Seminar Satzgrammatik
WS 2003/2004

Tokenizer, Lexer und Parser
(Vorversion)

Hans Leiß, CIS

29.10.2003

Zusammenfassung

Dieses Skriptum dokumentiert die Programme zur Tokenisierung, lexikalischen Analyse und syntaktischen Analyse von Eingabesätzen, die im Seminar verwendet werden. (Es ist eine Weiterentwicklung eines Teils des Skriptums zur Vorlesung und Seminar „Grammatikentwicklung" aus dem Wintersemester 2001/2002.)

Die Benutzung der Programme geschieht auf eigenes Risiko. Es wird keine Haftung übernommen. Die Lexika und Kategorienkonventionen zur Satzgrammatik werden in einem getrennten Skriptum beschrieben.

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(first_base(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 :-
        consult('Lexika/word.pl'),
        (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 angeht; 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)', first_base(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('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')),
        nl,write('First-Relation: '),
        (setof([B,H],(first_base(B),tc(H)),[BH]) -> write(BH) ; write('keine Wahl getroffen')),
        nl,write('Predictor: '),
        (setof(P,predictor(P),Predictor) -> write(Predictor) ; 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* den 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 8.

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

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 lexikalischen Kategorien zugeordnet (vgl. Aufgabe 4.1).

(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),
        !,eintraege_abk(String,W,N).
eintraege_abk(String,W,N) :-
        abkuerzung(String,Kategorie),
        eintrag(W,Kategorie,N),
        fail.
eintraege_abk(_String,_W,_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.) Da einige Abkürzungen (wie `Fr.`) verschiedene Lesarten haben, müssen im Allgemeinen mehrere Kategorien ermittelt werden.

(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).
```

(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).
```

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

[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!

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(',').
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.

Aufgabe 4.1 Zahlen sind bisher nur als Ziffernfolgen berücksichtigt. Die Mischformen wie `der 27ste`, `27mal`, `27fach` und die natürlichsprachlichen *Zahlwörter* wie `siebenundzwanzig` und `-ste`, `-mal`, `-fach`, aber auch `siebenundzwanzigstens`, werden als Token isoliert, aber von der lexikalischen Analyse nicht erkannt. Sie können nicht in einem Lexikon gesammelt werden (wenn man keine Beschränkung an die Größe der Zahl festlegen will). Erweitere die lexikalische Analyse um eine Behandlung dieser Mischformen und Zahlwörter.

Hinweis: Zahlwörter wie `sieben` bzw. `7te` sind als als Adjektive (im Positiv bzw. Superlativ, und – außer für `ein,eine,ein`– im Plural und daher mit Genus `unsp`), solche wie `siebtens` als Adverbien zu behandeln. Beachte, daß `der 7te Versuch` und `ein 7ter Versuch` eine Adjektivdeklination zeigen, die bei der Zifferndarstellung durch `der 7. Versuch` verschwindet. Andererseits ist `7.` ambig, da es als Adjektiv im Superlativ, aber auch als Adverb `siebtens` verwendet wird. Außerdem bleiben die Nominalisierungen der Bruchzahlen, also `ein Drittel`, `vier Fünftel`, `die Hälfte`, zu behandeln.

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 *Wortformen* oder kurz *Wörter* `<Wort>` werden durch Prolog-Atome dargestellt: Wörter mit klein geschriebenem Anfangsbuchstaben sind schon Prolog-Atome, z.B. `und`; Wörter mit groß geschriebenem Anfangsbuchstaben sind in einfache Anführungszeichen zu setzen, z.B. oder `'Hund'`.

Als lexikalische Kategorien `<Kategorie>` sind beliebige (geschlossene) Prolog-*Terme* zugelassen, z.B. `conj` oder `n(nom,sg)` in

⟨*Beispiel*⟩≡
```
word(oder,conj).
word('Hund',n(nom,sg)).
```

Die Lexikoneinträge können auch *schematisch* angegeben werden, z.B.

⟨*Beispiel*⟩+≡
```
word('Hund',n(Kasus,sg)) :- Kasus = nom ; Kasus = akk.
```

Zwei kleine Beispiellexika sind im kommenden Abschnitt unter Beispiel 5.1 und 5.3 angegeben.

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, sodaß die Korrektur dieser Information einfach ist.

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

Siehe dazu den Abschnitt ?? zur Erstellung des Vollformenlexikons für die Beispielgrammatik des Deutschen.

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.

Eine Grammatik besteht aus (lexikalischen und grammatischen) *Kategorien* und einer Menge von *Grammatikregeln*. Unter den Kategorien werden *Haupt-* oder *Start-Kategorien* festgelegt: nur syntaktische Strukturen für Ausdrücke dieser Kategorien werden gesucht.

5.1 Format der Grammatiken

Wir verwenden als Format der Grammatiken eine eingeschränkte Form der *Definite Clause Grammar* von Prolog. Es ist nützlich, bestimmte einfachere Grammatikregeln besonders zu benennen, da für diese die Parser ursprünglich entwickelt wurden und z.T. für die erweiterte Version nicht unbedingt funktionieren.

Ein *Grundterm* von Prolog ist ein Term, in dem keine Variable vorkommt. Einen Prolog-Term der Form `P(T1,...,Tk)` (oder `P` bei $k = 0$) nennen wir *einfach*, wenn das Funktionszeichen `P` weder `[]`, noch die Listenerweiterung `|`, die Konjunkion `,`, die Disjunktion `;`, das Fragezeichen `?` oder die Mengenklammer `{.}` ist.

5.1.1 Deklarationen und Hilfsprädikate

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

Jede Grammatik enthält zuerst eine Deklaration, die den Regelpfeil `--->`, der in den folgenden Programmstücken leider fälschlich als `-->` gedruckt wird, zu einem Infix-Operator von Prolog macht:

⟨*Operatordeklaration*⟩≡

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

In jeder Grammatik wird durch Fakten oder Regeln der Form

⟨*Startymbole*⟩≡

```
start_symbol(A).
start_symbol(A) :- bedingung(A).
```

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

Außerdem muß durch

⟨*Hilfsprädikate*⟩≡

```
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 (vgl. Abschnitt 5.3.3) an den jeweiligen Parser übernommen.

Bei größeren Grammatiken sollte man unter diesen Hilfsprädikaten zwei Prädikate `wortkategorie/1` und `kategorie/1` definieren. Dann wird nach Einschalten der Option `pruefeKategorien` während der Übersetzung der Grammatik geprüft, ob alle in der Grammatik verwendeten Kategorien diesen Deklarationen entsprechen.

5.1.2 Kontextfreie Grammatiken (CFG)

Eine *Kategorie* ist ein einfacher Grundterm. Eine *kontextfreie Grammatikregel* hat die Form

⟨*kontextfreie Grammatikregel*⟩≡
```
A --> B,...,C.
```

(Eigentlich: `A ---> B,...,C.`, was leider inkorrekt gedruckt wird!)

Hierin sind `A`,`B`,..., `C` Kategorien. Gelegentlich erlauben wir unter `B,...,C` auch Listen `[T|Ts]` und `[]`, mit denen *konkrete Wortfolgen* als Konstituentenkategorien erlaubt werden. (Warnung: Die Behandlung nicht-leerer Listen `[T|Ts]` ist nicht vollständig implementiert, da normalerweise das Lexikon abgetrennt ist.)

Eine *Löschregel* ist eine Regel der Form `A --> []`, bei der rechts die leere Liste steht. Eine *Kettenregel* ist eine Regel der Form `A --> B`, bei der `B` eine grammatische oder lexikalische Kategorie ist.

Eine Kategorie `A` ist eine *grammatische Kategorie*, wenn sie in einer Regel auf der linken Seite von `-->` auftritt, in der rechts nicht nur `[]` steht. Eine Kategorie `B` außer `[]`, die höchstens in Löschregeln auf den linken Seiten von `-->` auftritt, ist eine *lexikalische Kategorie* oder *Wortart*.

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

Beispiel 5.1 Wir betrachten den allereinfachsten Fall, wo die Kategorien Prolog-*Atome* sind. Die Grammatik enthält zuerst die Deklarationen für `-->`, für die Hilfsprädikate und die Startsymbole.

⟨*Grammatiken/cfg.1.grm*⟩≡
```
% Deklarationen:
:- op(1100,xfx,'-->').

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

Dann folgt die Definition der Hilfsprädikate und die Angabe der Regeln, d.h. die Definition von `-->`:

⟨*Grammatiken/cfg.1.grm*⟩+≡
```
% Hilfsprädikate:

start_symbol(s).
start_symbol(np).

% Regeln:

s --> np, vp.

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

vp --> v.
vp --> v, advp, np.

advp --> adv.
advp --> pp.

pp --> p, np.
```

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

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

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

Ein zu dieser Grammatik passendes, d.h. nur diese lexikalischen Kategorien verwendendes, Lexikon hat ebenfalls keine Aufgliederung der Kategorien nach morphologischen oder anderen Kriterien:

⟨*Lexika/cfg.1.wrd*⟩≡

```
word(der,det).                word(arme,a).
word(die,det).
word(ein,det).                word(in,p).
                              word(für,p).
word('Lied',n).
word('Hamster',n).            word(sehr,grad).
word('Hamsterin',n).          word(gerne,adv).
word('Grube',n).
                              word(und,conj).
word(pfeift,v).               word(oder,conj).
```

Damit kann man z.B. die Eingaben `der arme Hamster pfeift` und `ein Hamster pfeift ein Lied` analysieren.

5.1.3 Erweitert kontextfreie Grammatikregeln (EBNFG)

Eine *erweitert kontextfreie Regel* ist eine Regel `A --> B`, wobei `A` eine grammatische Kategorie einer kontextfreien Grammatik ist und `B` aus grammatischen Kategorien mit Konjunktion `(T,S)`, Disjunktion `(T;S)` und Option `?(T)` aufgebaut sein darf. Wir unterscheiden gelegentlich die *einfachen Kategorien* `P(T1,...,Tk)` von den *zusammengesetzten Kategorien* `(T,S)`, `(T;S)`, und `?T`.

Eine Grammatik mit erweitert kontextfreien Regeln ist eine *Grammatik in Extended-Backus-Naur-Form*, kurz: *EBNF-Grammatik*, außer daß wir keine *Iteration* `*(T)` erlaubt haben.

Durch Regeln der Form `A --> B,(C;D;E),F.` können mehrere Grammatikregeln, hier `A --> B,C,F.`, `A --> B,D,F` und `A --> B,E,F.`, durch *Linksfaktorisierung* eines gemeinsamen Präfixes `B` und *Rechtsfaktorisierung* eines gemeinsamen Suffixes `F` mit *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. (Bei optionalen Kategorien `?C` wird im Folgenden meistens angenommen, daß `C` eine einfache Kategorie ist.)

Die zusammengesetzten Kategorien sind aus zwei Gründen nützlich: erstens kann man mit ihnen mehrere Regeln zu einer zusammenfassen, und zweitens kann man sie beim Parsen auf eine besondere Weise behandeln und dadurch zu einer Beschleunigung der Analyse kommen.

Dabei wird man i.a. die zusammengesetzten Kategorien *nicht* im Ergebnis der Analyse angeben, sondern nur die jeweils verwendete einfachen Teilkategorie. Man kann die zusammengesetzten Kategorien daher als *implizite Kategorien* oder *Abkürzungen* auffassen.

Beispiel 5.2 Wir variieren die Grammatik aus Beispiel 5.1 zu einer erweitert kontextfreie Grammatik, indem wir einige Regeln durch Faktorisierung zusammenfassen:

⟨*Grammatiken/ebnfg.1.grm*⟩≡

```
% Deklarationen:

:- op(1100,xfx,'-->').
auxiliary_predicates([start_symbol/1]).

% Hilfsprädikate:

start_symbol(s).
start_symbol(np).

% Regeln:

s --> np, vp.

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

vp --> v, ?((?(advp), np)).

advp --> (adv ; pp).

pp --> p, np.

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

In der ersten `np`-Regel ist `ap` als optionale Konstituente angegeben. Da die Löschregel `a --> [].` fehlt, sind nur noch die Adjektivphrasen löschbar, die der Nomenmodifikation dienen; bei der Konjunktion von Adjektivphrasen dürfen beide Teilphrasen nicht gelöscht werden. Bei der `vp`-Regel haben wir zwei Regeln durch Faktorisierung zusammengefaßt, ebenso bei der `advp`-Regel. (Hieran noch die Parser testen! Die Deklarationen `wortkategorie/1` und `kategorie/1` sollten nur für die Parser `.epl` und `.glc` nötig sein.)

5.1.4 Schematische kontextfreie Grammatikregeln (sCFG)

Eine *schematische Kategorie* ist ein einfacher Prolog-Term, der keine Variable ist (aber Variable als echte Teilterme enthalten kann). Eine *schematische* oder *bedingte kontextfreie Grammatikregel* ist eine Prolog-Klausel der Form

⟨*schematische kontextfreie Grammatikregel*⟩≡

```
(A --> B1,...,Bk) :- C.
```

Hierin ist die *Bedingung* `C` ein Prolog-Term, und die *Grammatikregel* `(A --> B1,...,Bk)` besteht aus einer schematischen Kategorie `A` und einer *rechten Regelseite* `B1,...,Bk` aus schematischen Kategorien `Bi`.

Typischerweise werden in `C` die Wertebereiche der in `A,B1,...,Bk` verwendeten Variablen, z.B. Kasus und Numerus, eingeschränkt, oder Beziehungen zwischen Eigenschaften der Kategorien `B1,...,Bk` formuliert, d.h. Kongruenz- und Rektionsbeziehungen.

Beispiel 5.3 Als Kategorien verwenden wir zweistellige Prolog-Terme `Kategorie(Unterklasse,Form)`, wobei sowohl die Unterklasse als auch die Form durch eine Liste von Merkmal(wert)en dargestellt werden.

⟨*Grammatiken/cfg.2.grm*⟩≡
  ⟨*cfg.2.deklarationen*⟩
  ⟨*cfg.2.grammatikregeln*⟩

Zuerst getrennt die Deklarationen, die in einem anderen Beispiel wieder verwendet werden:

⟨*cfg.2.deklarationen*⟩≡

```
% Deklarationen:

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

% Hilfsprädikate:

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

genus(mask).    numerus(sg).    kasus(nom).    person(1).
genus(fem).     numerus(pl).    kasus(gen).    person(2).
genus(neut).                    kasus(dat).    person(3).
                                kasus(akk).
```

In den folgenden schematischen Grammatikregeln 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. Die Formen eines transitiven Verbs in der dritten Person im Singular haben daher die Kategorie `v([[nom,akk]],[sg,3])`:

⟨*cfg.2.grammatikregeln*⟩≡

```
% Grammatikregeln:

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

% Kongruenz zwischen Konstituenten wird durch gleiche Merkmalvariable
% (Num,Kas) dargestellt, Rektion durch Formmerkmale, die von Artmerkmalen
% der regierenden Konstituente abhaengen (Gen):

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

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).

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

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

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/cfg.2.wrd*⟩≡

```
word(der,det([mask,sg,nom])).        word(arme,a([mask,sg,nom])).
word(der,det([fem,sg,dat])).         word(frohe,a([mask,sg,nom])).
word(der,det([mask,pl,gen])).        word(pfeift,v([[nom]],[sg,3])).
word(ein,det([mask,sg,nom])).        word(pfeift,v([[nom,akk]],[sg,3])).
word(ein,det([neut,sg,akk])).
                                     word(vor,p([[dat],lok])).
word('Lied',n([neut],[sg,akk])).     word(in,p([[dat],lok])).
word('Hamster',n([mask],[sg,nom])).
word('Hamster',n([mask],[sg,akk])).  word(sehr,grad).
word('Hamster',n([mask],[pl,gen])).  word(und,conj).
word('Grube',n([fem],[sg,dat])).     word(oder,conj).
```

Als Bedingungen in den Regeln werden hier nur die Bereiche der Merkmalvariablen angegeben; Kongruenz- und Rektionsbedingungen behandeln wir später.

Bemerkung 5.4 Normalerweise schreibt man einen Parser so, daß er mit Grammatikregeln, nicht mit Regelschemata arbeitet. Durchsucht der Parser die Grammatik nach einer *Regel*, so entspricht das im Prolog-Code des Parsers einem Ziel `(A --> B)`, wobei `A` und `B` schon teilweise instantiierte Terme sein können; Prolog findet die beweisbaren Instanzen `A'--> B'` hiervon durch Auswerten von Bedingungen passender Regelschemata.

Soll dagegen der Parser die Grammatik nach einem *Regelschema* durchsuchen, so entspricht das im Prolog-Code einem Ziel `clause((A --> B),C)`, wobei wieder `A`, `B` und `C` teilweise instantiiert sein können; Prolog findet die passenden Instanzen `A'--> B' :- C'` hierzu unter den Regelschemata der Grammatik, wobei die Bedingungen nicht ausgewertet werden.[6]

Durch Auswerten der Bedingungen erhält man de facto eine große kontextfreie Grammatik; die Parser werden dann viel zu langsam – daher sind in den Grammatiken später entsprechende Zeilen auskommentiert, so daß es weniger Regeln gibt, diese aber freie Variable enthalten. [7]

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.

[6] Dieser Unterschied ist für das Schreiben der Grammatiken, Parser und die Analysegeschwindigkeit *sehr wichtig*. In Beispiel 5.7 vergleichen wir den Unterschied zwischen der Suche nach Regeln und nach Regelschemata mit `dcg.2.grm`.

[7] Zumindest werden wegen der Bereichsprädikate bei den `.epl` und `.glp`-Parsern die vorberechneten Tabellen zu groß, d.h. man muß mit geeigneten Restriktoren arbeiten. Vielleicht kann man die Vorberechnungen schematisch machen, indem man Regeln mit `clause((A --> B),C)` statt mit `(A --> B)` sucht, also dabei die Bedingungen noch nicht auswertet?

Problem 5.5 Regeln, in denen freie Variablen auftreten, sind mit Vorsicht zu benutzen: In den Parsetabellen dürfen die freien Variablen nicht belegt werden, sondern man muß Kopien mit entsprechener Belegung anlegen, sonst werden Lesarten mit anderer Variablenbelegung verhindert. *Ist das inzwischen korrekt behoben?* (Testbeispiel?)

5.1.5 Hornformelgrammatiken (DCG)

Neben den schematischen kontextfreien Grammatikregeln betrachten wir schematische EBNF-Regeln. Das führt zum Format der *Definite Clause Grammar (DCG)* oder *Hornklauselgrammatik*.

Eine *Kategorie* ist auch hier ein einfacher Prolog-Term, der keine Variable ist. Eine Grammatikregel

⟨*DCG-Regel*⟩≡
```
A --> B.
```

ist eine *DCG-Regel*, wenn `A` eine einfache Kategorie ist und `B` aus einfachen Kategorien mit (sequentieller) Konjunktion `(B1,B2)`, Disjunktion `(D1;D2)` und *Instruktion* `{Goal}` aufgebaut ist, worin `Goal` ein Prolog-Term, aber keine Variable ist.

Eine *bedingte DCG-Regel* ist eine Grammatikregel der Form

⟨*bedingte DCG-Regel*⟩≡
```
(A --> B) :- C.
```

wenn `A --> B` eine DCG-Regel ist und die Bedingung `C` ein Prolog-Term, der keine Variable ist. Sie bedeutet dasselbe wie die DCG-Regel `A --> ({C},B)`, außer daß im Parser im ersten Fall die Bedingung `C` beim Holen der Regel `(A --> B)` ausgewertet wird, während sie im zweiten Fall erst bei der Anwendung ausgewertet wird.[8]

Wir verlangen folgende Einschränkung: alle freien Variablen einer bedingten Regel werden (a) durch Vergleich mit einem Terminal der Eingabe, oder (b) durch Ausführen einer Instruktion `{Goal}`, oder (c) durch Beweisen der Bedingung `C` mit einem *Grund*term belegt. Die Bedingung `C` darf keine Operationen auf der Datenbank (wie `assert`, `retract` usw.) ausüben.

Als syntaktische Vereinfachung lassen wir *optionale Konstituenten* `?D` ebenfalls in `B` zu; für jedes solche `?D` muß die Grammatik durch spezielle Löschregeln `(?D --> []) :- C` festlegen, wie die Variablen in `D` bei der Löschung zu lesen sind. Weitere Regeln mit linker Seite `?D` darf es nicht geben – die Parser tun aber so, als gebe es auch die Regel `(?D --> D)`.

Eine *Pseudo-DCG-Regel* ist eine bedingte DCG-Regel `(A --> B1,...,Bk) :- C.`, deren rechte Seite eine Konjunktion von Kategorien `Bi` ist, die wie folgt aufgebaut sein dürfen:

(i) `Bi` ist eine Disjunktion `(D1;...;Dm)` von ($1 \leq m$) einfachen Kategorien, oder

(ii) `Bi` ist von der Form `?(D)` mit einer Disjunktion `D` von einfachen Kategorien.

Bem.: Solche Regeln werden in der Satzgrammatik aus Teil ?? (noch) verwendet. Problematisch an ihnen ist, daß bei `(A --> B) :- C.` mit Disjunktionen in `B` sich die Bedingungen in `C` auf *alle* Disjunktionsglieder beziehen, man aber oft für unterschiedliche Disjunktionsglieder unterschiedliche Bedingungen braucht.

Beispiel 5.6 (Valenz, optionale Argumente, Vertauschbarkeit von Argumenten) In der ersten Satzregel wird hier die Vertauschbarkeit von Subjekt und Objekt beschrieben, in der zweiten wird ein optionales Argument im Verbrahmen mit einer optionalen Konstituenten im Satz verglichen. Nach

[8] Dagegen würde sie bei `A --> (B,{C})` erst nach dem Finden von Konstituenten `B` ausgewertet. Beachte, daß bedingte DCG-Regeln (bei entsprechendem `-->`-Symbol) aber nicht von der Einleseroutine in Prolog-Klauseln umgewandelt werden können, da sie keine Fakten sind.

den angegebenen Löschregeln sollen alle Nominalphrasen, aber von den Adjektiven nur die im Nominativ löschbar sein!

⟨*Grammatiken/dcg.1.grm*⟩≡

```
:- op(1100,xfx,'-->').
auxiliary_predicates([start_symbol/1, kasus/1, anordnung/2]).

start_symbol(s).
start_symbol(np([Kas])) :- kasus(Kas).

kasus(nom).
kasus(akk).

anordnung([A,B],[A,B]).
anordnung([A,B],[B,A]).

s -->  np([U]), v([X,?Y]), np([V]), { anordnung([X,Y],[U,V]) }.
s -->  np([X]), v([X,?Y]), ?np([Y]).

np([Kas]) --> art([Kas]), ?a([Kas]), n([Kas]).

?a([nom]) --> [].
?np([Kas]) --> [].
```

Da die Parser implizit annehmen, daß es ein Regelschema `?A --> A.` gibt, enthält die Grammatik die Regeln `?a([Kas]) --> a([Kas]).` und `?np([Kas]) --> np([Kas]).` nicht.

Im Lexikon geben wir ein Verb mit optionalem Objekt im Verbrahmen[9] vor:

⟨*Lexika/dcg.1.wrd*⟩≡

```
word(das,art([nom])).      word(das,art([akk])).
word(wilde,a([nom])).      word(wilde,a([akk])).
word('Tier', n([nom])).    word('Tier', n([akk])).
word(frißt, v([nom,?(akk)])).
```

Mit der ersten Regel werden beide Stellungen der Komplemente erkannt. Die zweite Regel kodiert die Valenz durch Gleichheit von Elementen im Komplementrahmen mit *Merkmalen von* Nominalphrasen-Konstituenten. Sie erlaubt das Fehlen des Objekts, wenn das Subjekt vor dem Verb steht:

⟨*Testbett/Beispiele/dcg.1.txt*⟩≡

```
% a. Vertauschbarkeit von Subjekt- und Objekt-NP:
das wilde Tier frißt das wilde Tier.
% b. Weglassen des optionalen np-Objekts:
das wilde Tier frißt.
% c. Weglassen des Adjektivs im Nominativ:
das Tier.  das Tier frißt.
% d. Kein Weglassen des Adjektivs im Akkusativ:
das Tier frißt das Tier.
```

Diese Behandlung optionaler Komplemente wird noch verbessert, da man „dieselbe“ Satzregel einmal für Verben, deren Objekt obligatorisch ist, und einmal für Verben, deren Objekt optional ist, schreiben müßte.

[9] Die Angabe optionaler Argumente im Komplementrahmen, also `[nom,?akk]`, ist unabhängig davon, ob `np([akk])` eine Löschregel hat. Das kann verwirren – man braucht also eine klare Konvention!

Beispiel 5.7 Wir formulieren die Grammatik `cfg.2.grm` so um, daß die Bereichsbeschränkungen `C` aus den Regeln `(A --> B) :- C.` zu Bestandteilen der rechten Regelseite werden, d.h. zu Instruktionen `{C}` in `(A --> B,{C})`.

⟨*Grammatiken/dcg.2.grm*⟩≡
  ⟨*cfg.2.deklarationen*⟩
  ⟨*dcg.2b.grammatikregeln*⟩

Die geänderten Regeln lauten:

⟨*dcg.2a.grammatikregeln*⟩≡
```
% Grammatikregeln:

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)}.

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)}.

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

ap([Gen,Num,Kas]) --> grad, a([Gen,Num,Kas]),
        {genus(Gen),numerus(Num),kasus(Kas)}.

ap([mask,sg,nom]) --> ap([mask,sg,nom]), conj, ap([mask,sg,nom]).

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

Wir testen diese Grammatik mit dem Lexikon `Lexika/cfg.2.wrd` für `cfg.2.grm`. Natürlich kann sie nur mit den für DCG-Grammatiken vorgesehenen Parsern benutzt werden.

Beim Beispielsatz `der Hamster preift ein Lied` erhält man folgende Anzahl von Inferenzen (die Laufzeiten und Anteile in einzelnen Funktionen sind bei so kleinen Beispielen nicht aussagekräftig):[10]

⟨*Statistik für die Regeln* `dcg.2a`⟩≡
```
First-Relation: [explicit, inductive]   Predictor: [explicit]
Parser: [earley_doerre_dcg]             Grammatik: [dcg.2]
% 8,182 inferences in 0.02 seconds (409100 Lips)

First-Relation: [explicit, inductive]    Predictor: [explicit]
Parser: [earley_pratt_leiss]             Grammatik: [dcg.2]
```

[10] Nach einem früheren Test ist die Abtrennung der Bereichsbedingungen (d.h. `cfg.2`) effektiver als ihre Formulierung durch Instruktionen (d.h. `dcg.2a`). Das kann ich z.Zt. nicht reproduzieren, da die `.ed`-Analyse jetzt sehr mehrdeutig wird (doppelte `closure/6`-Klauseln)??

```
% 8,690 inferences in 0.02 seconds (434500 Lips)

First-Relation: [explicit, inductive]     Predictor: [explicit]
Parser: [leftcorner_generalized]          Grammatik: [dcg.2]
% 8,526 inferences in 0.01 seconds (852600 Lips)
```

Bei der Option `predict(explizit)` werden einige Tests schon bei der Vorberechung ausgeführt und danach die rechten Regelseiten vereinfacht, so daß man gewisse Tests zur Parsezeit nicht mehr machen muß.

Fairerweise muß man aber mit den DCG-Regeln vergleichen, bei denen die Instruktionen *vor* den Konstituentenkategorien auftreten, denn dann werden sie –ähnlich wie die Bedingungen in den schematischen Regeln, zuerst ausgewertet. Die so geänderten Regeln lauten:

⟨*dcg.2b.grammatikregeln*⟩≡

```
% Grammatikregeln:

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

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

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

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

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

ap([Gen,Num,Kas]) -->
        {genus(Gen),numerus(Num),kasus(Kas)},
        grad, a([Gen,Num,Kas]).

ap([mask,sg,nom]) --> ap([mask,sg,nom]), conj, ap([mask,sg,nom]).

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

In diesem Fall erhält man

⟨*Statistik für die Regeln* `dcg.2b`⟩≡

```
First-Relation: [explicit, inductive]     Predictor: [explicit]
Parser: [earley_doerre_dcg]               Grammatik: [dcg.2]
% 9,156 inferences in 0.04 seconds (228900 Lips)

First-Relation: [explicit, inductive]     Predictor: [explicit]
```

```
Parser: [earley_pratt_leiss]              Grammatik: [dcg.2]
% 8,689 inferences in 0.02 seconds (434450 Lips)

First-Relation: [explicit, inductive]     Predictor: [explicit]
Parser: [leftcorner_generalized]          Grammatik: [dcg.2]
% 9,879 inferences in 0.03 seconds (329300 Lips)
```

Beispiel 5.8 (Faktorisierung und Merkmalbelegung) Wenn man in Merkmalgrammatiken mehrere Regeln durch Faktorisierung zusammenfassen will, muß man i.a. die Belegung von Merkmalen in den einzelnen Disjunktionsgliedern durch Instruktionen `{Goal}` vornehmen.

In Nominalphrasen ist die Adjektivdeklination `Adekl` ein Merkmal, das z.B. bei der Verwendung der Nominalphrase in Vergleichskonstruktionen wie `ein dümmerer Fehler als deiner` eine Rolle spielt. Es wird von der Art des Artikels regiert und hat die Werte `st,sw,ge` (stark, schwach, gemischt), und den Sonderwert `-`, falls kein Adjektivattribut vorkommt. Um nun die beiden Regeln

⟨*Nominalphrasen mit und ohne Adjektivattribut*⟩≡

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

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

regiert(Art,Adekl) :- Adekl = Art.
```

durch Faktorisierung zusammenzufassen, muß man die Rektionsbedingung in der entsprechenden Alternative als Instruktion angeben:

⟨*Rektionsbedingung in einzelnen Disjunktionsgliedern*⟩≡

```
np([Gen],[Num,Kas,Adekl]) -->
        det([Art],[Gen,Num,Kas]),
        ( ap([Adekl,Gen,Num,Kas]), {regiert(Art,Adekl)}
        ; [], {Adekl = -}
        ),
        n([Gen],[Num,Kas])
        :- genus(Gen), numerus(Num), kasus(Kas).

regiert(Art,Adekl) :- Adekl = Art.
```

Beachte, daß man zwar in der ersten der getrennten Regeln die Bedingung `regiert(Art,Adekl)` durch Verwenden derselben Variablen vermeiden kann, aber nicht in der faktorisierten Regel. Denn beim Linksfaktor `det([Art],[Gen,Num,Kas])` tritt der Sonderwert `-` nicht als `Art` auf, sondern nur die normalen Werte.

Beispiel 5.9 (Rektion bei optionalen Konstituenten) In Grammatiken mit schematischen Regeln kann man keine *universelle* Bedingung, etwa `?(A) --> [].` mit einer *Variablen* `A`, für die Löschbarkeit optionaler Konstituenten angeben. Es kann sein, daß `A` ein synthetisches Merkmal hat, das aber regiert wird, etwa die vom Artikel regierte Adjektivdeklination `Adekl` innerhalb der Nominalphrase:

⟨*Rektion eines synthetischen Merkmals*⟩≡

```
np([Num,Kas]) -->
        det([Art],[Gen,Num,Kas]),
```

```
        ?ap([Adekl,Gen,Num,Kas]),
        n([Gen],[Num,Kas])
        :- genus(Gen), numerus(Num), kasus(Kas), regiert'(Art,Adekl).
```

Damit diese Regel auch die Fälle erfaßt, wo keine `ap` vorkommt, muß die Löschregel für `?ap` sagen, daß `Adekl` mit einem Sonderwert belegt wird, und die Rektionsbedingung `regiert'/2` muß diesen berücksichtigen. Die optionale Kategorie erfordert also eine Regel der Form:

⟨*optionale Konstituente*⟩≡

```
  ?ap([Adekl,Gen,Num,Kas]) -->
        ( ap([Adekl,Gen,Num,Kas]), {adekl(Adekl)}
        ; [], {Adekl = -}
        )
        :- genus(Gen), numerus(Num), kasus(Kas).
```

Hier wird gesagt, daß *entweder* die Adjektivphrase nicht leer ist und die Variable `Adekl` durch `adekl(Adekl)` auf die normalen (synthetischen) Werte beschränkt ist, *oder* daß die Adjektivphrase fehlt und `Adekl` den Sonderwert `-` hat. Die Rektionsbedingung `regiert'(Art,Adekl)` der Nominalphrase muß dann diese Fallunterscheidung nachspielen.

Da in einer solchen Regel

⟨*optionale Konstituente*⟩+≡

```
  ?A --> ( A ; [], {sonderwerte(A)} ).
```

die linke Alternative `?A --> A` uniform in `A` ist, kann man sie fest in die Parser einbauen. Es genügt also, wenn der Grammatikschreiber Löschbedingungen wie

⟨*Belegung synthetischer Attribute beim Löschen*⟩≡

```
  (?A --> []) :- sonderwerte(A).
```

angibt, mit denen die synthetischen Attribute fehlender optionaler Konstituenten belegt werden. (Da beide Alternativen durch Ketten- und Löschregeln ausgedrückt werden, erzeugt `?A` *keine* Knoten in den kanonischen Syntaxbäumen.)

5.1.6 Grammatiken mit Metaregeln

Metaregeln sind Ausdrücke, aus denen Grammatikregeln erzeugt werden; in diesem Sinne sind schon die schematischen Regeln `(A --> B) :- C.` Metaregeln, da sie zur Erzeugung von Grammatikregeln `(A' --> B')` dienen. Wir bezeichnen als *Metaregeln* aber nur solche Regelschemata, bei denen in `B` *auch Variable* als Konstituentenkategorien erlaubt sind.

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 vollständigen Phrasen (an den Stellen `...`), deren Kategorien „unbekannt“ sind, d.h. erst durch `X` genauer bestimmt werden. Hier soll eine besondere Beziehung zwischen `X'` und `X` bestehen, z.B. die zwischen Verbalphrase `X'` und dem finiten Verb `X`, und bei `...` stehen die Kategorien der Argumente des Verbs.

An die variablen Konstituenten `Y` einer Metaregel müssen wir noch die Einschränkung machen, daß *spätestens* dann, wenn der Parser nach eine `Y`-Konstituenten sucht, die Variable `Y` teilweise belegt ist. Das sichern wir dadurch, daß wir nur

⟨*erlaubte Metaregeln*⟩≡

```
  (A --> B1,...,{Goal},...,Y,...,Bk) :- Cond.
```

verwenden, bei denen `Y` durch Auswerten vorangegangener Instruktionen `{Goal}` oder durch die Bedingung `Cond` teilweise belegt wird.

Beispiel 5.10 (Kopf-Komplement-Schema, Version 1) Man kann mehrere CFG-Satzregeln in eine DCG-Regel zusammenfassen, indem man die Kategorien der Komplemente eines Verbs und verschiedene ihrer Stellungsvarianten zur Parsezeit durch Auswerten einer Instruktion `{Goal}` aus dem Komplementrahmen des Verbs berechnet.

Damit die Vorberechnung der Hilfstabellen für die `.epl`- und `.glp`-Parser möglich bleibt, muß allerdings die Kategorie der linkesten Konstituente im Regelschema genannt sein, hier `conj`.

⟨*Grammatiken/meta.1.grm*⟩≡

```
  ⟨cfg.2.deklarationen⟩

  auxiliary_predicates([expandiereCat/2,expandiereCats/2,komplementrahmen/1,
                        argumentstellung/3,anordnung/2, kongruenz/2]).

  expandiereCat(?X,?XP) :- expandiereCat(X,XP).
  expandiereCat(nom, np([Gen],[Num,nom])).
  expandiereCat(dat, np([Gen],[Num,dat])).
  expandiereCat(akk, np([Gen],[Num,akk])).

  expandiereCats([],[]).
  expandiereCats([X|Xs],[C|Cs])
          :- expandiereCat(X,C), expandiereCats(Xs,Cs).

  argumentstellung(Rahmen,SubjCat,Cats) :-
          expandiereCats(Rahmen,Expandiert),
          Expandiert = [SubjCat|_],
          anordnung(Expandiert,Cats).

  anordnung([A,B],[A,B]).
  anordnung([A,B],[B,A]).

  kongruenz(np([_Gen],[Num,nom]),[Num,3]).

  restrict(X,X).

  % Grammatikregeln: eine Satzregel als Metaregel

  s -->
          conj, { komplementrahmen(Rahmen),
                  argumentstellung(Rahmen,NOM,[XP,YP]) },
          XP, v([Rahmen],[Num,Pers]), ?pp(lok), YP
          :-
          kongruenz(NOM,[Num,Pers]).

  komplementrahmen([nom,?akk]).

  % Grammatikregeln: sonstige wie bisher:

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

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

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

?grad --> [].
?pp(lok) --> [].
?ap([Gen,Num,Kas]) --> []
        :- genus(Gen), numerus(Num), kasus(Kas).

% Neuer Lexikoneintrag:

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

Macht man die linke Ecke `conj` der Satzregel optional (mit einer Löschregel für `?cond`), so werden die Vorberechnungen der `first/2`-Relation für die `.epl` und `.gpl`-Parser falsch, da die unbekannte Kategorie `X` nicht berücksichtigt werden kann. (Der `.eddcg`-Parser hat damit kein Problem.)

Beispiel 5.11 (Kopf-Komplement-Schema, Version 2) Will man erreichen, daß auch die linke Ecke eines Regelschemas eine Variable sein darf, so muß man die Expansion des Komplementrahmens und die Vertauschungen der Komplementkategorien schon bei der Grammatikübersetzung berechnen:

⟨*Grammatiken/meta.2.grm*⟩≡
  ⟨*cfg.2.deklarationen*⟩

```
auxiliary_predicates([expandiereCat/2,expandiereCats/2,komplementrahmen/1,
                      argumentstellung/3,anordnung/2, kongruenz/2]).

expandiereCat(?X,?XP) :- expandiereCat(X,XP).
expandiereCat((A;B),(AP;BP))
        :- expandiereCat(A,AP), expandiereCat(B,BP).

expandiereCat(nom, np([Gen],[Num,nom])).
expandiereCat(dat, np([Gen],[Num,dat])).
expandiereCat(akk, (np([Gen],[Num,akk]), {genus(Gen)})).

expandiereCats([],[]).
expandiereCats([X|Xs],[C|Cs])
        :- expandiereCat(X,C), expandiereCats(Xs,Cs).

argumentstellung(Rahmen,SubjCat,Cats) :-
        expandiereCats(Rahmen,Expandiert),
        Expandiert = [SubjCat|_],
        anordnung(Expandiert,Cats).

anordnung([A,B],[A,B]).
anordnung([A,B],[B,A]).

kongruenz(np([_Gen],[Num,nom]),[Num,3]).
```

```
restrict(X,X).

% Grammatikregeln: eine Satzregel als Metaregel (mit variabler linker Ecke):

s -->  XP, v([Rahmen],[Num,Pers]), ?pp(lok), YP
        :- komplementrahmen(Rahmen),
           argumentstellung(Rahmen,NOM,[XP,YP]),
           kongruenz(NOM,[Num,Pers]).

komplementrahmen([nom,?(dat;akk)]).

% Grammatikregeln: sonstige wie bisher:

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

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

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

?grad --> [].
?pp(lok) --> [].
?ap([Gen,Num,Kas]) --> []
        :- genus(Gen), numerus(Num), kasus(Kas).

% Wie kann man wissen, welche optionalen Argumente in den Komplementrahmen stehen?
?np(_,_) --> [].

% Neuer Lexikoneintrag:

word(pfeift,v([[nom,?(dat;akk)]],[sg,3])).
```

Beachte, daß die Abkürzung `akk` zur Kategorie `np([Gen],[Num,akk])` mit einer zur Parsezeit auszuwertenden Bedingung `{genus(Gen)}` expandiert wird.

Bem. Gibt man optionale Argumente in den Komplementrahmen unabhängig von einander an, etwa `[nom,?dat]` und `[nom,?akk]` für dasselbe Verb, so wird eine Eingabe, die nur die obligatorischen Argumente (hier `nom`) hat, mehrdeutig: einmal mit gelöschtem `?dat` und einmal mit gelöschtem `?akk`. Um das zu vermeiden, sollte man *optionale Komplementalternativen* wie `[nom,?(dat;akk)]` angeben dürfen. Das geht provisorisch, bei `Der Hamster pfeift ([] | ein Lied | der Grube).` Aber die Behandlung komplexer Kategorien `C` unter `?C` ist noch unvollständig.

5.1.7 Merkmalgrammatiken

Für größere Grammatiken ist es nützlich, wenn man die lexikalischen und syntaktischen Kategorien in verschiedene Unterklassen zerlegt, je nach den gemeinsamen syntaktischen Eigenschaften der Ausdrücke dieser Unterklassen. Diese Eigenschaften werden in den Grammatikregeln benutzt, um in bestimmten Kontexten nur Ausdrücke mit bestimmten Eigenschaften zuzulassen.

Die syntaktisch relevanten Eigenschaften einer Unterkategorie speichert man als *Merkmale* im Namen der Unterkategorie, um darauf schnell zugreifen zu können.[11]

Es gibt zwei Möglichkeiten, die syntaktischen Analysen von Ausdrücken einer Merkmalsgrammatik darzustellen:

(i) Wird jede lexikalische Lesart separat behandelt, so genügt es, bei den syntaktischen Kategorien mit *synthetischen* Merkmalen zu arbeiten: jede Konstituente hat als Kategorie einen *geschlossenen Term* von Prolog. Der Nachteil ist, daß viel Struktur doppelt gespeichert werden muß:

⟨*Beispiel*⟩≡
```
1. np([def, 3, mask], [pl, gen])
      det([def, sw], [unsp, pl, gen])
         der
      n1([mask], [-, sw, pl, gen])
         n([mask], [pl, gen])
            Wissenschaftler
2. np([def, 3, mask], [sg, nom])
      det([def, sw], [mask, sg, nom])
         der
      n1([mask], [-, sw, sg, nom])
         n([mask], [sg, nom])
            Wissenschaftler
```
Hier besteht der Unterschied nur in den Formeigenschaften `Num,Kas`.

(ii) Werden verschiedene lexikalische Lesarten simultan behandelt, so sind die lexikalischen Kategorien *offene Terme* mit *Nebenbedingungen an die freien Variablen* (Constraints): jede Einsetzung von Merkmalwerten in die Variablen, die die Nebenbedingungen erfüllt, vervollständigt den offenen Term zu einer Kategorie im vorigen Sinne. Dann ist der Speicheraufwand geringer, etwa wie bei

⟨*Beispiel*⟩+≡
```
Struktur:
   np([def, 3, mask], [Num, Kas])
      det([def, sw], [unsp, Num, Kas])
         der
      n1([mask], [-, sw, Num, Kas])
         n([mask], [Num, Kas])
            Wissenschaftler
Constraint:
([Num,Kas] = [pl,gen] \/ [Num,Kas] = [sg,akk]),
```
aber bei den Grammatikregeln muß gesagt werden, wie sich die Nebenbedingungen zur Kategorie einer Phrase aus den Nebenbedingungen der Kategorien der Teilphrasen berechnen lassen. Man muß diese Nebenbedingungen auf Lösbarkeit testen bzw. in gelöster Normalform halten.

Wir wollen nur den ersten Fall von Merkmalverwaltung behandeln, wo als Kategorien nur *geschlossene Terme im Herbrand-Universum* erlaubt sind, also auch keine Termauswertung im Spiel ist.

Dafür wollen wir bekannte Parser so verallgemeinern, daß sie mit gewöhnlichen DCG-Regel arbeiten können, zumindest in der Form, wo die Instruktionen `{Goal}` in Regeln `A --> B,...,{Goal},...,C` nur zur Berechnung von Attributen bei `A` oder zum Vergleich der Attribute von `B,...,C` verwendet werden.

[11] Selten zu berücksichtigende Eigenschaften sollten nicht zur Einführung feinerer Unterklassen und entsprechender Merkmale führen; sie sollten als Tests an die jeweils an einer syntaktischen Konstruktion beteiligten Konstituenten formuliert werden. Wie das technisch geht, lassen wir hier offen; bei Tabellenparsern kann man aber –im Unterschied zu Top-Down-Parsern– im Prinzip die gesamten Syntaxbäume der beteiligten Konstituenten überprüfen.

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.[12]

(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.'))
          )
        ).
```

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

Nach der lexikalischen Analyse der Eingabe wird mit einem aus der Grammatik vorberechneten Prädikat `lexCat/1` (vgl. Abschnitt 6.1.1) 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[13]:

⟨*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.')))
        ).
```

[13] 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 $w(i,j) := a_i \cdots a_{j-1}$ von Position i bis j in der Eingabe $w := a_0 \cdots a_n$ 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 $w = w(0,n+1)$ im Feld $F(0,n+1)$. Daran kann man also erkennen, ob w ein korrekter Satz ist.

Tabellenverfahren sind meist nur *Erkennungsverfahren.* Aus der Tabelle zu w kann man aber (bis auf unwichtige Unterschiede) alle Analysen rekonstruieren - indem 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

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

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* Kanten $A \to \alpha \cdot$ bzw. *aktiven* Kanten $A \to \alpha \cdot \beta$ mit $\beta \neq \epsilon$.) Die Eingabe $a_0 \cdots a_n$ ist erkannt, wenn es eine Kante $0 \overset{S \to \alpha \cdot}{\longrightarrow} n + 1$ gibt.

Nach der lexikalischen Analyse der Eingabe $w = a_0 \cdots a_n$, die eine Tabelle `lexword/2` festlegt, erfolgt der Aufbau der Kanten durch Abschließen unter den folgenden Regeln:

(i) Festlegen der Erwartungskategorien für die gesamte Eingabe:

$$\frac{\mathit{start}(S) \qquad S \to \alpha \in P}{0 \overset{S \to \cdot \alpha}{\longrightarrow} 0} \quad \text{(Start)}$$

(ii) Lesen der Eingabewörter:

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

(iii) Voraussagen brauchbarer Regeln:

$$\frac{i \overset{C \to \gamma \cdot A\delta}{\longrightarrow} j \qquad A \to \alpha \in P}{j \overset{A \to \cdot \alpha}{\longrightarrow} j} \quad \text{(Predict)}$$

(iv) Abschluß unter Verkettung von Kanten:

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

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,Baumliste,Beta).
```

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

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

(Beim Erkenner schreiben wir statt der `Baumliste` den Teil `Alpha` der rechten Regelseite.) 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 $0 \xrightarrow{S \to \cdot\alpha} 0$ mit Startsymbol S und leerer Analyse berechnet:

⟨*Earley-Dörre, Parser (mit Löschbäumen)*⟩≡
```
:- dynamic kante/5.

parse(_,_) :-
        retractall(kante(_I,_J,_A,_alpha,_beta)),
        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,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]],[]),
        fail.
shift(_,_,_).
```

Beim Abschließen der Kanten unter Verkettung, `closure/5`, werden die Baumfolgen der entsprechenden Felder verkettet. Man ruft `closure(I+,J+,A+,Baumliste+,Beta+)` 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,Baumliste,[]) :-
        neue_kante(I,J,Kopf,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,Baumliste,[B|Bs]) :-
        deletable(B),
        append(Baumliste,[[B,[]]],BaumlisteB),
        closure(I,J,Kopf,BaumlisteB,Bs).
```

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,Baumliste,[?(B)|Bs]) :-
        !,closure(I,J,Kopf,Baumliste,[B|Bs]).

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

(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,Baumliste,[B|Bs]) :-
        neue_kante(I,J,Kopf,Baumliste,[B|Bs]),
        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 8). Wenn ja, ist die Kante nicht neu:

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

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

Wenn nicht, macht man den Eintrag und fährt fort:

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

```
  neue_kante(I,J,Kopf,Baumliste,Offen) :-
          (ein(verbose)
          -> nl,nl,write_kante(I,J,Kopf,Baumliste,Offen)
          ; true),
          assertz(kante(I,J,Kopf,Baumliste,Offen)).
```

Bem.: Die einfachere Regel

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

```
  neue_kante(I,J,Kopf,Baumliste,Offen) :-
          kante(I,J,Kopf,Baumliste,Offen),
          !,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,Baumliste,Offen) :-
          kante(I,J,KopfX,BaumlisteX,OffenX),
          matches((Kopf,Baumliste,Offen),
                  (KopfX,BaumlisteX,OffenX)),
          !,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,Baumliste,[B|Beta]),
          append(Baumliste,[[B|Teilbaeume]],BaumlisteB),
          closure(K,J,A,BaumlisteB,Beta),
          fail.
  complete(_,_,_,_).
```

Wenn ab j ein Ausdruck der Kategorie A erwartet wird, muß der Abschluß der neuen Kanten $j \stackrel{A\rightarrow\cdot\alpha}{\longrightarrow} j$ 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,Baumliste,Offen) :-
        tab(3),write(I),write(','),
        write(J),
        write(': ('),
        write(Kopf),write('-->'),write('.'),write(Offen),
        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 5.12 Hier ist die entsprechende Grammatik zur DCG aus Beispiel 5.1:

⟨*Grammatiken/cfg.1.grm.ed*⟩≡
```
:- 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 5.12 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. Außerdem wird in diesem Parser gleich auch die Behandlung von EBNF-Regeln ermöglicht. Für die durch `?C` markierten optionalen Konstituenten wird auch ohne Angabe der Option `omitEmtyTrees` kein leerer Teilbaum aufgebaut.

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)),
        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 5.13 (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

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:[14]

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

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

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

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

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

   (b) für jedes i die Teilanalysen zu $B_i \Rightarrow^* w_i$ 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 $A \Rightarrow^* B$ und die zu den $B_i \Rightarrow^* w_i$ kanonische Analysen sind.

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.

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 $\vec{t} \in F(i,j,A \rightarrow \alpha \cdot B)$ verbunden werden soll, folgende Fälle ausnehmen:

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

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

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.

5.3.4 Erweiterung des Earley-Doerre-Parsers auf DCG-Regeln

Aufgabe 5.5 Modifiziere den Parser von Earley in der Version von Dörre so, daß er mit *bedingten Definite-Clause-Regeln* nach Abschnitt 5.1.5 arbeiten kann. Die Idee ist, daß alle freien Variablen von `(A --> B) :- C` für *synthetische Attribute* stehen, die bei einer Analyse durch Grundterme belegt werden, die aus Teiltermen der lexikalischen Elementen und den Instruktionen `{Goal}` konstruiert werden.

Lösung zu Aufgabe 5.5: Wir modifizieren die Implementierung aus der Lösung zu Aufgabe 5.4. Dabei lassen wir in den Tabellen in den gepunkteten Regenl `A --> Alpha.Beta` den Teil `Alpha` weg, da er nur zur besseren Lesbarkeit diente. Stattdessen setzen wir die schon gefundene `Baumliste` ein.

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

Vorerst werden die Bedingungen `C` ignoriert. Instruktionen `{Goal}` sind nicht(!) löschbar im Sinne von `deletable/1`:

⟨*Earley-Dörre, dcg*⟩≡

```
:- dynamic kante/5.
:- ensure_loaded('Parser/subsumption').

parse(_,_) :-
        retractall(kante(_I,_J,_A,_Baeume,_Beta)),
        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,Baeume,[]),
        Baum = [S|Baeume].

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

closure(I,J,Kopf,Baumliste,[]) :-
        neue_kante(I,J,Kopf,Baumliste,[]),
        complete(I,J,Kopf,Baumliste).

closure(I,J,Kopf,Baumliste,[B|Bs]) :-
        deletable(B),
        (ein(omitEmptyTrees)
        -> BaumlisteB = Baumliste
        ; append(Baumliste,[[B,[]]],BaumlisteB)
        ),
        closure(I,J,Kopf,BaumlisteB,Bs).
```

Instruktionen auf der rechten Regelseite werden (analog zu einer Suche in der Eingabe bzw. in der Tabelle der Zwischenergebnisse) während der Analyse ausgeführt. Mit den durch die Ausführung erzeugten Belegungen wird die Tabelle weiter abgeschlossen:

⟨*Earley-Dörre, dcg*⟩+≡

```
closure(I,J,Kopf,Baumliste,[{Goal}|Bs]) :-
        !, call(Goal),
        closure(I,J,Kopf,Baumliste,Bs).
```

Bei optionaler Kategorie `?B` auf der rechten Regelseite wird zuerst so getan, als sei dies die Kategorie `B`, d.h. als habe die Grammatik eine Regel `?B --> B`. Anschließend wird durch Aufrufen von `deletable(?B)` gesichert, daß sie nur bei den Belegungen ignoriert werden, für die es in der Grammatik (vor der Vorberechnung) eine Löschregel `(?B --> []) :- C` gab:

⟨*Earley-Dörre, dcg*⟩+≡
```
closure(I,J,Kopf,Baumliste,[?(B)|Bs]) :-
        !,closure(I,J,Kopf,Baumliste,[B|Bs]),
        deletable(?B),
        closure(I,J,Kopf,Baumliste,Bs).
```

Disjunktionen und Konjunktionen auf der rechten Regelseite werden wie bisher behandelt:

⟨*Earley-Dörre, dcg*⟩+≡
```
closure(I,J,Kopf,Baumliste,[(B;C)|Bs]) :-
        !,closure(I,J,Kopf,Baumliste,[B|Bs]),
        closure(I,J,Kopf,Baumliste,[C|Bs]).

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

⟨*Earley-Dörre, dcg*⟩+≡
```
closure(I,J,Kopf,Baumliste,[B|Bs]) :-
        neue_kante(I,J,Kopf,Baumliste,[B|Bs]),
        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, dcg*⟩+≡
```
neue_kante(I,J,Kopf,Baumliste,Offen) :-
        known_instance(kante(I,J,Kopf,Baumliste,Offen)),
        !,fail.
neue_kante(I,J,Kopf,Baumliste,Offen) :-
        (ein(verbose)
        -> nl,nl,write_kante(I,J,Kopf,Baumliste,Offen)
        ; true),
        assertz(kante(I,J,Kopf,Baumliste,Offen)).

complete(I,J,B,Teilbaeume) :-
        kante(K,I,A,Baumliste,[B|Beta]),
        ((ein(omitEmptyTrees), Teilbaeume = [])
        -> BaumlisteB = Baumliste
        ; 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,BaumlisteB,Beta)),
        fail.
complete(_,_,_,_).

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

```
write_kante(I,J,Kopf,Baumliste,Offen) :-
        tab(3),write(I),write(','),
        write(J),write(': ('),
        write(Kopf),write('-->'),write('.'),write(Offen),
        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, dcg*⟩+≡
```
stamm(A,[A|Teilbaueme]) :- !.
stamm(A,[B,Teilbaum]) :-
        stamm(A,Teilbaum).
```

Für diese Lösung ist auch wichtig, daß man die Löschbarkeit `deletable/1` auf zusammengesetzte Kategorien ausdehnt. (vgl. Abschnitt 6.2.1). △

6 Grammatikübersetzung

Um ein Analyseverfahren zu erhalten, das nur die (endlich vielen) kanonischen Bäume zu einem Eingabesatz 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 $A \Rightarrow^* \epsilon$ und $A \Rightarrow^* B$ 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 8.1 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*⟩
⟨*Syntaktische Kategorien*⟩
⟨*Kategorienprüfung*⟩
⟨*Vereinfachung von Bedingungen*⟩
⟨*Vereinfachung von Kategorien*⟩
⟨*Löschbare Kategorien*⟩
⟨*Kettenbeziehung*⟩
⟨*Vorberechnung der First-Relation*⟩
⟨*Vorberechnung brauchbarer Regeln*⟩

Der Dateiname `cfg-vorberechnungen.pl` ist etwas überholt: es wird versucht, möglichst nicht nur kontextfreie, sonderen auch erweitert kontextfreie und DCG-Regeln zu berücksichtigen.

Warnung. Die Korrektheit und Vollständigkeit dieser Vorberechnungen für EBNF-Regeln und DCG-Regeln ist nicht gesichert. Es werden Varianten mit Grundfakten, schematischen Fakten und bedingten Fakten angegeben, die z.T. noch weiter zu testen sind.

Von den Konjunktion `(A,B)` von Kategorien werden nur die interessanten Fälle `(A,{C})` und `({C},B)` behandelt. Bedingungen `{C}` werden nur bei den Löschregeln ausgewertet, sonst werden die Bedingungen in erzeugte Klauseln übernommen oder Fakten mit freien Variablen gebildet, da man sonst zu große Tabellen bekommt. Im allgemeinen Fall müßte man voraussetzen, daß alle Bedingungen `{C}` zur Vorberechnungszeit auswertbar sind. (Die Behandlung der zusammengesetzten Kategorien erfolgt in `instance/2` so, daß dafür keine Knoten in den Syntaxbäumen gebildet werden! Das gilt für `(A;B)` und `?A`, der Fall `(A,B)` ist nur erfaßt, wenn ein Konjunktionsglied eine Instruktion ist.)

Bem. Für den Earley-Dörre-Parser muß nur die Eigenschaft `deletable/1` und die Relation `chain/2` zur Bestimmung der kanonischen Bäume passend definiert werden. Für den Earley-Pratt-Parser und den Leftcorner-Parser muß die First-Relation und die Menge der vorausgesagten Regeln, `predicted/6` bzw. `predicted/7` geeignet vom kontextfreien auf den DCG-Fall erweitert werden. Das geschieht durch Änderung von `instance/2`.

Können die Ausführungen der Ziele durch Unifikationen die Parsetabelle verderben?. Solange alle Instruktionen zur Vorberechnungszeit auswertbar sind, sollte mindestens die Berechnung der Löschbarkeit stimmen, vielleicht auch die der Kettenregeln (falls keine eingebettete Konjunktion verwendet wird). Für den allgemeinen Fall braucht man vielleicht eine Form der partiellen Auswertung, um die Bedingungen `{Goal}` besser zu berücksichtigen.

6.1 Berechnung und Überprüfung der verwendeten Kategorien

6.1.1 Vorberechnung der lexikalischen Kategorien

Es ist sinnvoll, bei den Vorberechnungen auch eine Tabelle `lexCat/1` der lexikalischen Kategorien der Grammatik anzulegen. Als lexikalische Kategorien werden hier alle Kategorien `A` betrachtet, für die *in der ausgewählten Grammatik* keine Syntakregeln `A --> B` definiert sind.

Mit `lexCat/1` 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 (vgl. `erkenneWortarten/0`). Sofern die Grammatik eine Deklaration `wortkategorie/1` enthält, kann man außerdem mit `lexCat/1` auch prüfen, ob alle in den Regeln verwendeten lexikalischen Kategorien ihren Deklarationen nach `wortkategorie/1` entsprechen und ggf. ein Restriktor dafür definiert ist.

⟨*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) :-
        var(C),!,collect_lexCats(Cs,Cats).
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,Cats).
collect_lexCats([C|Cs],Cats) :-
        !,collect_lexCats(Cs,CsCats),
        (has_syntaxrules(C) -> Cats = CsCats ; Cats = [C|CsCats]).
collect_lexCats([],[]).

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)).
```

Der folgende Test, ob es zu einer Kategorie eine eigentliche, d.h. nicht-löschende, Syntaxregel gibt und damit die Kategorie keine lexikalische Kategorie ist, ist etwas ungenau:

⟨*Lexikalische Kategorien*⟩+≡

```
has_syntaxrules(C) :-
        \+ \+ ((C --> X),(not(X = []), not(X = ([],{_Test})))).
```

Sowohl die Merkmale, die zur Löschung von `C` berechtigen, wie auch anders aufgebaute rechte Regelseiten, die auf eine Löschung hinauslaufen, müßte man berücksichtigen.

Bemerkung 6.1 Zusätzlich sollte man vielleicht die Kategorien `C` als lexikalische Kategorien zählen, zu denen es nur Kettenregeln `C --> L` gibt, wo `L` eine lexikalische Kategorie (oder `[]`) ist. Mit ihnen könnte man Wortarten aufspalten (z.B. `a(Art,Form) --> ordinalzahl ; 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.

6.1.2 Vorberechnung der syntaktischen Kategorien

Wir halten auch die syntaktischen Kategorien `C` in einem Prädikat `synCat/1` fest, um damit überprüfen zu können, ob alle syntaktischen Kategorien in den Grammatikregeln auch den Deklarationen der Kategorien entsprechen.

⟨*Syntaktische Kategorien*⟩≡
```
% ------- Berechnung der syntaktischen Kategorien -----
:- dynamic synCat/1.

compute_synCat :-
        retractall(synCat(_)),
        setof(A,B^((A --> B),not(B = []),not(B = ([],{_Test}))),SynCats),
        add_SynCats(SynCats).

add_SynCats([C|Cs]) :-
        add(synCat(C)),
        add_SynCats(Cs).
add_SynCats([]).
```

Bem. Die Prädikate `lexCat/1` und `synCat/1` sind ungenau, da sie freie Variable für Merkmale enthalten können; außerdem fallen darunter nicht nur die allgemeinsten Fakten. (Man sollte die (zur Zeit der Grammatikübersetzung auswertbaren) Bedingungen in den Regeln auswerten und die genauen Merkmalwerte ebenfalls auf ihre Übereinstimmung mit den Deklarationen überprüfen.)

6.1.3 Überprüfung der in den Regeln verwendeten Kategorien

Da Prolog keine Typen hat, prüfen wir mit den folgenden Tests, ob alle in den Grammatikregeln verwendeten (atomaren) Kategorien den Deklarationen der lexikalischen und syntaktischen Kategorien entsprechen:

⟨*Kategorienprüfung*⟩≡
```
pruefe_lexCats :-
        lexCat(C),
        (wortkategorie(C) -> true
        ; write(user,'\nWarnung: nicht deklarierte terminale Kategorie: '),
           write(user,C)
        ),
        fail.
pruefe_lexCats.

pruefe_synCats :-
        synCat(C),
        (kategorie(C) -> true
```

```
        ; write(user,'\nWarnung: nicht deklarierte syntaktische Kategorie: '),
          write(user,C)
        ),
        fail.
pruefe_synCats.
```

6.1.4 Vereinfachung Boole'scher Bedingungen und zusammengesetzter Kategorien

Bei der Vorberechung der Hilfstabellen aus bedingten Regeln `(A --> B) :- C` kann durch Belegung von Variablen die Bedingung `C` variablenfreie Teilbedingungen `D` enthalten. Wenn die Hilfstabellen durch *Regeln* beschrieben werden sollen, kann man darin statt der Bedingungen `C` einfachere Bedingungen verwenden, die durch Auswertung der geschlossenen Teile `D` (zur Vorberechnungszeit) entstehen:

⟨*Vereinfachung von Bedingungen*⟩≡
```
% ------- Vereinfachung Boole'scher Bedingungen --------

simplify((A,B),C) :-
        !,simplify(A,AS),
        (ground(AS)
        -> (AS = true -> simplify(B,C) ; C = fail)
        ; simplify(B,BS),
          (ground(BS)
          -> (BS = true -> C = AS ; C = fail)
          ; C = (AS,BS)
          )
        ).
simplify((A;B),C) :-
        !,simplify(A,AS),
        (ground(AS)
        -> (AS = true -> C = true ; simplify(B,C))
        ; simplify(B,BS),
          (ground(BS)
          -> (BS = fail -> C = AS ; C = true)
          ; C = (AS;BS)
          )
        ).
simplify(A,C) :-
        ground(A),
        !, (call(A) -> C = true ; C = fail).
simplify(A,A).
```

Diese Vereinfachung von Bedingungen wird nur für die Vorberechnung verwendet; zur Parsezeit braucht man `simplify/2` nicht mehr. Dasselbe gilt für die darauf aufbauende Vereinfachung von Kategorien:

⟨*Vereinfachung von Kategorien*⟩≡
```
% ------- Vereinfachung von zusammengesetzten Kategorien --------

simplify_cat(A,A) :-
        var(A), !.
simplify_cat({A},{SA}) :-
        !, simplify(A,SA).
```

```
simplify_cat((A,B),S) :-
        !, simplify_cat(A,SA), simplify_cat(B,SB),
        ((nonvar(SA), SA = {true})
        -> S = SB
        ; (nonvar(SB), SB = {true})
        -> S = SA
        ; S = (SA,SB)
        ).
simplify_cat((A;B),S) :-
        !, simplify_cat(A,SA), simplify_cat(B,SB),
        ((nonvar(SA), SA = {true})
        -> S = SB
        ; (nonvar(SB), SB = {true})
        -> S = SA
        ; S = (SA;SB)
        ).
simplify_cat([],[]) :- !.
simplify_cat([W|Ws],[W|Ws]) :- !.
simplify_cat(A,A).   % kann man noch etwas f"ur Optionale ?A tun?

simplify_cats([],[]) :- !.
simplify_cats(Cs,SCs) :-
        listToTuple(Cs,Cat),
        simplify_cat(Cat,SCat),
        tupleToList(SCat,SCs).

listToTuple([A],(A)) :- !.
listToTuple([A|As],(A,Bs)) :- listToTuple(As,Bs).
```

Mit dieser Vereinfachung von Kategorien werden bei den Vorberechnungen die rechten Regelseiten vereinfacht, wenn dort Instruktionen `{Goal}` ausgewertet und dadurch zur Parsezeit z.T. nicht mehr gebraucht werden.

6.2 Berechnung von Hilfsrelationen zwischen den Kategorien

6.2.1 Vorberechnung der löschbaren Kategorien

Aus den DCG-Regeln der Grammatik wird eine Tabelle `del/1` der löschbaren Kategorien

$$\{A \mid A \Rightarrow^{+} \epsilon\}$$

gebildet. Dazu setzen wir voraus, daß alle Bedingungen `Cond` von Löschregeln `A --> B :- Cond.` oder `A --> B, {Cond}.` zur Vorberechnungszeit auswertbar sind, was plausibel ist.

⟨*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:[15]

⟨*Löschbare Kategorien*⟩+≡
```
compute_del_basecase :-
    (clause(A --> [],Cond) ; clause(A --> ([],{Cond}),true)),
    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.
```

Der Löschbarkeitstest folgt dem Aufbau der Kategorienausdrücke einer DCG-Regel. 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. (Später werden *Unbestimmte*, als nicht löschbar verstandene Komplemente durch Variable dargestellt; dafür dient der erste Fall.)

⟨*Löschbare Kategorien*⟩+≡
```
deletable(A) :-
    var(A), !, fail.
deletable((A,B)) :-
    !,deletable(A),deletable(B).
deletable((A;B)) :-
    !,(deletable(A);deletable(B)).

deletable(?(A;B)) :-        % Testhalber f"ur Optionen ?(dat;akk) im Komplementrahmen
        !, deletable(?A) ; deletable(?B).
deletable(?(A,B)) :-
```

[15] Andere Formen von Löschregeln als `(A --> []) :- Cond.` und `(A --> [], {Cond}).` werden *nicht* berücksichtigt; man sollte das Grammatikformat in diesem Sinn normieren.

```
        !, deletable(?A) , deletable(?B).

deletable(?C) :- !, del(?C).
deletable({Goal}) :- !, call(Goal).
deletable([_A|_B]) :- !, fail.
deletable([]) :- !, true.
deletable(A) :- !, del(A).
```

Beachte, daß nicht nur durch `call(Goal)`, sondern auch durch die Aufrufe `del(A)` freie Variable in den Kategorienausdrücken belegt werden können, da zu `del/1` nicht nur Fakten, sondern Klauseln gebildet wurden. Daher wird durch `deletable(A)` meist eine Grundinstanz von `A` erzeugt; offene Instanzen erhält man nur, wenn die Löschregeln `(A -> []) :- C` unbeschränkte freie Variable enthalten.

6.2.2 Vorberechnung der Kettenbeziehung

Als nächstes wird eine Tabelle `chain/2` der durch Kettenableitungen erreichbaren Kategorien

$$\{(A, B) \mid A \Rightarrow^+ B\}$$

gebildet; die Paare `(A,A)` werden später separat behandelt. Neben der Grammatik muß hierfür auch die Tabelle `del/1` der löschbaren Kategorien geladen sein.

$\langle$*Kettenbeziehung*$\rangle\equiv$

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

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

Zuerst werden die Paare (A, B) gesammelt, für die es eine Regel $A \to \alpha B \beta$ mit löschbaren α und β gibt. Alle Regeln `(?A --> B)` mit optionaler Kategorie auf der linken Seite werden ignoriert, da diese in den Parsern implizit behandelt werden. Fälle, wo rechts nur eine Instruktion `{Goal}` steht, braucht man ebenfalls nicht zu behandeln:

$\langle$*Kettenbeziehung*$\rangle{+}\equiv$

```
compute_chain_base :-
    clause(A --> Body,Cond),
    not(A = ?(_)),                  % ignoriere Regeln für optionale Konstituenten
    tupleToList(Body,BodyL),
    append(Beta,[B|Gamma],BodyL),    % ineffizient: nach 2 nicht löschbaren aufhören
    (B = {Goal}
    -> true
    ; (var(B)
      -> write(user,'\nWarnung: variable Kategorie; ignoriert')
      ; deletables(Beta),
        deletables(Gamma),
        simplify(Cond,NewConds),
        addChains(A,B,NewConds)
      )
    ),
    fail.
```

```
compute_chain_base.

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

Falls die Grammatik direkte, d.h. nicht durch Löschungen von Konstituenten entstehende, Kettenregeln hat, 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,B,Cond) :-
        var(B),!.  % tritt nicht auf, da Koepfe nicht loeschbar sind.
addChains(A,(C,D),Cond) :-
        !, ( deletable(D), simplify(Cond,NewCond), addChains(A,C,NewCond)
           ; deletable(C), simplify(Cond,NewCond), addChains(A,D,NewCond)
           ).
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) :-
        !.
addChains(_A,{_Goal},_Cond) :-
        !. % tritt nicht auf.
addChains(A,B,Cond) :-
        (A == B -> true ; simplify(Cond,NewCond), add(chain(A,B) :- NewCond)).
```

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),{C}` sind. Da außer in den hier irrelevanten Löschregeln `?D --> [] :- C.` 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_itc :-
    findall((X,Y),itc(chain,X,Y),Chains),
    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 8.1.1 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_itc :-
    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 `first_base(restrict)` vereinfachen und mit `predictor(uniform)` den uniformen statischen Predictor benutzen (s.u.).

Beispiel 6.2 Zum Testen der Berechnung der Kettenableitungen in DCGs betrachten wir

⟨*Grammatiken/dcg.3.grm*⟩≡
```
:- op(1100,xfx,'-->').

start_symbol(s(X)).

auxiliary_predicates([start_symbol/1]).

s(X) --> a(X), (b(X) ; d(X), {X=3}) :- member(X,[1,2,3]).
b(2) --> [].
d(X) --> b(X), c(X).

word(a,a(1)).     word(b,b(1)).     word(c,c(1)).
word(a,a(2)).     word(b,b(2)).     word(c,c(2)).
word(a,a(3)).     word(b,b(3)).     word(c,c(3)).
```

Man erhält folgende Tabellen:

⟨*Vorberechnete Tabellen*⟩≡
```
del(b(2)).            chain(s(2), a(2)).     chain(d(2), c(2)).

first(a(1), s(1)).    first(a(2), s(2)).     first(a(3), s(3)).
first(b(A), d(A)).    first(c(2), d(2)).
```

6.2.3 Vorberechnung der First-Relation

Eine für verschiedene Parser nützliche Relation zwischen Kategorien ist

$$\mathit{first} := \{(B,A) \mid \exists\beta\ A \Rightarrow^{+} B\beta\}.$$

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

Zuerst wird das Ergebnis einer vorherigen Berechnung von `first/2` gelöscht, dann bestimmt man die aus den einzelnen Regel direkt ablesbare Grundrelation

$$\mathit{first\_base} := \{(B,A) \mid (A \rightarrow \alpha B\beta) \in P,\ \alpha \Rightarrow^{*} \epsilon\},$$

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(_,_)),
        compute_first_base,
        compute_first_itc.
```

Die Grundrelation kann auf verschiedene Weisen dargestellt werden: durch (schematische) Fakten, mit der (Default-) Option `first_base(explicit)`, oder durch Regeln, mit der Option `first_base(rules)`:

⟨*Vorberechnung der First-Relation*⟩+≡

```
compute_first_base :-
        (ein(verbose) -> write('Base relation:'),nl ; true),
% pl 4.0.2 hat die Flags noch nicht:
%         (prolog_flag(debug,true)
%         -> write(user,'\n *** vorhandener Restrictor: ***\n'),
%            listing(restrict)
%         ; true),
        (first_base(rules)
        -> clause(A --> Alpha, Cond),
           collect_first_base(A,Alpha,Cond)
        ; (A --> Alpha),
           collect_first_base(A,Alpha)
        ),
        fail.
compute_first_base.
```

Beim Sammeln der Tupel durch `collect_first_base/2` und `collect_first_base/3` müssen die rechten Regelseiten zerlegt werden, bis man auf eine `[Wortfolge]` oder einen einfache Kategorie stößt, damit auch (Pseudo-) DCG-Regeln behandelt werden können.

Fall 1 Bei der Option `first_base(explicit)`, dem Default, werden durch `collect_first_base(A,Alpha)` 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[16], kann man Tupel `(A,?B)` und `(A,(B;C))` aus der Relation herauslassen. Da `(A;B)` nicht als Kopf einer Regel auftritt,

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

kommen keine Tupel `((A;B),C)` in der Relation vor. Damit man `?A` als Kopf von Löschregeln erlauben kann, nehmen wir die Tupel `(?A,B)` nicht in die Grundrelation auf; es genügt, später die Tupel `(?A,A)` mit `instance/2` zu behandeln.

⟨*Vorberechnung der First-Relation*⟩+≡

```
collect_first_base(_A,B) :-
        var(B),!,
        write(user,'\nFEHLER: variable Kategorie; ignoriert!\n'). % Fehler
collect_first_base(?A,_) :- !.
collect_first_base((A;B),_) :- !.

collect_first_base(A,(B,C)) :-
        !,collect_first_base(A,B),
        deletable(B),
        collect_first_base(A,C).
collect_first_base(A,(B;C)) :-
        !,
        collect_first_base(A,B),
        collect_first_base(A,C).
collect_first_base(A,?(B)) :-
        !,
        collect_first_base(A,B).
collect_first_base(_A,[]) :-
        !.
collect_first_base(_A,{_Goal}) :-
        !.
```

(Bei `collect_first_base(A,(B,C))` muß `collect_first_base(A,C)` für *jede* Belegung aufgerufen werden, die `deletable(B)` beweisbar macht.)

Fall 2 Da im allgemeinen die (unten angegebene) 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. Durch das Lexikon und die Grammatik muß dazu ein Prädikat `restrict/2` definiert werden. (vgl. Abschnitt 6.2.4.)

⟨*Vorberechnung der First-Relation*⟩+≡

```
collect_first_base(A,[B|Rest]) :-
        !,(first_base(restrict)
          -> restrict_all([A,B|Rest],[AR,BR|RestR]), add(first([BR|RestR],AR))
          ; add(first([B|Rest],A))).
collect_first_base(A,B) :-
        !,(A == B -> true
          ; (first_base(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.

Fall 3 Für die Option `first_base(rules)` werden zur Darstellung der Grundrelation `first/2` Prolog-*Regeln* in die Datenbank geschrieben (wobei die Restriktion keine Rolle spielt, d.h. ignoriert wird):

⟨*Vorberechnung der First-Relation*⟩+≡

```
collect_first_base(A,B,Cond) :-
        var(B),!,
        write(user,'\nWarnung: variable Konstituentenkategorie; werte Bedingungen aus. \n'),
        call(Cond),
        collect_first_base(A,B).
collect_first_base(?A,_,_Cond) :-
        !.
collect_first_base((A;B),_,_Cond) :-
        !.
collect_first_base(A,(B,C),Cond) :-
        !, collect_first_base(A,B,Cond),
        deletable(B),
        simplify(Cond,NewCond),
        collect_first_base(A,C,NewCond).
collect_first_base(A,(B;C),Cond) :-
        !, collect_first_base(A,B,Cond),
        collect_first_base(A,C,Cond).
collect_first_base(A,?(B),Cond) :-
        !, collect_first_base(A,B,Cond).
collect_first_base(_A,[],_Cond) :-
        !.
collect_first_base(_A,{_Goal},_Cond) :-
        !.
collect_first_base(A,[B|Rest],Cond) :-
        !, simplify(Cond,NewCond),
        add((first([B|Rest],A) :- NewCond)).
collect_first_base(A,B,Cond) :-
        !,(A == B -> true ; simplify(Cond,NewCond), add((first(B,A) :- NewCond))).
```

Bemerkung 6.3 (Berechnung für DCG-Regeln) Um in den rechten Seiten der Regeln `A --> Gamma` komplexe „Kategorien“ erlauben zu können, insbesondere Instruktionen `{Goal}`, die keiner Konstituenten der Eingabe entsprechen, werden mit `deletables(Alpha)` löschbare Präfixe `Alpha` von `Gamma` gesucht; dabei werden durch Auswertung von Instruktionen `{Goal}` in `Alpha` auch Variablen belegt. (Das setzt voraus, daß diese `{Goal}` zur Zeit der Vorberechnung und zur Parsezeit dasselbe liefern!) Für die so spezialisierten Regeln wird dann die auf `Alpha` folgende Kategorie `B` rekursiv zerlegt und ihre atomaren Teile `C` als mögliche Kategorien am Anfang von `A`-Phrasen als `first(C,A)` gespeichert.

(??) Durch die Berechnung der transitiven Hülle werden dann (aber) die durch Auswerten von `Cond` erhaltenen Tupel mit konkreten Merkmalen benutzt; das macht die First-Relation genauer, aber weniger schematisch und daher (vermutlich) zeitaufwendiger bei der Benutzung.

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.[17]

[17] 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 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_itc :-
    (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).
```

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 `X=Y; first(X,Y)` erreicht.

6.2.4 Überprüfung der Restriktion der Kategorien

Die zu berechnenden Erreichbarkeitsrelationen (`first/2` und `chain/2`) zwischen den Kategorien können so groß werden, daß man sie nicht mehr berechnen kann oder daß ihre Verwendung zur Parsezeit zu langsam ist. Dann muß man die Kategorienausdrücke vereinfachen, indem man Teilausdrücke durch Variable ersetzt, und mit Erreichbarkeitsrelationen zwischen diesen (schematischen) Kategorien arbeiten. Diese Vereinfachung erfolgt durch ein Prädikat `restrict(Kategorie+,Vereinfachung-)`, das in der Grammatik für jede Kategorie unterschiedlich gemacht wird. (Z.B. für die Satzgrammatik in den Dateien `wortarten.cat.pl` und `np.cat.pl` usw., die zu den Teilgrammatiken wie `np.grm` gehören. Daher ist `restrict/2` genau wie die Kategoriendefinitionen *nicht* unter den `auxiliary_predicates`.)

Beim Erzeugen eines `.epl`- oder `.glp`-Parsers prüfen wir mit der folgenden Funktion, ob jede in der Grammatik benutzte Kategorie eine Einschränkung durch `restrict/2` hat; für Kategorien `Cat`, die keinen Restriktor haben (bzw. nicht mit einer Kategorie unifizieren, die einen Restriktor hat), wird eine Warnung ausgegeben und eine Default-Klausel `restrict(Cat,Cat)` ergänzt:

⟨*Vorberechnung der First-Relation*⟩+≡

```
pruefe_restrictor :-
        write(user,'\n***** Prüfe, ob alle lexCats eine Restriktion haben.\n\n'),
        lexCat(Cat),
        (restrict(Cat,_) -> true
        ; write(user,'Warnung: '), write(user,Cat),
            write(user,' hat keinen Restriktor; Default ergänzt.\n'),
            assertz(restrict(Cat,Cat)) % Kein ! ergaenzen!
        ),
        fail.
pruefe_restrictor :-
        write(user,'\n***** Prüfe, ob alle synCats eine Restriktion haben.\n\n'),
        synCat(Cat),
        (restrict(Cat,_) -> true
```

```
        ; write(user,'Warnung: '), write(user,Cat),
          write(user,' hat keinen Restriktor; Default ergänzt.\n'),
          assertz(restrict(Cat,Cat))
        ),
        fail.
pruefe_restrictor.
```

Wegen der automatisch ergänzten Default-Klauseln ist es *nicht zwingend*, in jeder Grammatik ein Prädikat `restrict/2` zu definieren. Bei einer kleinen Anzahl von Kategorien genügen die ergänzten Default-Klauseln. Für Grammatiken mit vielen (expliziten) Kategorien werden so aber die Hilfstabellen zu groß, und man muß einen eigenen Restriktor definieren. (Falls man keinen Restriktor braucht, sollte man einfach in der Grammatik eine Default-Klausel `restrict(X,X).` angeben.)

Bem. Falls der Restriktor zu wenig Tupel enthält, wird `first/2` zu klein und man verliert Analysen! Die Warnungen zu fehlenden Restriktionen erscheint auch dann, wenn die entsprechende Kategorie und ihr Restriktor nur in einer temporär nicht geladenen Teilgrammatik definiert werden. Dann kann man die Warnung ignorieren.

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

Bemerkung 6.4 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.

Damit die Suche nach einer Restriktion nach dem ersten passenden Fall abgebrochen werden kann, sollten die vom Grammatikschreiber angegebenen `restrict`-Klauseln von der Form

⟨*Restriktor-Klauseln*⟩≡
```
restrict(Cat,Cat') :- bedingung(Cat), !.
```
sein.

Problem: Wenn die vom Programmierer angegebenen Restriktionen unvollständig war, werden nur Klauseln der Form `restrict(Cat,Cat).` ergänzt, also ohne `!`, und das für jedes `Cat`, für das es noch keine Restriktion gab. Diese Klauseln schließen einander *nicht* aus. Im Prinzip reichte es, stattdessen einen globalen Sonst-Fall `restrict(X,X).` mit Kategorien*variable* `X` zu erzeugen, aber dann bekäme man zu allen weiteren Kategorien ohne vordefinierten Restriktor keine Fehlermeldung mehr.

Man darf aber nicht Klauseln der Form `restrict(Cat,Cat) :- !.` ergänzen: zu jeder Grammatikregel `(A --> B,...) :- C.` wird durch `collect_first_base(A,B)` u.a. mit einem Aufruf `restrict(B,BR)` getestet, ob `B` einen Restriktor hat. Die freien Variablen der Kategorie `B` würden aber bei ergänzten `restrict(Cat,Cat) :- !.` auf die Instantiierung festgelegt, die zum *ersten* solchen `Cat` paßt. Andere Belegungen (die nach `C` erlaubt sind), werden weggeschnitten, so daß der Restriktor zu klein wird.

(Tritt ein ähnliches Problem nicht auch dann auf, wenn keine `restrict`-Klauseln erzeugt werden – immerhin benutzen die anderen Restriktionsklauseln jeweils auch einen `!` zur Einschränkung?)

6.2.5 Vorberechnung brauchbarer Regeln

Der Parser muß bei der Analyse einer Eingabe feststellen, mit welchen Grammatikregeln welche Teilausdrücke der Eingabe zusammengefaßt werden können. Statt dabei jeweils alle Regeln zu durchlaufen, sollte man die Regeln ignorieren, die in der jeweiligen Situation ungeeignet sind, und nur eine ausreichend große Teilmenge von 'möglicherweise noch brauchbaren" Regeln durchlaufen. Ob eine Regel zu den vielleicht noch brauchbaren gehört, hängt ab von

(i) den Erwartungskategorien für Präfixe der Resteingabe,

(ii) den lexikalischen Kategorien der noch nicht verarbeiteten Eingabewörter, insbesondere den lexikalischen Kategorien des nächsten Eingabewortes,

(iii) den lexikalischen Kategorien unter denen der rechten Regelseite,

(iv) der Kategorie der linken Regelseite.

Für jede Erwartungskategorie und jede (Folge von) lexikalischen Kategorie(n) kann man eine solche Teilmenge „brauchbarer" Regeln *statisch*, d.h. vor der Analyse einer Eingabe, berechnen und abspeichern und diese *dynamisch*, d.h. zur Parsezeit, in den Situationen während der Analyse durchsuchen, in denen die Erwartungskategorie tatsächlich eine der Erwartungen ist und die lexikalischen Kategorien tatsächlich die lexikalischen Kategorien der Restwörter umfassen.

Zu ihrer Berechnung werden die vorher berechneten Tabellen `del/1` und `first/2` verwendet. Außerdem werden drei unter den Hilfsfunktionen des `.epl`-Parsers definierte Relationen benutzt, nämlich `instance/2`, `connect/4` und `leftcorner-rtc/2`:

⟨*Vorberechnung brauchbarer Regeln*⟩≡
  ⟨*instance*⟩
  ⟨*connect*⟩
  ⟨*leftcorner-rtc*⟩

Mit ihr werden (auch beim Parsen) an einer disjunktiven oder optionalen Kategorie `C` ihre Spezialisierungen `D` abgelesen, so als gäbe es eine Regel `C --> D.`, für `(A;B) --> A.`, `(A;B) --> B.`, und `?(A) --> A.`. Eine partielle Behandlung zusammengesetzter Kategorien `(A,B)` ist ebenfalls darunter.

⟨*Vorberechnung brauchbarer Regeln*⟩+≡
```
% --- Bei Erwartung C und Wortkategorie LexCat brauchbare Regeln ---
:- dynamic predicted/4.
:- dynamic predicted/6.
:- dynamic predicted/7.

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

Wir unterscheiden zwischen „explizit", d.h. für jede Situation speziell, und „uniform", d.h. für alle Situationen gleich, vorberechneten Regelmengen, die erst zur Parsezeit auf die Situation spezialisiert werden:[18]

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
comp_static_predictor :-
        (predictor(uniform) -> predict_static_uniform
        ; ((parser(earley_pratt_leiss),
            first(LexCat,E),lexCat(LexCat),
            predict_static(E,LexCat),
            fail)
          ;(parser(leftcorner_generalized),
            first(D,E),lexCat(LexCat),  % ; D=E ?
            predict_static(E,D,LexCat),
            fail)
          )
        ).
comp_static_predictor.
```

Bem. Die Vorberechnungen der Erreichbarkeitsrelationen ist aber noch zu kontrollieren, da man nun zwischen Konstituenten `B` und Instruktionen `{C}` unterscheiden muß!

Für unterschiedliche Parser sind braucht man unterschiedliche Regelmengen als „brauchbare Regeln". Der Earley-Parser analysiert, mit der vorgegebenen Erwartungskategorie als Startsymbol, die Reseingabe ab Position j wie eine neue Eingabe. Der Leftcorner-Parser analysiert, ebenfalls mit der vorgegebenen Erwartungskategorie als Startsymbol, die Resteingabe als Fortsetzung einer vollständigen Phrase. Im allgemeinen sind dafür andere Regeln brauchbar.[19]

Explizite, für jeden Situationstyp speziell berechnete Regelmenge

(i) Für den Earley-Parser werden zu einer Erwartungskategorie `C` und einer lexikalischen Kategorie `LexCat` des nächsten Eingabewortes die Regeln `A -> Alpha B Beta` als brauchbar vorausgesagt, bei denen `A` eine Anfangskategorie von `C`-Phrasen sein kann, `Alpha` löschbar ist, und `LexCat` eine Anfangskategorie von `B`-Phrasen sein kann.

Die bei der Erwartungskategorie E und der Wortkategorie L *brauchbaren Regeln* sind also

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

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

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
predict_static(E,LexCat) :-
        (ein(verbose) -> nl,write('static predictor (explicit) for '),write((E,LexCat)),nl ; true),
        setof((A,Alpha,[B,B2|Beta],AlphaTrees),
              Alphas^As^((A=E ; first(A,E)), % first(A,E) kann man nicht sparen!
                         (A --> Alphas),
                         tupleToList(Alphas,As),
```

[18] Außerdem kann man die bedingten Grammatikregeln oder deren Instanzen abspeichern, s.u. Die Genauigkeit der vorberechneten Regelmenge muß mit dem Aufwand zum Durchsuchen abgewogen werden: schematischere Regeln sind ggf. schneller zu durchsuchen.

[19] Gewinnt man etwas, wenn man mehr als *eine* vollständige Phrase als Linkskontext berücksichtigt?

```
                        append(Alpha,[B,B2|Beta],As), % erzwingt Verzweigung, braucht chain/2.
                        deletables(Alpha),
                        leftcorner_rtc(LexCat,B),
                        canonicalTrees(Alpha,AlphaTrees)),

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

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

(ii) Für den Leftcorner-Parser dienen die vorausgesagten Regeln zur Erweiterung schon vollständig erkannter Phrasen. In den als brauchbar vorausgesagten Regeln `A -> Alpha B Beta` soll nicht die lexikalische Kategorie `LexCat` des nächsten Eingabewortes eine Anfangskategorie von `B`-Phrasen sein, sondern eine (atomare) Kategorie `D` einer vollständig erkannten Phrase soll von `B` durch eine Kettenableitung erreichbar sein, und `LexCat` muß am Anfang einer Folge von `Beta`-Phrasen stehen können.

Die bei der Erwartungskategorie E, der Kategorie D einer vollständigen Phrase unmittelbar vor der Resteingabe, und der lexikalischen Kategorie L des nächsten Eingabewortes brauchbaren Regeln sind also:

$$\mathit{predicted}(E, D, L) := \{A \to \alpha B \beta_1 C \beta_2 \mid E \Rightarrow^* A\gamma,\ \alpha \Rightarrow^* \epsilon,\ B \Rightarrow^* D, \beta_1 \Rightarrow^* \epsilon,\ C \Rightarrow^* L\delta\}$$

Sie werden durch `predict_static(E,D,L)` berechnet:

$\langle$ *Vorberechnung brauchbarer Regeln* $\rangle{+}{\equiv}$

```
  predict_static(E,D,LexCat) :-
          (ein(verbose)
          -> nl,write('static predictor (explicit) for '),write((E,D,LexCat)),nl
          ; true),
          setof((A,SimplAlpha,[B|SimplBeta1CBeta2],AlphaTrees),
                C^Alphas^As^((A=E ; first(A,E)),
                        (A --> Alphas),
                        tupleToList(Alphas,As),
                        % erzwingt Verzweigung (noch verhindern: B2 = {..} ?)
                        append(Alpha,[B,B2|Beta],As),
                        deletables(Alpha),
                        connect(B,D),
                        append(Beta1,[C|Beta2],[B2|Beta]),
                        deletables(Beta1),
                        leftcorner_rtc(LexCat,C),
                        simplify_cats([B2|Beta],SimplBeta1CBeta2),
                        simplify_cats(Alpha,SimplAlpha),
                        canonicalTrees(SimplAlpha,AlphaTrees)),
              Rules),
          (ein(verbose)
          -> write('Vorausgesagte Regeln: '),length(Rules,Lg),write(Lg),nl
          ; true),
          add_predicted(E,D,LexCat,Rules).
```

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

Beim Test auf Löschbarkeit werden Instruktionen `{Goal}` ausgeführt; sie können danach beseitigt werden, sofern sie keine freien Variablen mehr enthalten[20]. Das geschieht mit `simplify_cats/2` aus Abschnitt 6.1.4. (Allerdings werden die Bedingungen mit freien Variablen hier noch als gelöschte Konstituenten dargestellt.)

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

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

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

Da bei größeren Grammatiken sehr viele Paare `(C,LexCat)` bzw. `(C,D,LexCat)` auftreten, wird diese Tabelle der brauchbaren Regeln ein Vielfaches der Grammatikregeln und zu groß.

Uniforme, für alle Situationen gleich berechnete Regelmenge

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 zu berechnen (s.u. `predict_static/0`). Zur Parsezeit werden dann die für die jeweilige Erwartung `C`, ggf. die Kategorie `D` einer erkannten Phrase, und die lexikalische Kategorie `LexCat` des nächsten Worts passenden Regeln ausgewählt:

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
predict_static_uniform :-
        (ein(verbose) -> nl,write('static predictor (uniform)'),nl ; true),
        setof((A,Alpha,[B,B2|Beta],AlphaTrees,NewCond),
              Alphas^As^Cond^(clause(A --> Alphas,Cond),
                         tupleToList(Alphas,As),
                         append(Alpha,[B,B2|Beta],As),
                         deletables(Alpha),
                         simplify(Cond,NewCond),
                         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*:[21]

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
add_predicted([(A,Alpha,[B,B2|Beta],AlphaTrees,Cond)|Rules]) :-
```

[20] Das spart Parsezeit. Die `{Goal}`, die noch freie Variablen enthalten, können erst zur Parsezeit ausgewertet werden

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

```
        (ein(predictUniform)
        -> (  parser(earley_pratt_leiss),
               (var(B) % Metaregel: erst B durch vorheriges Auswerten von Cond belegen!
               -> add(predicted(E,LexCat,A,Alpha,[B,B2|Beta],AlphaTrees)
                     :- ((A = E ; first(A,E)),
                            Cond, leftcorner_rtc(LexCat,B)))
               ; add(predicted(E,LexCat,A,Alpha,[B,B2|Beta],AlphaTrees)
                    :- ((A = E ; first(A,E)),
                           leftcorner_rtc(LexCat,B), Cond))
               )
            ; parser(leftcorner_generalized),
               add(predicted(E,D,LexCat,A,Alpha,[B,B2|Beta],AlphaTrees)
                  :- ((A = E ; first(A,E)), Cond))
           )
        ; add_predicted_instances((A,Alpha,[B,B2|Beta],AlphaTrees,Cond))),
        add_predicted(Rules).
add_predicted([]).
```

Bem. Für den Leftcorner-Parser wird die Nebenbedingung `Cond` der Grammatikregel ausgewertet, *bevor* zur Parsezeit ein Vergleich von `D` mit `B` und von `LexCat` mit `[B2|Beta]` gemacht wird. Das ist für den Fall der Metaregeln, wo `B` eine Variable sein kann, *notwendig*, um `B` zu belegen; in den anderen Fällen könnte man diese Vergleiche auch zuerst auswerten, aber das scheint keinen Vorteil zu haben.

Ohne die Option `ein(predictUniform)` 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: (*für Metaregeln anpassen!*)

⟨*Vorberechnung brauchbarer Regeln*⟩+≡

```
add_predicted_instances((A,Alpha,[B,B2|Beta],AlphaTrees,Cond)) :-
        (parser(earley_pratt_leiss),
            (A = E ; first(A,E)),
            leftcorner_rtc(LexCat,B),
            lexCat(LexCat),
            add(predicted(E,LexCat,A,Alpha,[B,B2|Beta],AlphaTrees) :- Cond)
        ; parser(leftcorner_generalized),
            (A = E ; first(A,E)),
            connect(B,D),
            append(Beta1,[C|Beta2],[B2|Beta]),
            deletables(Beta1),
            leftcorner_rtc(LexCat,C),
            lexCat(LexCat),
            simplify(Cond,NewCond),
            add(predicted(E,D,LexCat,A,Alpha,[B,B2|Beta],AlphaTrees) :- NewCond)
        ),
        fail.
add_predicted_instances((A,Alpha,[B,B2|Beta],AlphaTrees,Cond)).
```

Zum Parsen mit Metaregeln (in Entwicklung!)

Um Metaregeln benutzen zu können, muß ein Parser sie in eine Menge normaler Syntaxregeln übersetzen oder besondere Instruktionen zum Umgang mit ihnen haben. Wir wollen haupsächlich Metaregeln nach dem $\bar{X}$-Schema benutzen können, d.h. solche der Form

⟨*X-Bar Regelschema*⟩≡

```
A --> ...,XP,..., v([nom,akk,präp],Form),... :- Cond
```

wo die Konstituenten `...,XP,...` durch den Komplementrahmen der Kopfkonstituenten bestimmt sind. Dazu ist zu beachten:

(i) Für die Vorberechnung der Löschregeln kann man annehmen, daß variable Kategorien in den Grammatikregeln *nicht löschbar* sind, d.h. nur durch nicht löschbare Kategorien instantiiert werden können. (Dazu braucht man eine Konvention, nach der löschbare Kategorien stets die Form `?Cat` haben, und optionale Elemente `?X` im Komplementrahmen auch nur zu Kategorien `?XP` expandiert werden – allerdings brauchen wir für diese auch Regeln `?XP --> [] :- Cond`.)

(ii) Bei der Vorberechnung der Kettenregeln kann man variable Konstituenten insofern (irgendwie) behandeln, als nicht alle Konstituenten einer rechten Regelseite Variable sein können, da ein Kopfelement auftritt; daher ist zur Zeit der Grammatikübersetzung feststellbar, ob die rechte Regelseite aus einer einzelnen Konstituente bestehen kann.

(iii) Wenn die linke Ecke einer Grammatikregel eine Variable ist, kann die `first/2`-Relation nur berechnet werden, wenn man zur Übersetzungszeit mindestens diese Ecke durch eine Instanz belegt – aber das so zu machen, daß man nicht eine Aufzählung aller Regelinstanzen bekommt, ist schwierig!

(iv) Bei der Vorberechnung des *uniformen* Prediktors, d.h. wenn man nur die löschbaren Präfixe berechnet, kann man die Variablen als nicht-löschbare Konstituenten annehmen. (Vergröbernd könnte man annehmen, daß `first(LexCat,X)` jeweils gilt.) Für die Vorberechnung des expliziten Prediktors muß man wohl ziemlich viel instantiieren, und das wird eine zu große Grammatik!

Ist `A --> Alpha,B,B2,Beta :- Cond` eine Metaregel, so kann `B` eine Variable sein; dann muß die Bedingung `connect(B,D,_,_,)` scheitern. Um das zu verhindern, werten wir in diesem Fall *zuerst* die Bedingungen `Cond` aus, wodurch (nach der Form der erlaubten Metaregeln) `B,B2,Beta` belegt werden, sodaß dann die Tests `connect/2` und `leftcorner_rtc` ausführbar werden (sollten). (Metaregeln, in denen variable Subkonstituenten komplexer `B` auftreten, werden also nicht behandelt — wären sie überhaupt brauchbar?)

Solange der Parser beim Suchen von Regeln die Variablen `XP` zu Kategorien instantiiert, müßte man auch mit Metaregeln und uniformem Predictor arbeiten können. Aber die Vorberechung muß die Bedingungen soweit auswerten, daß sie die `chain/2`- und `first/2`-Grundrelationen und die Voraussage brauchbarer Regeln berechnen kann.[22]

Da bei der Berechnung `comp_first_base` variable Konstituentenkategorien `B` in `A --> Alpha,B,beta :- C` auftreten können, muß man dann `C` auswerten und dadurch `B` konkret machen.

Bem.: Wenn man variable Kategorien bei `memberLexCats` ignoriert, dann ist der Filter der brauchbaren Regeln zwar weniger genau, aber solange alle in einem Regelschema explizit auftretenden lexikalischen Kategorien unter denen der Resteingabe sind, wird das Regelschema noch als nützlich vorausgesagt.

Es geht (im Prinzip, d.h. bei großem Speicher zur Vorberechnung) bei den Auswahlen

```
 First-Relation: [explicit|rules|restrict, inductive],
Predictor: [explicit|uniform].
```

für die angegeben Grammatik `meta.2`. *Das Parsen mit DCG- und Metaregeln muß weiter getestet werden.* Ob bei der Vorberechnung die Regeln `A --> Alpha,B,Beta :- Cond.` mit Zielen `B = {Goal}` oder Variablen `B` richtig ist, muß ebenfalls noch geprüft werden.

[22] Auch dann könnte man noch keine *Syntaxbäume* für `XP` im Komplementrahmen vorgeben, was etwa zur genaueren Steuerung des Parsens nützlich sein könnte; z.B. Vorgabe von Pro-Formen bei bestimmten Anordnungen im Mittelfeld.

Sollte man im statischen Predictor alle `{Goal}` schon zur Zeit der Berechnung von `predicted/6` auswerten, oder würde das nur dazu führen, daß man den Unterschied zwischen `A -> B, {Cond}.` und `(A -> B) :- Conc.` verwischt?

Folgende Probleme sind z.Zt. noch nicht gelöst:

Aufgabe 6.1 Es sollte möglich sein, alle die *Meta*regeln zur Parsezeit zu ignorieren, deren Kopf (mit teilinstantiiertem Rahmen) nicht unter den lexikalischen Kategorien der Resteingabe vorkommt (vgl. die Option `omitRulesWithUselessLexCat`), ohne die genaue Permutation und Belegung der Komplementkategorien(variablen) auszuwerten. Es wäre also denkbar, nicht unbedingt Regeln, sondern *Metaregeln* vorauszusagen!

Aufgabe 6.2 Wie kann man *Tests an die Konstituentenbäume* als Anwendungsbedingungen von Grammatikregeln implementieren (für Tabellen- oder Bottom-Up-Parser)?

Wie kann man Grammatiken mit solchen Tests auf Eigenschaften von Konstituentenbäumen automatisch in Grammatiken mit Kodierung dieser Eigenschaften durch Merkmale an Kategorienbezeichnungen umrechnen?

6.3 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.

Die Prädikate, die bei der Vorberechnung geändert werden, müssen als `dynamic` erklärt sein; nach der Vorberechnung wird diese Deklaration mit `abolish/2` beseitigt, so daß das aus der übersetzten Grammatik geladene Prädikat `static` ist.[23]

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

⟨*Vorberechnungen*⟩
⟨*Regeltransformation*⟩

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 Restriktion der Kategorien. Dann wird die Grammatik aus der angegebenen Datei `G` geladen:[24]

⟨*Vorberechnungen*⟩≡
```
precomputations(G) :-
        retractall(clause(_)),
        retractall('-->'(_,_)),
        retractall('restrict'(_,_)),
        consult(G),
```

[23] Es bleibt für den Benutzer unklar, wieso `restrict/2` nicht zu den `auxiliary_predicates` gehört. Kann man es ohne Schaden hinzunehmen? (Würde es dadurch `dynamic`??)

[24] Durch das Laden der Grammatik muß auch ein `restrict/2` für die Wortarten definiert werden!

(ii) Die lexikalischen, syntaktischen und die löschbaren Kategorien werden berechnet und als Fakten `lexCat/1`, `synCat/1` und `del/1` festgehalten:

⟨*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),
          compute_synCat,
          (ein(verbose) -> nl,listing(synCat/1) ; true),
```

(iii) Die in den Grammatikregeln verwendenten (atomaren) Kategorien werden auf Zulässigkeit nach den deklarierten Kategorien überprüft. Damit man bei kleinen Grammatiken keine Kategorien deklarieren muß, wird dieser Test nur bei einschalten der Option `pruefeKategorien` durchgeführt:

⟨*Vorberechnungen*⟩+≡
```
          (ein(pruefeKategorien) ->
              nl,write('*****  Checking the categories against their declarations '),nl,
              pruefe_lexCats,nl,
              pruefe_synCats,nl
          ; true),
```

(iv) Es wird ggf. geprüft, ob für jede in den Grammatikregeln verwendeten (atomaren) Kategorien ein Restriktor definiert wurde:

⟨*Vorberechnungen*⟩+≡
```
          ((parser(earley_pratt_leiss) ; parser(leftcorner_generalized))
           -> pruefe_restrictor ; true),
```

Falls für manche Kategorien kein Restriktor definiert ist, kann die nachfolgend berechnete Relation `first/2` zu klein sein, so daß man Analysen verliert.

(v) Für die verallgemeinerten Earley- und Leftcorner-Parser wird die Ketten- und die `first`-Relation zwischen Kategorien berechnet:

⟨*Vorberechnungen*⟩+≡
```
          ((parser(earley_pratt_leiss) ; parser(leftcorner_generalized))
          -> nl,write('*****  Computing the chain-relation ... (wait) '),nl,nl,
              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),
```

(vi) Für die verallgemeinerte 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),
```

Für den verallgemeinerten Leftcorner-Parser werden die vorhersagbaren Grammatikregeln als `predicted/7` vorberechnet, für jedes Tripel von möglicher Erwartungskategorie, Phrasenkategorie und lexikalischer Kategorie; hieraus wird zur Analysezeit eine Teilmenge ausgesucht:

⟨*Vorberechnungen*⟩+≡
```
          (parser(leftcorner_generalized)
          -> compute_predictable_rules,
             (ein(verbose) -> nl,listing(predicted/7) ; true)
          ; true),
```

(vii) 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),
```

(viii) Unter neuen Dateinamen `<Datei>.grm.<Parsername>.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_doerre_dcg)
          -> atom_concat(BaseG,'.grm.eddcg.pl',PrepGrm) ; true),
          (parser(earley_pratt_leiss)
          -> atom_concat(BaseG,'.grm.epl.pl',PrepGrm) ; true),
          (parser(leftcorner_generalized)
          -> atom_concat(BaseG,'.grm.glc.pl',PrepGrm) ; true),
          expand_file_name(PrepGrm,[Grammar]), % expand the path to the home-directory
          nl,write('*****  Exporting the transformed grammar to file: '),
          write(Grammar),nl,
          (parser(earley_doerre)
          ->
              Kopfzeile = ' Precomputed for parser: Parser/Earley/earley.doerre.pl ',
              Regelzeile = '/***** Non-deleting rules  A --> B,... (transformed) ******/'
          ;
              parser(earley_doerre_dcg)
          ->
              Kopfzeile = ' Precomputed for parser: Parser/Earley/earley.doerre.dcg.pl ',
              Regelzeile = '/***** Non-deleting rules  A --> B,... (transformed) ******/'
          ;
              parser(earley_pratt_leiss)
          ->
              Kopfzeile = ' Precomputed for parser: Parser/Earley/earley.pratt.leiss.pl ',
              Regelzeile = '/***** Branching rules  A --> B,C,... (transformed) ******/'
          ;
              Kopfzeile = ' Precomputed for parser: Parser/Leftcorner/leftcorner.generalized.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) ; parser(leftcorner_generalized))
-> write(' Parameters: First-Relation: '),
    first_base(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(leftcorner_generalized)
-> portray_clause(parser_version((glc,first(BF,TC),predictor(PR)))),nl ; true),
(parser(earley_doerre)
-> portray_clause(parser_version(ed)), nl ; true),
(parser(earley_doerre_dcg)
-> portray_clause(parser_version(eddcg)), 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),
nl,write('/*****            Syntactical categories             ******/'),nl,
listing(synCat/1),
(parser(earley_pratt_leiss)
-> nl,write('/*****                  Restrictor                 ******/'),nl,
    listing(restrict/2),
    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/6)
; true),
(parser(leftcorner_generalized)
-> 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,D,LexC,A,Alpha,[B|Beta],Tree) ******/'),nl,
    listing(predicted/7)
; true),
nl,write('/*****                  End                          ******/'),
told,
% nl,write('*****  Precomputation of the grammar is finished.'),
nl,write('*****  Transformed grammar will now be loaded.'),
```

```
        abolish('-->'/2),
        abolish(restrict/2), % aber auch in .wrd-Dateien !?
        [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! Entsprechend werden Klauseln des Restriktors `restrict/2` beseitigt: damit die in der Grammatik fehlenden Klauseln ergänzt werden können, muß `restrict/2` dynamisch sein und zum Schluß wieder statisch gemacht werden.

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]).
```

Nur die Tupel auf oberster Ebene werden in eine Liste umgewandelt.

7 Tabellenparser für übersetzte Grammatiken

Die folgenden Parser-Varianten, die auf den vorberechneten Hilfsrelationen beruhen, bezeichnen wir als *verallgemeinerte* (Tabellen-) Verfahren. Sie terminieren auch für kontextfreie Grammatiken mit Lösch- und Kettenregeln und liefern alle kanonischen Analysen. Die Programme sind so geschrieben, daß sie auch für entsprechende DCG-Grammatiken und Grammatiken mit Metaregeln arbeiten (zur Zeit teilweise erst im Experimentierstadium).

7.1 Verallgemeinerte Earley-Analyse

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

7.1.1 Verallgemeinerter Earley-Parser

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

$$\mathtt{ptrees}(\mathtt{I}, \mathtt{J}, \mathtt{Kategorie}, \mathtt{Baumliste}, \mathtt{Beta}),$$

wobei `Baumliste` eine Folge $\vec{t}$ von Bäumen mit Wurzeln α ist.

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

$$\mathtt{ctrees}(\mathtt{I}, \mathtt{J}, \mathtt{Kategorie}, \mathtt{Baum}),$$

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:

$\langle$*Parser/earley.pratt.leiss.pl*$\rangle\equiv$
```
:- ensure_loaded('Parser/main.pl').
:- dynamic inputlaenge/1, expect/2, ctrees/4, ptrees/5.
```
$\langle$*Earley-Pratt-Leiß*$\rangle$
$\langle$*Hilfsfunktionen*$\rangle$

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

$\langle$*Earley-Pratt-Leiß*$\rangle\equiv$
```
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).
```

Durch `earley/2` wird die Tabelle aufgebaut; die abgeschlossenen Analysen sind darin leicht zu finden, da sie in `ctrees/4` abgetrennt wurden.

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

$\langle$*Earley-Pratt-Leiß*$\rangle+\equiv$
```
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.

Die Hauptschleife, `earley/2`, liest die Wortfolge der Eingabe von links nach rechts. Eine Verbesserung des Parsers (gegenüber der ursprünglichen Version von Earley) 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(_,_,_).

writeTriple(I,K,J) :-
        write('('),write(I),write(','),write(K),write(','),write(J),write(')').
```

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.[25]

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.

⟨*Earley-Pratt-Leiß*⟩+≡
```
predict(J) :-
        (ein(verbose) -> write('predict('),write(J),write(')'),nl ; true),
        (ein(omitRulesWithUselessLexCat)
        -> inputLexCats(J,InputLexCats) ; true),
        nextword_cat(J,LexCat),
        (J = 0 -> expect_initial(LexCat) ; true),
        expect(J,C),
        predicted(C,LexCat,A,_Alpha,[B|Beta],AlphaTrees),
        (ein(omitRulesWithUselessLexCat)
        -> memberLexCats([B|Beta],InputLexCats) ; true),
        (ein(omitEmptyTrees)
        -> add(ptrees(J,J,A,[],[B|Beta]))
```

[25] In `predicted/6` wird sowohl die Baumfolge `AlphaTrees` als auch deren Wurzelfolge `Alpha` vermerkt; für den Parser brauchen wir nur die Baumfolge, für den Erkenner nur die Wurzeln.

```
        ; add(ptrees(J,J,A,AlphaTrees,[B|Beta]))),
        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/5` 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:[26]

⟨*Earley-Pratt-Leiß*⟩+≡

```
memberLexCats([A|As],LexCats) :-
        var(A),!,memberLexCats(As,LexCats).
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 der durch `omitRulesWithUselessLexCat` eingeschaltete 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!

Bem.3: Man könnte bei `predict(J)` auch die Regeln *nicht* als nützlich voraussagen, deren rechte Seite mehr als $n-j$ Konstituenten verlangt. Das würde aber höchstens gegen Ende der Eingabe eine Beschleunigung bewirken und wird vorläufig nicht eingebaut.

Nachdem das Wort $w(j, j+1)$ gelesen wurde, geht man zurück zu 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 $k = j, j-1, \ldots, i$.

⟨*Earley-Pratt-Leiß*⟩+≡

```
complete(J) :-
        (0 < J -> I is J -1, complete(I,J) ; true).
complete(I,J) :-
        K is J - 1,
```

[26] Der erste Fall ist für Regeln mit variablen, erst zur Parsezeit durch Auswertungen von `{Tests}` instantiierten Konstituenten nötig.

```
        complete(I,K,J),
        (0 < I  -> M is I - 1, complete(M,J) ; true).
complete(I,K,J) :-  % 0 =< I und K < J garantiert
        (ein(verbose) -> write('complete'),writeTriple(I,K,J),nl ; true),
        (I < K
        -> complete1(I,K,J), complete2(I,K,J),
            M is K - 1,
            complete(I,M,J)
        ; true),
        (I = K -> complete2(I,K,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:[27]

(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 $\vec{t}s'\vec{e}$:

⟨*Earley-Pratt-Leiß*⟩+≡

```
complete1(I,K,J) :-
    ctrees(K,J,D,DTree),
    ptrees(I,K,A,AlphaTrees,[B|Beta]),
    % deletables(Beta), % zu frueh bei {Goal} = B = {definitheitswert(..,Def-)}
    connect(B,D,DTree,BTree),
    deletables(Beta),
    (ein(omitEmptyTrees)
    -> BetaTrees = []
    ; canonicalTrees(Beta,BetaTrees)),
    append(AlphaTrees,[BTree|BetaTrees],Subtrees),
    add(ctrees(I,J,A,[A|Subtrees])),
    fail.
complete1(_,_,_).
```

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

(Wir suchen zuerst nach den `ctrees(K,J,D,DTree)` und danach nach den `ptrees(I,K,...)`, da es normalerweise weniger vollständige als partielle Analysen eines Segments der Eingabe gibt. Dadurch sollte man sich mehr unnötige Arbeit sparen.)

(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 $\vec{e}$ der $\vec{B}$ und hat eine *partielle* Analyse $\vec{t}s'\vec{e} \in F(i,j,A \to \alpha B\vec{B} \cdot C\gamma)$ von $w(i,j)$:

⟨*Earley-Pratt-Leiß*⟩+≡

```
complete2(I,K,J) :-
    inputlaenge(N),          % C must derive at least one input word
    J < N,
    ctrees(K,J,D,DTree),
```

[27] Mit der Option `omitEmptyTrees` kann man einstellen, daß keine leeren Teilbäume in die Analyse eingebaut werden sollen. Damit kann man in der Grammatik optionale Konstituenten als löschbare Konstituenten, die in der Analyse ggf. gar nicht erscheinen, einführen.

```
        ptrees(I,K,A,AlphaTrees,[B|Beta]),
        connect(B,D,DTree,BTree),
        append(Bs,[C|Gamma],Beta),
        deletables(Bs),
        nextword_cat(J,LexCat),
        leftcorner_rtc(LexCat,C),
        (ein(omitEmptyTrees)
        -> BsTrees = []
        ; canonicalTrees(Bs,BsTrees)),
        append(AlphaTrees,[BTree|BsTrees],AlphaBBsTrees),
        add(ptrees(I,J,A,AlphaBBsTrees,[C|Gamma])),
        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.

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 welche Phrasenkategorien folgen können (in Ausdrücken der Erwartungskategorien).

Hilfsfunktionen zur Verwaltung der Tabellen

Wir sammeln ein paar Hilfsfunktionen, die sowohl für den Earley-Parser wie für den Leftcorner-Parser zum Aufbau der Tabellen gebraucht werden. Die Funktionen `connect/4` und `instance/2` sorgen zusammen mit `deletable/1` dafür, daß man z.T. reguläre Kombinationen einfacher Kategorien behandeln kann, also einen Teil von DCG-Grammatiken.[28]

⟨*Hilfsfunktionen*⟩≡
  ⟨*add*⟩
  ⟨*addExpect*⟩
  ⟨*connect*⟩
  ⟨*instance*⟩
  ⟨*leftcorner-rtc*⟩
  ⟨*nextword-cat*⟩
  ⟨*canonical-trees*⟩
  ⟨*extract*⟩

[28] Konjunktionen gehen wohl nur in der Form `(A,{B})` und `({A},B)` – zumindest ist der allgemeine Fall nicht durchdacht. In den Syntaxbäumen werden nur die den einfachen Kategorien entsprechenden Knoten gebildet.

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.)

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

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

⟨*addExpect*⟩≡
```
addExpect(J,({B},C)) :-
        !,call(B),addExpect(J,C).
addExpect(J,(B,{C})) :-
        !,call(C),addExpect(J,B).
addExpect(J,(B;C)) :-
        !,addExpect(J,B),addExpect(J,C).
addExpect(J,?(B)) :-
        !,addExpect(J,B).
addExpect(_J,{_Goal}) :-
        !.
addExpect(_J,[]) :-
        !.
addExpect(J,C) :-
        add(expect(J,C)).
```

Um an eine Erwartungskategorie `B` einen vollständigen Baum mit Wurzel `B` durch eine (kanonische) Kettenableitung anzuschließen, bzw. um diese Anschließbarkeit festzustellen, dienen folgende beiden Prädikate:

⟨*connect*⟩≡
```
connect(B,D,DTree,BTree) :-
        instance(BI,B),
        ( BI = D, BTree = DTree
        ; chain(BI,D), BTree = [BI,DTree]
        ).
connect(B,D) :-
        instance(BI,B),
        ( BI = D ; chain(BI,D) ).
```

Die einfachen Instanzkategorien `D` einer zusammengesetzten Kategorie `C` werden durch `instance(D,C)` bestimmt. Der Fall der Konjunktion ist nur für Konjunktionen `(A,{Cond})` und `({Cond},B)` gedacht[29]

⟨*instance*⟩≡
```
instance(D,X) :- var(X), !, fail.
instance(D,(A,B)) :-
```

[29] Für echte Konjunktionen von Konstituenten wäre noch zu überlegen, was man tun sollte. Die Parser werden also nicht für EBNF-Grammatikregeln mit expliziter Verwendung von Konjunktion wie in `A --> ((B,C),D)` funktionieren; Rechtsklammerung `(B,(C,D))` ist kein Problem.

```
        !,( deletable(A), instance(D,B)
          ; deletable(B), instance(D,A)
          ).
instance(D,(A;B)) :-
        !, (instance(D,A); instance(D,B)).
instance(D,?(A)) :-
        !, instance(D,A).
instance(D,{_Goal}) :-
        !, fail.
instance(D,D).
```

Durch `deletable/1` werden die Instruktionen ausgewertet und dadurch Spezialisierungen der Kategorien bestimmt.

Die Kategorien des bei Position `J` folgenden Eingabewortes wird durch `nextword_cat/2` ermittelt:

⟨*nextword-cat*⟩≡

```
nextword_cat(J,LexCat) :-
    J1 is J + 1,
    ctrees(J,J1,LexCat,_).   % LexCat = category of next word, already scanned!
```

Ob die Wortkategorie `LexCat` als linke Ecke eines Ausdrucks der zusammengesetzten Kategorie `C` auftreten kann, wird mit folgendem Prädikat getestet:

⟨*leftcorner-rtc*⟩≡

```
leftcorner_rtc(LexCat,C) :-
    instance(CI,C), (LexCat = CI ; first(LexCat,CI)).
```

Diese beiden Prädikate werden nicht zusammengefaßt, da man sie beim Leftcorner-Parser nicht direkt nacheinander ausführt.

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

⟨*canonical-trees*⟩≡

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

Da die Tabelle nur kanonische Bäume sammelt, muß zum Schluß mit `extract/3` eine Kette von einem Startsymbol zur Kategorie des gefundenen Baums ergänzt werden:

⟨*extract*⟩≡

```
extract(I,J,Baum) :-
        start_symbol(S),
        ctrees(I,J,Cat,Tree),
        (chain(S,Cat) -> Baum = [S,Tree] ; S = Cat, Baum = Tree).
```

7.1.2 Verallgemeinerter Earley-Erkenner

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 `ctrees` hat 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) :-
    ctrees(K,J,D),
    ptrees(I,K,A,_Alpha,[B|Beta]),
    connect_e(B,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),
    connect_e(B,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(_,_,_).

connect_e(B,D) :-
        instance(BI,B),
        (BI = D ; chain(BI,D)).
```

Beispiel 7.1 Für dieselbe Eingabe wie in Beispiel 5.13, 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.

7.2 Left-Corner-Analyse

Die Leftcorner-Analyse nutzt wesentlich die oben als `first/2`-Relation, aber sonst oft als *leftcorner*-Relation bezeichnete Beziehung zwischen Kategorien aus.

7.2.1 Einfache Leftcorner-Analyse

Die einfachste Form der Leftcorner-Analyse kann durch folgende Regeln zum Aufbau von Kanten beschrieben werden. Der Aufbau der Kanten zu $w = a_0 \cdots a_n$ erfolgt durch

(i) Lesen der Eingabewörter:

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

(ii) Voraussagen brauchbarer Regeln:

$$\frac{A \to B\gamma \in P \qquad j \overset{B \to \beta \cdot}{\longrightarrow} k}{j \overset{A \to B \cdot \gamma}{\longrightarrow} k} \quad \text{(Predict)}$$

(iii) Abschluß unter Verkettung von Kanten:

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

Beachte, daß die Voraussage –im Unterschied zu der des Earley-Parsers– vollständig durch erkannte Phrasen gesteuert wird und den Baum „nach oben“ erweitert. Es werden hier nur Regeln $A \to B\gamma$ benutzt, für deren *linke Ecke* B schon eine Analyse eines Teils der Eingabe gefunden wurde.

7.2.2 Verallgemeinerte Left-Corner-Analyse

Erwartet man, daß die Eingabe w von einer bestimmten Kategorie S ist, so kann man verhindern, daß bei der Voraussage auch Bäume aufgebaut werden, die nicht mit dieser „Erwartungskategorie“ verträglich sind. (Der Aufbau der Bäume *von unten* wird durch einen *Top-Down-Filter* beschränkt.) Dazu muß man allerdings –unabhängig von der Eingabe w– einmal die *Linke-Ecken-Relation*

$$<_{lc} \quad := \quad \{(B, A) \mid (A \to \alpha B\gamma) \in P, \ \alpha \Rightarrow^* \epsilon\}$$

und deren reflexive, transitive Hülle

$$<_{lc}^* \quad := \quad \{(B,A) \mid \exists\gamma \in Cat^*\ A \Rightarrow^* B\gamma\}$$

berechnen und als Tabelle speichern; während der Analyse einer Eingabe w wird diese Tabelle zur Beschleunigung der Suche verwendet.

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

(i) Festlegen von Erwartungskategorien für die Resteingabe:

$$\frac{start(S)}{0 \cdot S}\ \text{(Start)} \qquad \frac{i \overset{A \to \alpha \cdot B\beta}{\longrightarrow} j}{j \cdot B}\ \text{(Expect)}$$

(ii) Lesen der Eingabewörter:

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

(iii) Voraussagen brauchbarer Regeln:

$$\frac{i \cdot E \qquad A <_{lc}^* E \qquad A \to B\gamma \in P \qquad j \overset{B \to \beta \cdot}{\longrightarrow} k}{i \overset{A \to B \cdot \gamma}{\longrightarrow} k}\ \text{(Predict)}$$

(iv) Abschluß unter Verkettung von Kanten:

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

Wir implementieren eine Verallgemeinerung hiervon, bei der neben dem Top-Down-Filter durch die Erwartungskategorien auch die mögliche Verbindung zum nächsten Eingabewort als Filter benutzt wird; dazu wird in (Predict) und (Complete) geprüft, ob das nächste Eingabewort a_k mit dem ersten (nicht gelöschten) Kategorie C von γ in der $<_{lc}^*$-Relation steht.

Ein Nachteil des Earley-Parsers ist die Voraussage brauchbarer Regeln, die i.a. sehr viele Regeln vorschlägt. Der Left-Corner-Parser vermeidet diesen Schritt, indem er, ausgehend von einer Erwartungskategorie E für die Resteingabe ab Position j, zuerst das nächste Eingabewort a_{j+1} analysiert und dann zu jeder seiner Kategorien B die Regeln der Form $A \to BC\beta$ zum Aufbau partieller Bäume $t \in ptrees(j, j+1, A \to B \cdot C\beta)$ und der neuen Erwartung C für die Resteingabe ab Position $j+1$ benutzt, sofern A-Phrasen die linkeste Konstituente von E-Phrasen sein können. Entsprechend wird $A \to B$ zum Aufbau vollständiger Bäume $t \in ctrees(j, j+1, A)$ benutzt.

Zuerst werden die (noch leicht zu ändernden!) 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/leftcorner.generalized.pl*⟩≡
```
:- ensure_loaded('Parser/main.pl').
:- dynamic inputlaenge/1, expect/2, ctrees/4, ptrees/5.
```
⟨*Leftcorner*⟩
⟨*Hilfsfunktionen*⟩

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

⟨*Leftcorner*⟩≡

```
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
    leftcorner([W|Ws],0),
    !,extract(0,N,Baum).
```

Durch `leftcorner/2` wird die Tabelle aufgebaut; die abgeschlossenen Analysen sind darin leicht zu finden, da sie in `ctrees/4` abgetrennt wurden. Sie werden der Tabelle mit `extract/2` entnommen.

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

⟨*Leftcorner*⟩+≡

```
expect_initial :-
        ctrees(0,1,Cat,_),
        start_symbol(C),
        first(Cat,C),
        addExpect(0,C),
        fail.
expect_initial.
```

Das wird in `complete(0)` eingebaut (anders als beim Earley-Parser in `predict(0)`).

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` und `leftcorner/1` unten.

Die Hauptschleife, `leftcorner/2`, liest die Wortfolge der Eingabe von links nach rechts. Wir lesen *zuerst* das Eingabewort $w(j,j+1)$ und benutzen seine Kategorien, um bei der Vervollständigung der Analysen aller $w(i,j)$ und dem Aufbau der partiellen Analysen überflüssige Einträge zu vermeiden:

⟨*Leftcorner*⟩+≡

```
leftcorner([W|Ws],J) :-
        M is J+1,
        scan(J,M,W),
        complete(J),
        leftcorner(Ws,M).
leftcorner([],J) :-
        !, complete(J).
```

Die `scan`- und `complete`-Schleife ist fast dieselbe wie beim Earley-Parser, außer daß man `complete2(I,K,J)` für `I=K` nicht aufzurufen braucht (da man keine partiellen Bäume für `ptrees(I,I,A,Alpha,B,Beta)` voraussagt).[30]

⟨*Leftcorner*⟩+≡

[30] Und außer der Umformulierung mit `connect/4`. Teste: Wenn man in `complete1` und `complete2` die Aufrufe `ctrees(K,J,D,DTree)` *vor* den Aufrufen `ptrees(I,K,A,AlphaTrees,B,Beta)` macht, sollte die Analyse schneller werden, da es nur wenige vollständig analysierte Teilphrasen gibt, also die Suche in `deletable`, `chain` usw. oft unterbleibt.

```
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(_,_,_).

writeTriple(I,K,J) :-
        write('('),write(I),write(','),write(K),write(','),write(J),write(')').

complete(J) :-
        (0 < J -> I is J -1, complete(I,J) ; expect_initial).
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) :-  % 0 =< I und K < J garantiert
        (ein(verbose) -> write('complete'),writeTriple(I,K,J),nl ; true),
        (I < K
        -> complete1(I,K,J), complete2(I,K,J),
            M is K - 1,
            complete(I,M,J)
        ; true),
        (I = K -> predict(K,J) ; true).

complete1(I,K,J) :-
    ptrees(I,K,A,AlphaTrees,[B|Beta]),
    ctrees(K,J,D,DTree),
    % deletables(Beta), % zu früh bei {Goal} in Beta, wenn dadurch Vars(B) belegt wird
    connect(B,D,DTree,BTree),
    deletables(Beta),
    (ein(omitEmptyTrees)
    -> BetaTrees = []
    ; canonicalTrees(Beta,BetaTrees)),
    append(AlphaTrees,[BTree|BetaTrees],Subtrees),
    add(ctrees(I,J,A,[A|Subtrees])),
    fail.
complete1(_,_,_).

complete2(I,K,J) :-
    inputlaenge(N),          % C must derive at least one input word
    J < N,
    ptrees(I,K,A,AlphaTrees,[B|Beta]),
    append(Bs,[C|Gamma],Beta),
    deletables(Bs),
    ctrees(K,J,D,DTree),
    connect(B,D,DTree,BTree),
    nextword_cat(J,LexCat),
    leftcorner_rtc(LexCat,C),
    (ein(omitEmptyTrees)
    -> BsTrees = []
    ; canonicalTrees(Bs,BsTrees)),
    append(AlphaTrees,[BTree|BsTrees],AlphaBBsTrees),
```

```
    add(ptrees(I,J,A,AlphaBBsTrees,[C|Gamma])),
    addExpect(J,C),
    fail.
complete2(_,_,_).
```

Neu ist der Aufbau der Struktur mit `predict/2`. Ihm entspricht im Earley-Parser die Verbindung der durch `predict(K)` als brauchbar vorhergesagten Bäume aus `ptrees(I,I,A,Alpha,B,Beta)` mit den vollständigen Bäumen aus `ctrees(B,I,J)` in `complete2(I,I,J)`.

⟨*Leftcorner*⟩+≡

```
predict(K,J) :-
        (ein(omitRulesWithUselessLexCat)   % Schlecht: nur einmal pro J berechnen!
        -> inputLexCats(J,InputLexCats) ; true),
        expect(K,E),
        ctrees(K,J,D,DTree),
        nextword_cat(J,LexCat),
        predicted(E,D,LexCat,A,Alpha,[B|Beta],AlphaTrees),
        connect(B,D,DTree,BTree),
        append(Bs,[C|Gamma],Beta),
        deletables(Bs),
        leftcorner_rtc(LexCat,C),
        (ein(omitRulesWithUselessLexCat)
        -> memberLexCats([C|Gamma],InputLexCats) ; true),
        (ein(omitEmptyTrees)
        -> NewAlphaTrees = [BTree]
        ; canonicalTrees(Bs,BsTrees),
            append(AlphaTrees,[BTree|BsTrees],NewAlphaTrees)
        ),
        add(ptrees(K,J,A,NewAlphaTrees,[C|Gamma])),
        addExpect(J,C),
        fail.
predict(K,J).

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).

memberLexCats([A|As],LexCats) :-
        (lexCat(A) -> \+ \+ memberchk(A,LexCats) ; true),
        memberLexCats(As,LexCats).
memberLexCats([],_).
```

Bem. Die Reihenfolge der Suche nach `ctrees(K,J,D,DTrees)` und `ptrees(I,K,...)` ist hier noch anders als im Earley-Parser, nur für Effizienzvergleiche. Das hat

Durch `predict/2` werden die als brauchbar vorhergesagten Regeln zur Erweiterung einer vollständigen Konstituente „nach oben" verwendet. (Hier scheint die Option `omitRulesWithUselessLexCat` weniger nützlich zu sein als beim Earley-Parser.)

Der Vorteil von `predict/2` des Leftcorner-Parsers gegenüber `predict/1` des Earley-Parsers besteht darin, daß man in `predict(K,J)` die brauchbaren Regeln `A -> Alpha B Beta` genauer herausfiltern kann, indem man (a) `B` durch eine Kettenableitung mit einer schon erkannten Phrase `ctrees(K,J,D,DTree)` verbindet und (b) mit der Kategorie `LexCat` des Worts bei `J` prüft, ob dies eine Anfangskonstituente von `Beta`-Phrasen sein kann. Im Earley-Parser benutzt man nur `LexCat` zur Bestimmung der brauchbaren Regeln. Wegen (a) enthält die im Prediktor des Leftcorner-Parsers aufgebaute Struktur mindestens ein Eingabewort, während die des Earley-Parsers noch gar nichts von der Eingabe abdeckt.

8 Anhang 1: Hilfsprogramme

8.1 Transitive Hülle einer zweistelligen Relation

Wir brauchen die transitive Hülle gewisser Relationen zwischen grammatischen Kategorien, insbesondere die der Relationen

$$\{(A,B) \mid (A \to B) \in P\} \quad \text{und} \quad \{(A,B) \mid \exists\alpha,\beta\,((A \to \alpha B\beta) \in P \land \mathit{deletable}(\alpha))\}.$$

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).
```

8.1.1 Der Algorithmus von C.Warshall für Relationen zwischen Grundtermen

Wir orientieren 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 8.1 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.

8.1.2 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 8.1.4 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 8.2 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 8.3 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.[31]

⟨*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)).
```

[31] 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[32]

⟨*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).
```

[32] 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 8.4 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 8.5 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 8.6 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.

8.1.3 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.[33]

⟨*link*⟩+≡
```
circularity_test(C,A) :-
    (circle(C,A) -> true ; circle(A,C)).

circle(C,A) :-
```

[33] 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.

8.1.4 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 $\{(X, f^n(X)) \mid n \in \mathbb{N}\}$ 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),nl)
    ; 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:[34]

⟨*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))),
```

[34]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).
```