstverständnis so verändern, dass wir uns gar nicht mehr als verantwortliche Personen verstehen. Das erscheint vor allem dann fraglich, wenn wir den Predigern der Wissenschaftsreligion folgen und die Forschungsergebnisse umstandslos in unser Selbstverständnis übernehmen: wenn wir zum Beispiel die Behauptung der Evolutionsspsychologie so übernehmen, dass wir uns wechselseitig als Geschlechtswesen deuten, ständig besorgt, ihre Position im Wettbewerb um die besten Gene für die Fortpflanzung zu verbessern. Wenn wir den Forschungen der Soziobiologie in der Weise folgen, dass wir uns ausschließlich als determinierte Wesen verstehen. Schließlich auch, wenn wir mit der Nanotechnologie die Grenze zwischen menschlicher Natur und Technik verwischen oder in immer größerem Umfang menschliche Gene zur Produktion von Ersatzteillagern verwenden.

Jürgen Habermas hat bereits die Frage gestellt, ob ein menschlicher Klon sich selbst noch als eine verantwortliche Person wird verstehen können, wenn er einen Designer für seine genetische Zusammensetzung verantwortlich machen muss. Wie berechtigt diese Frage ist, wird spätestens an der Vorstellung eines sich selbst reproduzierenden Roboters deutlich. Würde es selbstverständlich sein, diesen Roboter für seine „Handlungen" verantwortlich zu machen und vor einem Strafgericht anzuklagen? Oder würde uns das ähnlich merkwürdig erscheinen wie die Strafaktion des persischen Königs Xerxes, der das Meer auspeitschen ließ?

Das Konzept einer verantwortlichen Person gehört zu den zentralen Fundamenten einer demokratischen Verfassung. Schon immer lautete das wichtigste Argument der Demokratiegegner, bestimmte Menschen müssten von der politischen Meinungs- und Willensbildung ausgeschlossen werden, weil sie unfähig seien, verantwortlich zu entscheiden. Doch die Gründung der Gesetzgebung und des Staates auf die Zustimmung gleicher und freier Staatsbürgerinnen und Staatsbürger wird nur dann zu einer hinreichenden Legitimation, wenn diese sich wechselseitig als Personen deuten, die aus Freiheit entscheiden und Verantwortung übernehmen können. Sobald wir mit der Freiheit auch die Verantwortung aus unserer Selbstdeutung streichen, verliert eine demokratische Legitimation des Staates ihren Sinn.

'Werden Menschenrechte bald zu Züchtungsregeln?'

Es ist daher widersprüchlich, wenn Bill Joy angesichts der Bedrohungsszenarien eine Ethik der Wissenschaft fordert. Sein Ruf geht ins Leere, wenn wir uns nicht mehr als verantwortliche Personen verstehen können, wenn Verantwortung und Freiheit nicht mehr in unserer gemeinsam geteilten Selbstdeutung vorkommen. Dann wird es auch sinnlos, Wissenschaft und Technik über Normen steuern zu wollen. Diese Konsequenz droht aber in dem Maße, wie wir wissenschaftliche Ergebnisse so in unsere Selbstdeutung übernehmen, dass wir uns auf genetisch determinierte, den Gesetzen der Soziobiologie unterworfene Wesen reduzieren. Der Vorschlag, Werte und Normen des Humanismus als „Züchtungsregeln" zu verstehen, nach denen Menschen selektiert werden, ist ein Schritt in diese Richtung. Zwischen den Regeln, welche die Bildung eines autonomen Gewissens ermöglichen, und Regeln für die Auswahl reproduktionswürdiger Gene wird dann kein Unterschied gemacht.

Wenn wir anfangen, Menschenrechte als Züchtungsregeln zu verstehen, haben wir sie schon abgeschafft. Es geht nicht darum, ob dieser oder jener genchirurgische Eingriff zulässig ist. Die Gefahren beginnen dort, wo wir unser Selbstverständnis ausschließlich an naturwissenschaftlichen Erklärungen und technischen Eingriffsmöglichkeiten ausrichten. Was sollten denn Freiheit und Verantwortung aus naturwissenschaftlicher Sicht bedeuten?

Es wird schwierig werden, die Kultur der Verantwortung gegen das neue Priestertum der Wissenschaftsreligion aufrechtzuerhalten. Wer sich im Internet einmal die Seiten angeschaut hat, auf denen weibliche Eizellen und männliche Samenzellen zur Versteigerung angeboten werden, lässt sich vielleicht noch verstören von den Werbetexten, die den evolutionären Vorteil schöner Körper anpreisen. Die Bilder der Menschen, die sich dafür hergeben, zeigen die glatten, durchtrainierten Körper der Werbewelt, denen vor allem eines gemeinsam ist: ihre Verwechselbarkeit und Austauschbarkeit. Der Mensch, der sich selbst auf seine Funktion für die Evolution reduziert, wird zu einem schönen Niemand. Dass es allerdings immer noch viele Menschen gibt, die so nicht leben wollen, kann man an dem großen Zulauf sehen, den Kunstgalerien wie die Tate Modern in London derzeit haben. Ja, auch Kunst kann man evolutionsbiologisch erklären - aber das interessiert die Kunstbetrachter in dem Augenblick nicht, wo sie an einem Kunstobjekt erfahren, dass sie deutende und verstehende Personen sind. In London kann man auch die Bilder des englischen Malers Lucien Freud sehen, die zumeist nackte menschliche Körper darstellen, in welche sich Spuren einer individuellen Lebensgeschichte eingezeichnet haben und die allen Maßstäben der Werbewelt widerstreiten. In ihrer bloß raumfüllenden Anwesenheit, in der sie nichts aussagen als sich selbst, so wie sie als gelebte Körper in diesem Augenblick an diesem zufälligen Ort da sind - daraus beziehen sie ihre Würde. Es ist die Würde von Menschen, deren Körper endlich ist und zugleich eine einmalige, unwiederholbare und deshalb unendliche Lebensgeschichte ausdrückt. Für diese Unendlichkeit geht uns vielleicht der Sinn verloren, je mehr wir uns mithilfe von Wissenschaft und Technik der körperlichen Unendlichkeit annähern. Nach ihr sehnten sich alle Untoten der Literatur, die durch eine göttliche Strafe dazu verdammt waren, nicht sterben zu können.

Klaus Günther ist Professor für Rechtstheorie und Strafrecht an der Universität Frankfurt am Main

DIE ZEIT, 27/2000

13 Vorlesungs- und Seminarplan SS 2002

Bestandsaufnahme der bisherigen Grammatik

(i) verbindliche Kategorien und ihre Merkmale erläutern

(ii) Regelformat mit Besonderheiten erklären
(Problem: Merkmale bei optionalen Konstituenten)

(iii) Anpassung der Beschreibung und Programmierung der Teilgrammatiken

(iv) Anpassung der Lexika für die Phrasentransformationen

Erweiterung der bisherigen Grammatik

(i) Kategorien mit fehlender Konstituente
Literatur: Müller/HPSG

(ii) Extraktion und Verschiebung von indirekten Konstituenten
Relativsatzextraktion, PP-Objekt-Extraktion

(iii) Flexionsprogramme auf der Wortebene (machen wir als Übung)

Seminarvorträge

(i) Programmieren und Erklären von Transformationen komplexer Ausdrücke.

Der eingegebene Ausdruck wird mit dem vorhandenen (bzw. angepaßten) Parser analysiert; der ausgegebene Syntaxbaum soll dann durch die Transformationsprogramme (rekursiv nach dem Aufbau der Eingabephrase) in den Syntaxbaum der transformierten Phrase umgerechnet werden.

(a) Kasusflexion komplexer Nominalphrasen:
Um mit einem Programm Nominalphrasen in einen vorgegebenen Kasus beugen zu können, muß man für jede NP-Grammatikregel (und den Regeln für eingebettet Konstituenten) eine Transformationsregel schreiben, die (nach Analyse der Eingabe) entsprechend der Konstruktion der Nominalphrase die transformierte Phrase konstruiert. (Siehe das Beispiel einer einfachen Nominalphrase unten.)

(b) Tempus- und Modusflexion einfacher Sätze:
Das geht analog zur Kasusflexion bei Nominalphrasen. Bei der Tempusflexion treten neue Probleme auf, da man beim Übergang vom Präsens zum Perfekt das Hilfsverb einbauen muß:

i. Änderungen der Wortstellung
ii. Änderungen der Konstituentenanzahl

Er stellt die Frage, ob S. $\mapsto$ Er hat die Frage gestellt, ob S.

(c) Basissätze in adjektivisch modifizierte Nominalphrasen umwandeln (und zurück):

i. Prädikativsatz in adjektivisch modifizierte Nominalphrase umrechnen
Paul ist manchmal etwas faul. $\mapsto$ der manchmal etwas faule Paul.

ii. Basissatz mit Vollverb in adjektivisch modifizierte Nominalphrase umrechnen
Maria liebt den Paul sehr. $\mapsto$ die den Paul sehr liebende Maria.
Paul wird von Maria sehr geliebt. $\mapsto$ der von Maria sehr geliebte Paul.

(d) Basissatz in relativierte Nominalphrase umrechnen (und zurück)

Der persische König Xerxes ließ das Meer auspeitschen. $\mapsto$ (a) der persische König Xerxes, der das Meer auspeitschen ließ, (b) das Meer, das der persische König Xerxes auspeitschen ließ,

(e) Nominalisierung von Verben:

ein Problem der Quantentheorie lösen $\mapsto$ (a) das Lösen eines Problems der Quantentheorie, (b) die Lösung eines Problems der Quantentheorie (durch jemanden)

Hierfür sind Lexikoneinträge für nominalisierte Verben des Beispieltexts und dazu passende Grammatikregeln für Nomina mit Argumenten erst noch zu schreiben. (Es gibt grobe Lexikoneinträge zu nominalisierten Verben.) Komplizierte Beispiele als die obigen wären etwa:

Brutus ermordet Cäsar. $\mapsto$ (a) die Ermordung Cäsars durch Brutus, (b) der Mord des Brutus an Cäsar, (c) Brutus' Ermordung Cäsars, (d) (des) Brutus' Mord an Cäsar, (e) der von Brutus verübte Mord an Cäsar

Brutus plant, den Cäsar zu ermorden $\mapsto$ (a) Brutus' Plan, den Cäsar zu ermorden, (b) der Plan des Brutus, den Cäsar zu ermorden, (c) des Brutus' Plan zum Mord an Cäsar Einfachheitshalber sollen noch keine Nomenkomposita (wie Mordplan) behandelt werden, also noch nicht Transformationen wie

der Mensch hat Möglichkeiten, über seine eigene Natur zu verfügen $\mapsto$ die Verfügungsmöglichkeiten des Menschen über seine eigene Natur

Ohne Extraktion sollten die Aufgaben (i), (ii) und (iii) machbar sein. Für (iii) (a) muß noch die Adjektivgrammatik und für (b) die Satzgrammatik weiter überarbeitet werden. Für (iv) sollten zuerst die Regeln der Relativsatzgrammatik mit den Regeln der Sätze in Verbletztstellung vereinheitlicht werden.

(ii) Einbau von Extraktion und Verschiebung von Subkonstitutenten.

(a) Konstituenten mit Liste fehlender Subkonstituenten (im *slash*-Merkmal) und Propagierung dieses Merkmals zum Elternknoten im Baum,

Abgleich der *slash*-Merkmals beim kleinsten Vorgängerknoten, der auch die verschobenen Konstituente dominiert.

(b) Beispiel: Rechtsextraktion des Relativsat zes hinter das finite Verb.

Alle hatten (das Buch $(-)_i^{RelS}$) gelesen, (das zum Studium empfohlen worden war)$_i$.

(c) Beispiel: Linksextraktion des PP-Objekts ins Vorfeld.

(Über Goethe)$_i$ hat Emil (viele Bücher $(-)_i^{PP}$) gelesen.

Wie die Extraktion technisch behandelt werden kann, ist in manchen Büchern zur HPSG-Grammatik erklärt, z.B. in dem von Pollard/Sag „Head-Driven Phrase Structure Grammar" oder in dem von Gazdar e.a. „Generalized Phrase Structure Grammar", oder in dem von Stefan Müller „Deutsche Syntax deklarativ".

Beispiel 13.1 der Kasusflexion einer einfachen Nominalphrase. Zum besseren Testen schreiben wir ein Programm `beuge(+Kasus)`, das den Benutzer zur Eingabe einer Nominalphrase auffordert. Diese wird durch den Parser analysiert und die gefundenen Analysen durch `beuge_in_kasus(+Kasus,+Analysen,-Transformationen)` in die umgeformten Ausdrücke umgerechnet:

⟨*Testbett/transformiere.pl*⟩≡

```
beuge(Kasus) :-
        write('Eingabe einer Nominalphrase (mit <Punkt><Return> beenden):'),nl,
        read_sentence(user,EingabeNP,[]),
        stringToAtoms(EingabeNP,WortfolgeSz),
        entferneSatzzeichen(WortfolgeSz,Wortfolge),
```

```
        lexAnalyse(Wortfolge),
        (ein(pruefeLexWord) -> erkenneWortarten ; true),
        (gescheitert(lexAnalyse)
        -> (nl, write(user,'Keine syntaktische Analyse durchgeführt.'))
        ; (baeume(Wortfolge,Trees)
          -> (beuge_in_kasus(Kasus,Trees,TreesTr)
             -> nl, write(user,'Analyse(n) der Eingabe:'),
                 nl, showASCIItrees(Trees), nl,
                 nl, write(user,'Transformation(en) der Eingabe:'),
                 nl, showASCIItrees(TreesTr), nl
             ; (nl, write(user,'Keine Transformation der Eingabe gefunden.')))
          ; (nl, write(user,'Keine syntaktische Analyse der Eingabe gefunden.'))
          )
        ).

beuge_in_kasus(Kasus,[Baum|Baeume],[BaumTr|BaeumeTr]) :-
        (Baum = [np(Art,[Num,Kas])|Teile]
        -> (transformiere(Kas,Kasus,Baum,BaumTr) -> true ; BaumTr = [])
        ; BaumTr = Baum),
        beuge_in_kasus(Kasus,Baeume,BaeumeTr).
beuge_in_kasus(Kasus,[],[]).
```

Der Parser liefert für die Nominalphrase `der Mensch` folgende Analyse:

⟨*beispiel*⟩≡

```
87 ?- parse. der Mensch.

1. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
         der
      n1([mask], [-, -, sg, nom])
         n([mask], [sg, nom])
            Mensch
```

Um diese Eingabe in die Ausgabe `den Menschen` transformieren zu können, muß man für die benutzte NP-Grammatikregel

⟨*beispiel*⟩+≡

```
np([Def,3,Gen],[Num,Kas]) -->
        det([Def,AdeklD],[GenD,Num,Kas]),
        n1([Gen],[Komp,Adekl,Num,Kas])
        :-
        kongruenzGenusNP(Gen,GenD),
        kongruenzAdeklNP(Adekl,AdeklD).
```

eine Transformationsregel schreiben, und ebenso für verwendeten die lexikalischen Regeln. Das sind in diesem Falle und für Pronomina folgende Transformationen:

⟨*Testbett/transformiere.pl*⟩+≡

```
transformiere(Kas1,Kas2, [np(Art,[Num,Kas1]),
                          [det(ArtDet,[GenDet,Num,Kas1]),Det],   % i.a. GenDet =/= Gen !
                          [n1([Gen],[-,-,Num,Kas1]),N]],
                         [np(Art,[Num,Kas2]),DetTr,NTr]) :-
        !,transformiere(Kas1,Kas2,[det(ArtDet,[GenDet,Num,Kas1]),Det], DetTr),
        transformiere(Kas1,Kas2,[n1([Gen],[-,-,Num,Kas1]),N], NTr).
```

```
transformiere(Kas1,Kas2, [np(Art,[Num,Kas1]),
                          [pron(ArtPron,[Gen,Num,Kas1,Pers]),W]],
                         [np(Art,[Num,Kas2]),WTr]) :-
        !,flektiere([pron(ArtPron,[Gen,Num,Kas1,Pers]),W],[Gen,Num,Kas2,Pers],WTr).

transformiere(Kas1,Kas2,[det(Art,[Gen,Num,Kas1]),Det1],Variante) :-
        flektiere([det(Art,[Gen,Num,Kas1]),Det1],[Gen,Num,Kas2],Variante).

transformiere(Kas1,Kas2,[n1(Art,[-,-,Num,Kas1]),[n(Art,[Num,Kas1]),N]],
                        [n1(Art,[-,-,Num,Kas2]),NTr]) :-
        flektiere([n(Art,[Num,Kas1]),N],[Num,Kas2], NTr).
```

Ein Flexionsprogramm für Wörter erhalten wir über das Vollformenlexikon, wenn man in den Lexikoneinträgen die Stammformen vermerkt:

⟨*Testbett/transformiere.pl*⟩+≡
```
flektiere([Kat1,Vollform1], Form2, [Kat2,Vollform2]) :-
        word(Vollform1, Kat1, Stamm),
        Kat1 =.. [Kat,Art,Form1],
        Kat2 =.. [Kat,Art,Form2],
        word(Vollform2, Kat2, Stamm).
```

Die bisherigen Lexikoneinträge der Form `word(Vollform,Kategorie)` enthalten noch keine Stammform. Neue Lexika mit Einträgen der Form `word(Vollform,Kategorie,Stammform)` müssen wir für die Beispieltexte noch erstellen. In `Testbett/wissrel.wrd` steht eine vorläufige Version des neuen Lexikons für den Text `Testbett/wissenschaftsreligion.txt`, ohne die Anpassungen durch Ergänzung von Komplementrahmen.

⟨*Testbett/transformiere.pl*⟩+≡
```
:- multifile word/3.
:- ensure_loaded('Testbett/wissrel.wrd').
```

Auch von allen nicht aus dem CISLEX übernommenen Lexikoneinträgen muß man Varianten machen, die die Stammform enthalten. Für die bestimmten Artikel sind das die folgenden:

⟨*Testbett/transformiere.pl*⟩+≡
```
word(der, det([def,sw], [unsp, pl, gen]), d).
word(der, det([def,sw], [mask, sg, nom]), d).   % auch: rel
word(der, det([def,sw], [fem, sg, gen]), d).
word(der, det([def,sw], [fem, sg, dat]), d).    % auch: rel
word(des, det([def,sw], [mask, sg, gen]), d).
word(des, det([def,sw], [neut, sg, gen]), d).
word(dem, det([def,sw], [mask, sg, dat]), d).
word(dem, det([def,sw], [neut, sg, dat]), d).
word(den, det([def,sw], [unsp, pl, dat]), d).
word(den, det([def,sw], [mask, sg, akk]), d).
word(die, det([def,sw], [unsp, pl, nom]), d).
word(die, det([def,sw], [unsp, pl, akk]), d).
word(die, det([def,sw], [fem, sg, akk]), d).
word(die, det([def,sw], [fem, sg, nom]), d).
word(das, det([def,sw], [neut, sg, nom]), d).
word(das, det([def,sw], [neut, sg, akk]), d).
```

Statt der bisherigen Lexikoneinträge leiten wir das weiterhin im Parser verwendete `word/2` vom neuen Lexikon ab:

⟨*Testbett/transformiere.pl*⟩+≡

```
word(W,Cat) :-
        word(W,Cat,Stamm).
```

Nach Laden des Parsers `app.grm.ed.pl` und der Datei `Testbett/flektiere.pl` kann man dann Beispiele einfacher Nominalphrasen nach dem Kasus beugen:

⟨*beispiel*⟩+≡

```
66 ?- beuge(dat).
Eingabe einer Nominalphrase (mit <Punkt><Return> beenden):

Analyse(n) der Eingabe:

1. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
         der
      n1([mask], [-, -, sg, nom])
         n([mask], [sg, nom])
            Mensch

Transformation(en) der Eingabe:

1. np([def, 3, mask], [sg, dat])
      det([def, sw], [mask, sg, dat])
         dem
      n1([mask], [-, -, sg, dat])
         n([mask], [sg, dat])
            Menschen
```