A Discourse System for English[1]

Stefan Müller

March, 1993

[1]This work was done as AI3–Large Practical at Edinburgh University.

Contents

1 How to start the program

Load the file *start* or *startchart* into your Prolog system. Then there are several ways to use the program. *t(X)* tests whether the example X[1] has a parse tree and *p(X)* prints all parse trees the parser can find. *test* tries to recognise examples given on the task sheet. *test2* recognises further more complicate test examples I gave. *prall* prints all parse trees for all examples and stops after each example waiting for pressing the return key. Type *tf(X)* to see the ill formed sentences which the grammar rejects. *problems(X)* shows the ill formed sentences the grammar doesn't reject[2]. *testtime* and *testtime2* force Prolog to backtrack and to find all parse trees of the appropriate sets of examples. They print out the time they needed. *testtime* and *testtime2* do not print out parse trees. The only use these predicates have is to measure the running time.

go is a top level predicate which asks for input and tries to analyse it. The program doesn't make a difference between upper and lower cases. You can end the input by typing 'end.'. *go* does only analyse the syntax. It prints out all parse trees it can find.

disc is a top level predicate similar to *go*. But it computes the semantic of the input.

There are several trace modi. If you type in *trees.* during a discourse the program will print out parse trees. *notrees* switches this off again. You will get the DRSs printed out by typing *drs. nodrs.* switches this off. The evaluation of restrictions can be made visible by the *restr.* *norestr.* switch.

Part I
Syntax for a fragment of English

2 The Formalism

The grammar is a DCG-grammar using the build in DCG-formalism of SICSTUS Prolog. I use an extension of this formalism which allows to pass arguments in a DAG rather then in single arguments. Therefore the notation is clearer and the lexicon is more handy.

3 Parse Trees

Parse trees are build up in a usual way:

[1]see appendix A.4

[2]see problems 6

```
s(s(NP,VP)) --> np(NP),vp(VP).
```

NP and VP are the parse trees of the appropriate structures. One difficulty which arises if one parses conjunctions I solved in the following way:

```
s_fin(S) --> s(A,nogap,nogap,S1),more_s_fin(S1,S),
  { A:vform <=> fin}.

more_s_fin(S,S) --> [].
more_s_fin(S1,S3) --> coord(_,C), s_fin(S2),
                      more_s_fin(coordination(S1,C,S2),S3).
```

In the latter rule S1 could be a complex term which is built up recursively.

I use a pretty printer to put them out which is written by Christoph Lehner [6] (see A.5).

4 The Grammar

4.1 Introduction

In English there exists basically four clause types:

- Declarative
- Imperative
- Interrogative
- Exclamative

My grammar handles only declarative and interrogative. Its top level symbols are s_eng and question. s_eng can be built up out of several finite sentences which have to be connected by a coordinator like 'and', 'or' ,or 'but' and the like. 'but' and other coordinators are not implemented.

4.2 Declaratives

An ordinary sentence contains a noun phrase and a verb phrase, which have to agree in number and person. My grammar handles topicalised sentences as well. Therfore I wrote a rule which demands an object (an accusative noun phrase) followed by a sentences with a gap just for this object.

4.3 Interrogatives

There are some rules for questions which are using gap-threading to allow to handle questions that ask for the subject of a sentence. For instance: "Who sees him?". The grammar recognizes these questions by using the rule:

```
que(A0,que(interr(who),Q)) --> who(_),  s(A1,gap(np(NPGap)),nogap,Q),
  {
  NPGap:cas  <=> nom,
  NPGap <=> sing3,
  A1:vform <=> fin,

  ... }.
```

To handle questions like "Does he see him?" and "Has John said that Mary likes him?" I wrote two extra question rules:

```
que(A0,G0,G,que(Do,NP,VP)) --> vDo(A1,Do),
                              np(A2,G0,G1,NP),
                               vp(A3,G1,G,VP),
  {
  A1:vform <=> fin,
  A1:agr   <=> A2:agr,
  A3:vform <=> bse,

  ... }.

que(A0,G0,G,que(HAVE,NP,VP)) --> vhave(A1,HAVE),
                                 np(A2,G0,G1,NP),
                                  vp(A3,G1,G,VP),
  {
  A1:vform <=> fin,
  A1:agr   <=> A2:agr,
  A3:vform <=> pas,

  ... }
```

If one combines these rules again with gap-threading it is possible to parse questions like: "Who does see him?" and "What did he say?". In the latter question the gap is a relative sentence gap because we are asking for a relative sentence.

To handle 'which' questions I introduced a special noun phrase construction in which a noun is preceded of a 'which' instead of a determiner and perhaps an adjective. "John wrote the book?" is accepted as a yes/no question.

4.4 Adjectives

Adjectives of colour normally follow adjectives of size in English. But one can hardly say that 'a black big dog' is agrammatical. Therefore I decided to write the rules for adjectives in the following way:

```
adjs(A0,adj(DEG,ADJ)) --> deg(A1,DEG), adjs1(A0,ADJ),
  {
  A0:det <=> A1:det}.

adjs(A,ADJ)   --> adjs1(A,ADJ).
adjs1(A0,ADJ) --> adj(A1,ADJ1),moreadj(A2,ADJ1,ADJ),
  {
  A0:det <=> A1:det,

  ... }.

moreadj(A,ADJ,ADJ) --> [],
  {
  ... }.

moreadj(A0,ADJ,adjs(ADJ,ADJ2)) --> adj(A1,ADJ1),
                                   moreadj(A2,ADJ1,ADJ2),
  {
  ... }.
```

4.5 Coordinators

I have implemented 'and' and 'or'. The way I have done this for sentences and verb phrases is a straightforward one. In positions in which verbs of a particular category are allowed a conjunction or disjunction is allowed too:

```
verb(A,Cat,VP) --> v(A,Cat,VP).

verb(A0,Cat,vp_conj(V1,C,V2)) -->
            v(A1,Cat,V1),coord(and,C),v(A2,Cat,V2),
  {
  A0:vform <=> A1:vform,
  A0:vform <=> A2:vform,
  A0:agr   <=> A1:agr,
  A0:agr   <=> A2:agr,

  ... }.
```

The latter rule is needed to make sentences like: "I read and lost the book." possible.

It is allowed to combine topicalised sentences. Some problems occurred when I wrote the rules for noun phrases. They look as follows.

```
np3(A1,NP) --> np2(A2,NP1),
               more_np(A3,A2,Amo,NP1,NP),
  {
  A1:cas <=> A2:cas,
  A1:cas <=> A3:cas,
  A1:agr <=> Amo:agr}.

more_np(_,Ami,Amo,NP,NP) --> [],
  { Ami:agr <=> Amo:agr}.

more_np(A1,_,Amo,NP,NP2) --> coord(and,C),
          np3(A2,NP1),
          more_np(A3,Ami,Amo,coordination(NP,C,NP1),NP2),
  {
  A1:cas <=> A2:cas,
  A1:cas <=> A3:cas,
  Ami:agr:num <=> plur}.

more_np(A1,_,Amo,NP,coordination(NP,C,NP2)) --> coord(or,C),
                                 np3(A2,NP1),
                                 more_np(A3,A2,Amo,NP1,NP2),
  {
  A1:cas <=> A2:cas,
  A1:cas <=> A3:cas}.
```

In both types of coordinatives the case must be in all parts equal.

The number of conjunctions is always plural. The agreement of a noun phrase which is a disjunction must be the same as the last element of the disjunction.

Because of two possible ways to build up structures out of three parts there are parsings for both of the following sentences:
Mary and (the students or John) see him.
(Mary and the students) or John sees him.

4.6 Determiners

According to Huddleston [9] a NP may contain up to three determiners, as in "all her many virtues". The relative order is fixed and one distinguishes therefore three determiner positions, 1,2,3. One takes 2 as basic, and defines 1 and 3 in reference to 2.

1. all, both; such, what; half, one-third, ...; double, twice ...
2. the; a/an; this, these, that, those; my, your, ...; we, us, you; some, any, no, either, neither, another; each, enough, much, more, most, less; a few, a little; which, what
3. one, two, three, ...; dozen; many, several, few, little, every

There are, however, very severe limitations on how forms from the three positions may combine: a grammar will need to give detailed rules that will allow, say, 'both these books' but not '*both some books', 'her every gesture' but not '*this every gesture', and so on.

'Few' and 'little' belong under 3 because because they can follow 'the', demonstratives, possesives, relative 'what', etc., as in 'these few corrections', 'what little money I have'. 'A few' and 'a little' by contrast are each single determiners, not a sequence of 2 (a) + 3 (few/little): in 'a few corrections', for example, 'a' does not enter into construction with 'corrections', for it requires a singular ('a correction', not *'a corrections').

This is the reason why I have dispensed with rules for a determiner phrase and just have written the determiners into the lexicon.

np1 which is either a noun or a noun preceded by some adjectives has a 'det' feature which determines whether the noun phrase combines with 'a' or with 'an'. This feature can be influenced by the first adjective of *np1*.

Therefore two noun phrase rules look like that:

```
np2(A0,PM) --> det(A1,DETP), np1(A2,NP),
              post_modifier_np(A3,np(DETP,NP),PM),
  {
   A1:det   <=> A2:det,
   A1:agr   <=> A2:agr,
   A0:agr   <=> A2:agr,
   A0:cas   <=> A2:cas,
   A0:human <=> A2:human,
   A0:human <=> A3:human,

   ... }.

np1(A,N) --> n(A,N).
np1(A0,np(ADJ,N)) --> adjs(A1,ADJ), n(A2,N),
  {
  A0:det <=> A1:det,
  A0:agr <=> A2:agr,
  A0:cas <=> A2:cas,

  ... }.
```

4.7 Negation

The negation is a rather difficult problem. The grammar handles only verb negation.

A 'not' could not simply act as a pre_modifier of a verb phrase. A negated verb phrases must contain an operator. Operators are: be, have, do, can, may, must, will, shall, should, ought, need and dare. So I decided to have extra rules for every occurrence of 'not'. These rules can handle phenomena like aux + not, have + not and be + not.

4.8 Noun Phrases

I handle noun phrases in an usual way. See 4.6, 4.5 and 4.9.

4.9 Relative Clauses

A relative clause can be seen as a sentence with a gap. The gap has to have the same number and person as the noun phrase the relative clause refers to. The case might be different:

```
He likes the man I gave the book to.
The man I gave the book to likes him.
```

In the first sentence 'man' and the missing object do have the same case but in the second they don't because man is the subject of the sentence.

I distinguish between restrictive and non restrictive relative clauses. Non restrictive relative clauses have to be marked off by commas. Proper nouns can only be followed by non restrictive relative clauses. Normal noun phrases can have both kinds of relative clauses and pronouns neither restrictive nor non restrictive.

4.10 Topicalisation

The rule for topicalisation is the following:

```
s(A0,nogap,nogap,s(NP,comma(',')),S)) --> np(A1,nogap,nogap,NP),
                                                          comma(_T),
                                    s(A0,gap(np(A1)),nogap,S),
  {
  A1:cas   <=> acc,
  A0:vform <=> fin}.
```

The restriction for the topicalised noun phrase is made to exclude sentences like **Mary, I believe that told Bill that she likes him.* Sentences like *John, Mary told Bill that she likes.* will be excepted.

4.11 Verb Phrases

According to Huddleston [9] there are five major patterns of complementation for English:

S P	Intransitive	He won.
S P PC^s	Copulative	He became ill.
S P O^d	Monotransitive	He saw her.
S P O^d PC^O	Complex-transitive	She considered him foolish.
S P O^i O^d	Ditransitive	He gave her the book.

S means subject, P predicator, PC^s subjective predicative, PC^O objective predicative, O^d is the direct and O^i the indirect object.

I have rules for intransitive, copulative, monotransitive, and ditransitive verbs.

The indirect object (O^i) is in general an optional element though there is a rule:

```
vp1(A0,G0,G,vp(V,NP)) --> verb(A1,ditrans,V), np(A2,G0,G,NP),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A2:cas   <=> acc,

  ... }.
```

The sentence "John gave the book to Mary." will be recognized by this rule applying a prepositional phrase as post_modifier for the verb phrase.

In order to handle special constructions like verbs which need to be followed by a infinitive verb phrase I introduced particular verb classes like 'inf' and 'inf2'. The latter needs another noun phrase like in "He wanted *her* to go." in contrast to "He wanted to go."

Verbs which need a sentence as complement I put in a category named 'that'.

There are two rule which allow a form of 'be' in combination with any transitive or ditransitive verb to be a complete passive verb phrase.

```
verbtrans(A,VP) --> verb(A,trans,VP),
   {
   A:vform <=> pas}.

verbtrans(A,VP) --> verb(A,ditrans,VP),
   {
   A:vform <=> pas}.

vp1(A0,G,G,vp(BE,VP)) --> vbe(A1,BE),  verbtrans(A2,VP),
   {
```

```
   A0:vform <=> A1:vform,
   A0:agr   <=> A1:agr,

   ... }.
```

Again sentences like "The books were given to the students." and "The book was lost by a student." are handled by a vp rule on a higher level which adds a prepositional phrase.

Copula constructions do have the following form:

```
vp --> verb_copulative, direct_object.
vp --> verb_copulative, subjective_predicative.
```

Because there are some verbs which need a adjectivel PC^S and don't allow other complements, for instance He went mad/*a lunatic., I decided only to allow 'be' as a copulative verb.

In general there has to be number agreement between the subject of the sentence and the NP predicative. Therefore the rules are:

```
vp1(A0,G0,G,vp(VBE,PCs)) --> vbe(A1,VBE), pcs(A2,G0,G,PCs),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A0:agr   <=> A2:agr,

  ... }.

pcs(A,G0,G,ADJP) --> adjp(A,G0,G,ADJP).

pcs(A0,G0,G,NP) --> np(A1,G0,G,NP),
  {
  A0:agr:num <=> A1:agr:num,  % in general

  ... }.
```

Because the possibility to say "That is I." exists as well as "That is me." although the first one is rather uncommon I do not make a case restriction for noun phrases. Therefore the grammar excepts "Who am I?" and *"Who am me?". With a case restriction would be either "That is me." or "Who am I?" impossible.

5 The Lexicon

I extended the DAG notation given by C Mellish [1]. I implemented a template algorithm. It is possible to give one template multiple meanings. The *denotes* predicate looks now that way:

```
denotes(X,X) :- var(X), !.
denotes(X,Y) :- template(X,Y). % makes templates backtrackable
denotes(X:F,V) :- !, denotes(X,V1), member(F:V2,V1), !, V = V2.
denotes(X,X) :- not template(X,_).
```

A template can be a complex entry containing disjunctions.

```
template(general_tense,A) :-
  A:vform <=> fin,
  (
          A:agr:num <=> plur;
          A:agr:num <=> sing,
          (
                  A:agr:per <=> 1;
                  A:agr:per <=> 2)
  );
  A:vform <=> bse.
```

This denotes either a finite verb form which is plural or a singular form with person 1 or 2 or a base verb form.

6 Problems

6.1 Gaps within Conjunctions and Disjunctions

With the gap mechanism suggested by C Mellish & P Whitelock [1] which I am using it is not possible to handle sentences like "Roo, Tigger bounced and frightened." easily.

In a conjunction of verb phrases the gap would disappear after the first one and therefore only sentences like "Roo, Tigger bounced and frightened Roo." would be possible. I decided not to allow gaps in conjunctions or disjunctions at all.

6.2 Gaps in Subordinatives

Because I allow gap-threading in subordinatives to handle sentences like
What do you you believe that Mary told Bill that John had said?
the grammar accepts sentences like
Who do you believe that told Bill she likes him?
One could solve this problem with extending the restriction of subordinative rules to test the categories of gaps.

6.3 Noun Phrase Modifiers

The grammar accepts sentences like
"The man in the garden who had read a book is afraid of the dog."
as well as
"The man who had read a book in the garden is afraid of the dog."
In the last sentence 'in the garden' is parsed as a NP modifier too but is only understood as a VP modifier. So there might be the need to have modifiers in a particular order.

6.4 Parse Trees

There are two parse trees for "He won and he won and he won". That is because of two possibilities of parsing that sentence. Firstly you could parse it as the conjunction of two sentences of which the last is a conjunction itself and secondly you could parse it as a conjunction of two sentences of which the first is a conjunction of two sentences. The same would be the case for conjunctions with more then two NPs or VPs.

Part II
Parse Strategies

7 Introduction

The general problem with all parse strategies is the restriction of the rules, the DAG-unification, which happens at the very end of some rules. There are rules without restrictions too. You cannot apply the normal parse strategies because you have to handle complex categories like

```
vp(2,[vform:fin,agr:[num:sing,per:1|_]|_],nogap,nogap).
```

That means that the normal Prolog unification doesn't work any longer.

```
vp(2,[vform:fin|_],nogap,nogap)
```

for instance doesn't match.

Another problem are ϵ-productions. Top down strategies can cope with ϵ-productions. It is difficult to handle them in bottom up strategies. For bottom up strategies the gap information has to be explicit. One way of making this information explicit is shown in Pereira, Shieber[2]. I have written a top down parser using this technique of gap-handling. For the bottom up parsers I wrote a compiler which transforms grammars in ϵ-free grammars.

8 A Compiler that produces ϵ-free Grammars

The compiler transforms every rule of the form

```
lhs ---> cat1,cat2,cat3,...,{restriction}.
```

into a rule

```
rule(lhs,[cat1,cat2,cat3,...,{restriction}]).
```

The restriction is optional.

If any number of the categories on the RHS can be empty because there exist appropriate ϵ-productions then the compiler finds the rules with empty categories on the RHS as well.

If there is a rule

```
cat1 ---> [],{restriction1}.
```

the solution

```
rule(lhs,[cat2,cat3,...,{restriction1},{restriction}]).
```

will be found. The same applies if cat1 is a LHS of a rule which has a RHS in which all categories can be reduced to ϵ. An example of my grammar is:

```
pp(gap(pp),nogap) ---> [].

post_modifier_vp(G ,G) ---> [].
post_modifier_vp(G ,G) ---> adv.
post_modifier_vp(G0,G) ---> pp(G0,G1), post_modifier_vp(G1,G).

vp(G0,G) ---> v, post_modifier_vp(G0,G).
```

The rules the the compiler builds for vp are:

rule(vp(G,G),[v]).
rule(vp(gap(pp),nogap),[v]).
rule(vp(G0,G),[v, post_modifier_vp(G0,G)]).

The program works as follows:

```
rule(LHS,List) :-
 (LHS ---> RHS),
    RHS \= [],
    RHS \= ([],_Restr),
  conj_to_list(RHS,L),
  process(L,List).
```

One rule which is not an ϵ-production is selected, transformed into a list, and the list is processed.

To process the RHS of a rule means that every selection of elements of the RHS-list which can be reduced to ϵ has to be reduced to ϵ and the conditions which are necessary to do so (the restrictions) are given back.

```
process(Lin,Lout) :-
 select(Part,Lin,List),
 List \= [],
 List \= [{_Restr}],
 gaps(Part,Restr),
 xappend(List,Restr,Lout).
```

The second predicate *process* is written under the assumption that for each element in the RHS-list exists at least one non empty production. This assumption makes sense because a category which only can be ϵ should not occur in a grammar.

```
process(List,List).
```

select gives you the following results:

```
select(A,[a,b,c],R).

A = [a], R = [b,c] ? ;

A = [b], R = [a,c] ? ;

A = [c], R = [a,b] ? ;

A = [a,b], R = [c] ? ;

A = [a,c], R = [b] ? ;

A = [a,b,c], R = [] ? ;
```

It selects out of a list every subset and gives the rest back in its third argument.

```
select([X],L,R) :- rub(X,L,R).
select([X|Other],[X|L],R) :- select(Other,L,R).

rub(F,[F|X],X).
rub(F,[E|X],[E|Y]) :- rub(F,X,Y).
```

process tests each selection. It calls *gaps* which succeeds and returns the restrictions if the selection is reducible to ϵ. Otherwise it fails.

gaps tests for each element of the list whether it is possible to reduce it to ϵ by using *gap_test* or if there is a rule which has a RHS which is reducible to ϵ.

```
gaps([],[]).

gaps([L|R],Restrictions) :-
 gap_test(L,Restr),
 gaps(R,Rest),
 melt(Restr,Rest,Restrictions).

gaps([H|T],Restrictions) :-
 (H ---> RHS),
 conj_to_list_r(RHS,List,Restr),
 gaps(List,RestrList1),
 melt(Restr,RestrList1,RestrList2),
 gaps(T,RestrList3),
 xappend(RestrList2,RestrList3,Restrictions).

gap_test(Gap,[]) :-
 (Gap ---> []).

gap_test(Gap,Restr) :-
 (Gap ---> [],Restr).

conj_to_list_r((A,B),[A|List],Restr) :- !,
      conj_to_list_r(B,List,Restr).
conj_to_list_r({R},[],{R}) :- !.
conj_to_list_r(A,[A],[]).
```

melt is a simple predicate which includes one element into a list.

```
melt([],X,X) :- !.
melt(X,Y,[X|Y]).
```

9 Top-Down Parser

I have implemented a top-down parser for Filler-Gap DCGs in almost the same way Pereira and Shieber [2] did. The only thing I have changed is one operator definition to allow the rules like the one below. The definitions were:

```
:- op(300, xfx, fills).
:- op(300, xf, island).
```

and are replaced by:

```
:- op(300, xfx, fills).
:- op(299, xf, island).
```

The gap-threading does not happen in the grammar rules anymore. Therefore the places at which gaps can be introduced were made explicit in the rules. A rule can look like follows:

```
% topicalisation, objects in first position
s(A,s(NP,comma(','),S)) --->
        np(2,A1,NP) island fills np(2,A1,np('_')), [','],
        s(A,S),
  {
  A1:cas <=> acc}.

restr_rel_cl(A,restr(S)) ---> [] fills np(2,A2,np('_')), s(As,S),
  {
  As:vform <=> fin,
  A:agr    <=> A2:agr,
  A2:cas   <=> acc}.
```

The operator *island* means that during parsing the np to which it applies no gap rule is allowed to be used. This is equivalent to np(2,A1,nogap,nogap,NP).

The fills operator shifts the following np on a stack. It has to be consumed during the parsing of the following constituents.

To see how the whole grammar looks like see appendix B.1.2

10 Shift-Reduce Parser

The shift-reduce parser is quit simple. The only extension I made is the *append_bit* predicate which gives reversed parts of the stack back. It gives the largest parts first because the parser shall try to reduce as much as possible.

The parser doesn't handle ϵ-productions. Without an explicit notation of the gaps it would be hardly possible to write a shift-reduce parser which handles ϵ-productions because the parser can shift in every state a infinite number of these so that at least the backtracking would not terminate.

As I mentioned before (See 8.) I have written a compiler which transforms grammars with ϵ-productions in grammars without. The result of the compilation is a set of rules which have the form:

```
rule(cat,[cat1,cat2,cat3,...,{Restriction1},{Restriction2},...]).
```

The restrictions are optional.

```
recognise(Stack,Tree) -->              % reduce
    [],
    {
    append_bit(Bit,Rest,Stack),

    rule(LHS,RHS2),
    match(Bit,RHS2)},
    recognise([LHS|Rest],Tree).
```

The interesting thing about this predicate is the way *match* works. The Prolog database might give back a RHS2 which contains a restriction. Therefore the RHS would not match against any part of the stack because there are no restrictions on the stack. That means *match* has to evaluate the restrictions.

```
match([],[{Restr}|Restrictions]) :- !,
Restr,
        match([],Restrictions).
```

Because there can be complex categories on the stack which are already initialised *match* cannot simply use the Prolog unification. It has to use graph-unification as well.

```
match([X|R1],[X|R2]) :- !,match(R1,R2).

match([X|R1],[Y|R2]) :-
        X =.. [Fun,DAG1|Rest],
        Y =.. [Fun,DAG2|Rest],
        !,
        graph_unify(DAG1,DAG2), match(R1,R2).
```

```
recognise(Stack,Tree) -->                  % shift
    [Word],
    {
    word(Cat,[Word|Restr]),
    eval(Restr)},
    recognise([Cat|Stack],Tree).
```

The *eval* predicate is the only thing that needs explanation in the shift rule. The difference list `[Word|Restr]` fills Restr either with {Restriction} or with [ ]. In the latter case eval([ ]) applies and succeeds. In the first case actual evaluation will take place.

```
eval([]) :- !. % no restrictions made
eval([{Restr}]) :- Restr.
```

The predicate

```
recognise([s_eng(Tree)],Tree,[],[]).
```

terminates the parsing and returns the parse tree.

Because the uninitialised call of *rule* in the *recognise* predicate which is responsible for the reduction the parser is very inefficient. One can improve this a little bit by using a different rule notation. The predicate *transform* transforms a rule

```
rule(s(G0,G),[np(G0,G1),vp(G1,G),{restriction}]).
```

into

```
rule([s,G0,G],[[np,G0,G1],[vp,G1,G],{restriction}]).
```

It looks that way:

```
transform(rule(RHS,LHS),rule([Fun|Args],T)) :-
  RHS =.. [Fun|Args],
  transform(LHS,T).

transform([],[]) :- !.
transform([{X}|R],[{X}|T]) :- !,transform(R,T).
transform([X|Rest],[[Fun|Args]|T]) :-
  X =.. [Fun|Args],
  transform(Rest,T).
```

I rewrote the *recognise* predicate so that it works with these rules.

```
recognise(Stack,Tree) -->            % reduce
    {
    append_bit([[Cat|ArgsBit]|Rest_Bit],Rest,Stack),

    rule(LHS,[[Cat|ArgsRHS]|Rest_RHS]),
    match([ArgsBit|Rest_Bit],[ArgsRHS|Rest_RHS])},
    recognise([LHS|Rest],Tree).
```

match was changed too:

```
match([[Fun,DAG1|Rest]|R1],[[Fun,DAG2|Rest]|R2]) :- !,
        graph_unify(DAG1,DAG2), match(R1,R2).
```

The terminating predicate is modified into

```
recognise([[s_eng,Tree]],Tree,[],[]).
```

11 Left-Corner Parser

11.1 Compute Links for Left-Corner Parsers

I think for bigger grammars then the sample grammars in the lecture notes one needs a generator which computes the links for the left-corner parser. I have written such a generator. See appendix B.2.1.

It asserts a link for each rule, i.e. `s ---> np,vp.` causes the link *link(np,s)*. Because there is an ϵ-production for np the link *link(vp,s)* has to be asserted too. I use the predicate *links* to determine the transitive closure.

```
links(A,B) :- link(A,B).
links(A,C) :- link(A,B),links(B,C).
```

To avoid multiple entries of the same link which Prolog produces if there are different ways to find one link I use *assert_one* which asserts a link only if it isn't already asserted.

Because of the ϵ-productions their is the possibility to find a link like *link(s,s)*. This has to be avoided by testing, because otherwise this would allow more then one solution for *link(cat,cat)* in the case of backtracking.

The link relation the left-corner parser needs is reflexive and transitive. The transitive closure is computed by the *links* predicate described above. To make the relation reflexive I have added the general link *link(A,A)*.

11.2 The Parser

To see how left-corner parsers work in general see Mellish[1] and Pereira, Shieber [2].

Because DAG-unification is used in my grammar I have to have one predicate which tests this unification:

```
lc(P1,P2) --> [],
   { P1 =.. [Fun,DAG1|Rest],
     P2 =.. [Fun,DAG2|Rest],
     !,
     graph_unify(DAG1,DAG2)}.
```

The DAG has to occur at the first argument position of the term (np(DAG,G0,G)).

12 Chart Parser

I modified the chart parser given by Gerald Gazdar and Christopher Mellish [7].

I added one *foreach* call to write all ϵ-productions and the hypotheses they cause into the chart. For the position of edges of ϵ-productions I used variables. That means for the ϵ-production `more_s_fin(S,S) ---> [].` the edge

edge(V,V,more_s_fin(S,S),[ ],['_'],no_restr) will be added. Because of the use of variables (V,V) each gap combines at every position of the sentence with other appropriate categories.

Because of the necessity of graph-unification I did not evaluate the restrictions during building the chart. If one would do this one would have to define predicates for subsumption and graph-unification for lists. However, this is highly inefficient in Prolog. I guess for this one would have to write a new language with a different unification concept.

I built up the chart bottom up and left to right. The construction of the actual trees happens top down and left to right in the predicate *get_tree* and therefore there are no problems with graph-unification.

The left to right computation of the trees is necessary because of the semantic checks which have to be evaluated from left to right.

I have also modified the rule which enters new hypotheses:

```
foreach(rule(Category2,[Category1|Categories]),
    add_edge(V1,V1,Category2,[Category1|Categories],[Category2])),
```

has been replaced by:

```
foreach(rule(Category2,[Category1|Categories],Restr1),
    add_edge(V1,V2,Category2,Categories,[V2,Category1,V1],Restr1)),
```

because Category1 has been recognised already and the same edge would be added in a combination step. The edges from V1 to V1 aren't of any use.

I keep record about the position in the string at which each category is found so that I am able to build parse trees using the chart rather then the grammar. To avoid unnecessary work I removed the parse tree argument from all grammar rules.

13 Efficiency and Statistics

The pre-processing of the rules to transform them from the `--->` notation into *rule* predicates brought an enormous improvement of the efficiency of parsers which need such an interface to the grammar.

The grammars themselves and the transformations made are written in a way that the shortest parsing possibility is asserted first to the Prolog database.

That means that `s_fin ---> s.` precedes `s_fin ---> s,s_fin.` Because the latter rule is less likely used this speeds up the searching if only one solution is needed.

I have written the *tt(X)* predicate which compute all parse trees to make the different algorithms comparable. *testtime* tests the examples given on the task sheet (Examples 1 to 44 see appendix A.4). *testtime2* measures the time for the examples 70 to 129.

To compare the parsers one could count the access to grammar rules but the work the different parsers have to do apart from that differs so that this would not be a satisfying comparison.

The time Prolog needed for one call of *testtime* and *testtime2* respectively is:

	testtime	*testtime2*	sum
Prolog DCG parser	292.590 sec	885.831 sec	1178.421 sec
chart parser	66.973 sec	111.728 sec	178.701 sec
left-corner parser	158.704 sec	573.552 sec	732.256 sec
left-corner parser for lists	186.591 sec	682.921 sec	869.512 sec
top-down parser	1378.872 sec	3286.591 sec	4665.463 sec
shift reduce parser	many hours		
shift reduce parser for lists	some hours		

The time has been measured on a SUN machine (HAWK).

These examples are a colourful mixture of different syntactic constructions. Some parsers do have advantages for some particular constructions.

The chart parser for instance finds very fast all possible solutions if it finished building the chart. It avoids unnecessary work which would be done by a top down parser for instance if the top down parser tries to recognise vp which is a v,np and pp and applies first a rule `vp --> v, np, np.` then backtracks and does all the work for recognising the v and np a second time. So the chart parser is faster for large sentences which cause a lot of backtracking being processed with other parsers. However, I measured the time over all examples I have.

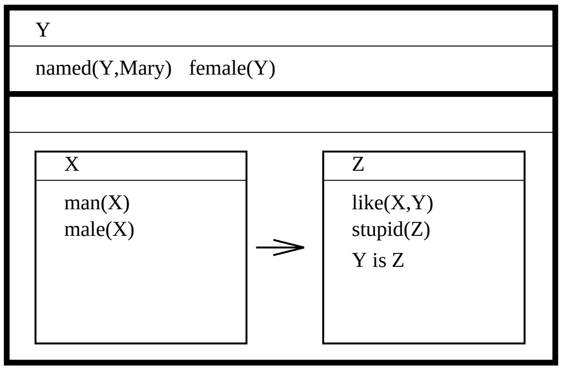

Figure 1: *Every man likes Mary, who is stupid.* The old DRS-Approach, *stupid(Y)* and *female(Y)* are hidden.

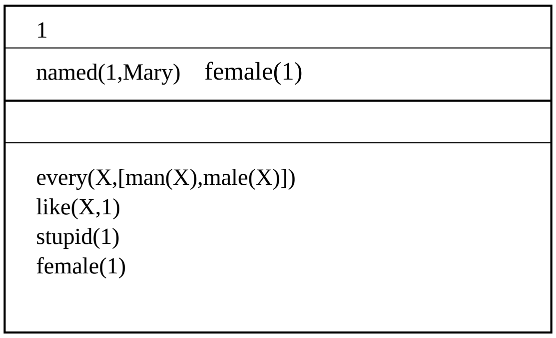

Figure 2: *Every man likes Mary, who is stupid.* The modified notation.

Part III
Semantic

14 The Formalism

14.1 Motivation

The DRS approach would represent the sentence *Every man likes Mary, who is stupid.* in the kind of Figure 1.

The problem here is that *stupid(Z)* is not accessible in the outer DRS, but it should be. Problem two with the old approach is the realisation of quantifiers. There is no handling of quantifier scope in the DRS-theory so far.

14.2 The New Approach

I decided to change the notation into what Figure 2 shows.

1
named(1,John) male(1)
2
every(X,[man(X),male(X),fat(X),afraid(X) of 1]) like(X,2) woman(2) female(2) beautyful(2)

Figure 3: *Every fat man who is afraid of John likes a beautiful woman.* Every man likes the same woman.

1
named(1,John) male(1)
every(X,[man(X),male(X),fat(X),afraid(X) of 1]) some(Z,[woman(Z),female(Z),beautiful(Z),like(X,Z)])

Figure 4: *Every fat man who is afraid of John likes a beautiful woman.* The men don't have to like the same woman.

Therefore the information about the stupidity of Mary is accessible. The sentence *Every fat man who is afraid of John likes a beautiful woman.* Has two different notations depending on the scope of *every*. They are shown in Figure 3 and Figure 4.

How this notations are found by using several arguments which one threads through the parse tree will be explained in the following sections.

14.3 Quantifier

In the notation presented above the all quantifier scopes over the whole dialog. It scopes over the information known so far and over information which will follow.

For the fact that *Every man loves a woman.* (lets say the same) the representation would be:

```
every(X,[man(X),male(X)])
love(X,2)
woman(2)
female(2)
```

There could be other facts in the DRS like *cat(3),neuter(3)* .... So that the scope of *every* over the whole dialog would lead to the following statement:

$\forall X(X \in man \wedge X \in male \rightarrow 3 \in cat)$ Since the desired information is present as well and the statement above doesn't cause problems we can use this notation.

15 Definite vs. Indefinite Articles

The following examples given by Heinz Giegerich in one of his lectures about English grammar show that the distinction between definite and indefinite articles is not that clear as it might seem. Definite articles can be used with a indefinite meaning and vice versa.

```
She needs a doctor.
She talked to a doctor.

The whale is an extincting species.
The whale was beached.
```

Another problem are adjectives. The can function restrictive or add information.

```
S1: I have two books for you here.
    One is interesting the other isn't.

S2: Well, put the interesting one on the table.
```

But:

```
S1: I have a book for you.

S2: Yes, I know it.
    Put the bloody book on the table over there.
```

People use intonation to get the wished meaning if the sentence would be ambiguous otherwise.

However, every time the article *a* is used in a non-quantified statement my program introduces a new instance. This means that *she* needs a particular doctor. Adjectives are used restrictive.

16 Pronoun Resolution

How complex the task of pronoun resolution really is one can read in Sidner [4] and Groz [3].

There are even examples where the pronoun *I* doesn't refer to the same person. Imagine the following situation: Some student did not log out properly and some other nasty student uses his login to send the following mail to the head of department:

```
Hi Dear,
I am not the one you might think I am.
I did simply not log out so that somebody else
is using my login.
```

The first *I* refers to the nasty student and the second to the inattentive.

Another example that a one hundred per cent correct resolution is not possible without world knowledge is the following:

```
1   Marry showed Bill to John.
2 a He is a really nice baby.
2 b He caressed the baby.
```

In 2 a *he* refers to Bill whereas it refers in 2 b to John. To get a right resolution for these pronouns one would need information about babies, relations between them and their environment like *can be showed, can be caressed* and so on. This however, is not possible for a whole language because this would lead to an explosion of data to be kept in the background.

I deal only with a part of the class of objects pronouns can refer to.

17 Focus

As B Groz [3] pointed out in her paper to handle discourse focus correct one had to consider the situation in which the discourse takes place. This includes:

1. preceding discourse context
2. the environment
   (a) physical environment
   (b) social setting
   (c) relationship between the participants of the dialog

She distinguishes between explicit and implicit foci. Explicit foci are things directly mentioned. Implicit foci are things related to them as there are *the owner*, *the windows*, and *the garden* if one speaks about a house.

This again shows the need of world knowledge and works for restricted domains only.

18 Nested Phrases and Modal Adjectives

The sentence *John said that he wants to help Scotland.* will be presented in the form Figure 8 shows. The information about what John said must not be

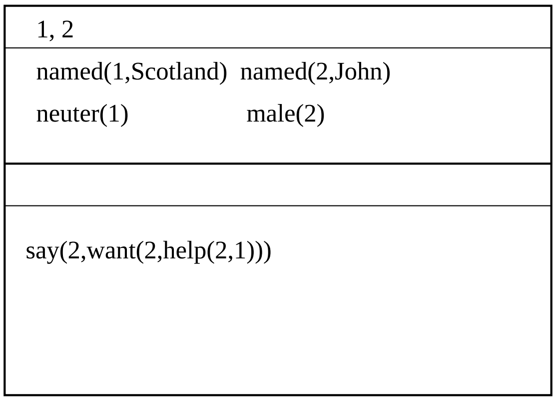

Figure 5: *John said that he wants to help Scotland.* Information hiding in nested phrases

accessible because the only fact the world knows is that he has said something. We don't know anything about the truth of his statements. Not to speak about Bill.

I ignore here that the facts the speaker utters could be untrue as well. The correct representation should be:

```
say(speaker,say(j,want(j,help(j,s)))).
```

In Germany there has been a court case about the sentence *Every soldier is a possible murderer.* The question was whether *Every soldier is a murderer.* follows from the first statement.

This shows the need to hide the information *murderer(X)*. The representation for the sentence above is:

```
every(X,[soldier(X),male(X)])
X is X
possible murderer(X)
```

19 Questions

My program uses a kind of three state logic. Facts are true, false, or unknown.

20 Implementation

20.1 Variable Binding and Late Evaluation

The old DRS approach uses variables to express relations between instances. I decided to change that. If there is one particular woman. She gets a number which can be used to refer to this woman. If a general statement about women is made then a variable is used for the women. This makes the question answering (20.5.1) easier.

Binding information is carried around the parse tree and the binding happens at each stage where the rules provide the necessary information. For ditransitive verbs the binding features in the vp rule are these:

```
vp1(A0,G0,G,vp(V,NP1,NP2)) -->
                 verb(A1,ditrans,V), np(A2,G0,G1,NP1),
                 np(A3,G1,G,NP2),
  {
  ...

  A1:binding   <=> Z^Y^X^Cond,
  A2:binding   <=> Y,
  A3:binding   <=> Z,
  A0:binding   <=> X^Cond,

  ... }.
```

The X will be bound by a NP to the left of the VP. Normally it is bound at that stage already because we have traversed a NP node before the VP node.

The binding of such arguments during parsing a sentence happens with variables although Y might be bound to 3 if 3 stands for *John* (as in *Bill gave John a book.*) The reason for that is that all tests which have to be performed to get a plausible pronoun resolution and a correct reference of definite articles have to be performed at the highest level that means after passing the *s_eng* node. We could not make the tests on a lower level because I didn't want to allow DCG rules like

```
s -->    {ListNpIn = []},
      np,{ListVpIn = ListNpOut},
      vp,{ListSOut=ListVpOut}.
```

My rules do have the form:

```
s --> np, vp,
  {
  ListNpIn = [],
  ListVpIn = ListNpOut,
  ListSOut=ListVpOut}.
```

The lists are therefore not initialised while they are threaded through the tree. No test of membership or any similar can be made. I pass all tests which have to be made in a restriction list up to the highest node of the tree.

One might think that is a lot of unnecessary work. But it is just the other way round. If one uses the chart parser I introduced in Chapter 12 all syntax checks will be made first and the semantic restrictions will be evaluated in a second pass. This saves a lot of time.

After checking the restrictions I instantiate the variables by using the *bind(List)* predicate.

20.1.1 Copula Constructions

The only copula verb my grammar coverages is *is*. Therefore there exists an identity which I can express using an *X is X* entry in the DRSs. The case would be different for copula verbs like *become* or *went*. In this cases the variables must be different.

The *is* entry in the DRSs is necessary to make questions like *Who am I?* possible.

20.2 Threading

I thread several arguments through the tree of a sentence:

- the global DRS
- a normal DRS
- a local DRS
- a focus list
- a scope list
- properties of articles
- a local list
- restrictions

The global DRS has the form [Dom,Constr]. In *Dom* numbers of persons or other proper nouns are stored. *Constr* contains the information for each such number. One might think that it isn't necessary anymore to have a global DRS because the information could be stored in the normal DRS as well. This is true. The only exception are questions: a person can be mentioned in a question the first time in a discourse. Then it would make sense to refer to that person in later sentences. This is only the case for proper nouns. If we are asking questions about a book the book must be introduced already.

The normal DRS contains all information that doesn't relate to proper nouns. In contrast to the local DRS it contains the complete information of the dialog so far.

The local DRS contains only the information of the actual sentence.

The focus list contains possible foci. The most likely foci are placed before unlikely foci in the list.

The scope list becomes filled during passing the lexicon entries of determiners. It contains all information which might be needed in a scoping procedure.

The local list contains information about the most recent noun phrases and is used to avoid a wrong pronoun resolution (to distinguish between *him* and *himself*).

The restriction list contains all semantic checks which have to be made for pronoun resolution and definite articles.

20.3 Gap-Threading

There is a very elegant way to thread arguments if one uses DAGs: One declares a feature in each DAG named *in* and one named *out*. Then all one has to write down is:

```
s(A0) --> np(A1),vp(A2),
 {
 A1:in  <=> A0:in,
 A2:in  <=> A1:out,
 A0:out <=> A2:out }.
```

Until one doesn't want to manipulate one of the features directly when they pass the rule one can use this notation. If one has to manipulate then one must write all unmanipulated threaded features explicitly down.

In order to recognise sentences like *Himself, John likes.* correct (or to recognise them at all) one needs a complicated threading of the appropriate lists. Figure 6 visualises this.

The rule for the NP-gap looks therefore in the following way:

```
np(NP,gap(np(NP)),nogap,np('_')) --> [].
```

The rule for topicalised sentences is quite simple:

```
s(A0,nogap,nogap,s(NP,comma(','),S)) --> np(A1,nogap,nogap,NP),
                                                        [','],
                                     s(A0,gap(np(A1)),nogap,S),
  {
  A1:cas   <=> acc,
  A0:vform <=> fin}.
```

But if it comes to questions which are using the gap-concept as well the matter gets more difficult:

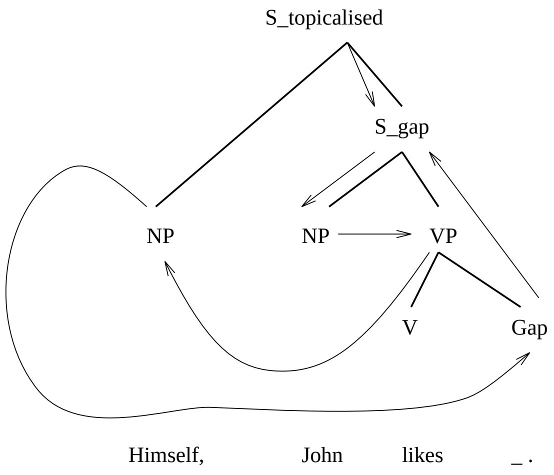

Figure 6: The Gap-Threading

```
que(A0,que(interr(who),Q)) --> who_or_what,
                                que(A1,gap(np(NPGap)),nogap,Q),
  {
  NPGap <=> sing3,

  NPGap:binding <=> X,

  NPGap:in:gl_DRS <=> NPGap:out:gl_DRS,
  NPGap:in:locDRS <=> NPGap:out:locDRS,
  NPGap:in:drs    <=> ... ,  % not important yet
  NPGap:out:drs   <=> ... ,
  NPGap:in:focus  <=> NPGap:out:focus,
  NPGap:locListIn <=> LocList,
  NPGap:locListOut<=> [X|LocList],

  ...

  }.
```

If all arguments are threaded through a NP it is OK. But if there is no NP like in the case of questions or relative clauses we have to make the connection, to fill the local list, and to take care of the binding.

20.4 Quantifier

I am threading a scope list through the parse tree which contains after recognising a complete sentence all quantifiers which could be needed for a scoping procedure. Entries in this list are made if it is threaded through a lexical entry of a determiner. For the sentence *Every man loves a woman.* the scope list would be: [every(A,B),some(C,D) & woman(C)]. The scope list for *Every fat man loves a fat woman.* would be exactly the same.

B and D will be getting filled by the scoping procedure. The information *woman(C)* which is provided by the scope list can be used in the scoping procedure to ask the user for the correct scope rather then give him some logical forms to decide which is the correct one.

In order not to bother the user with questions one could postpone the scoping and scope when it becomes clear from the further dialog which scope is meant. The classical donkey sentences:

```
Every man owns a donkey.
It loves them.
```

could be handled that way.

20.4.1 Quantifier and Copula Constructions

For general statements like *Every man is a lunatic.* only one scope is possible. The representation would be:

```
every(X,[man(X),male(X)])
X is X
lunatic(X)
```

20.4.2 A Simple Scoping Procedure

Depending on the information on fills in the scope list during traversing the tree this approach could use the Hobbs and Shieber [8] scoping algorithm. My procedure here deals only with simple cases.

If a flag is set which expresses the necessaryty for scoping then after parsing the complete sentence the scope list the domain and the local constraint are passed to the scoping procedure.

```
scope([some(X,Y) & Description|Scopes],Sin,Sout,DomIn,DomOut,Cin,Cout) :-
    ask_scope(some(X,Y) & Description,Sin,S1,DomIn,D1,Cin,C1),
    scope(Scopes,S1,Sout,D1,DomOut,C1,Cout).

scope([QTerm|Scopes],Sin,[QTerm|Sout],
       DomIn,DomOut,Cin,[plural(Var)|Cout]) :-
    QTerm =.. [_Quantifier,Var,Scope],

    % use the scoped QTerm in stead of the
    % unscoped entries in Cin
    % Cin is a part of a local constraint
```

```
    % which contains all information (unscoped)

    del_list(Scope,Cin,C1),
    scope(Scopes,Sin,Sout,DomIn,DomOut,C1,Cout).

ask_scope(some(X,Y) & Description,Sin,Sout,DomIn,DomOut,Cin,Cout) :-
    Description =..[Desc,_],
    write('Is there only one '),
    write(Desc),
    write(' ? (y/N): '),
    get0(C),
    (C = 10 -> true;get0(_)),
    (C =121 -> Cin = Cout,  DomOut = [X|DomIn],
               Sin = Sout
            ;  del_list(Y,Cin,Cout),
               Sout = [some(X,Y)|Sin],
               DomOut = DomIn).
```

If there are *some* terms in the scope list the user is asked for disambiguation. If there is only one individuum ($\exists X \forall Y$) the variable X is entered in the domain. Otherwise ($\forall Y \exists X$) the *some* term is taken over into the *Sout* list to be merged later with the constraints of the local and the normal DRS.

The second *scope* clause does not only cope with *every* terms. It handles *most* and other imaginable quantifiers as well.

The entry of the quantifier is simply taken from the scopelist and entered into *Sout*. All information stored in the input constraint has to be removed for it is hidden in the antecedens of the quantifier.

20.5 Questions

There are basically two types of questions. Yes/No questions and questions which ask for information. The latter are treated like Yes/No questions or sentences with gaps. If the question contains a phrase with a determiner *a* this will be represented as *some(...). Is John afraid of a dog?* for instance will be represented as yes^[2 is 2, afraid(2) of X,some(X,[dog(X),neuter(X)])]

The questions *Which dog is John afraid of?*, which asks for information, has the representation: X^[2 is 2, afraid(2) of X,some(X,[dog(X),neuter(X)])]

For writing the answering procedure one had to be aware about the following conclusions:

$\forall X(X \in man \wedge X \in male \wedge X \in fat \rightarrow like(X,3)$

$\rightarrow \forall X(X \in man \wedge X \in male \wedge X \in fat \wedge X \in stupid \rightarrow like(X,3))$

but not

$\forall X(X \in man \wedge X \in male \rightarrow like(X,3))$

$\forall X(X \in man \wedge X \in male \rightarrow \exists Y \in woman \wedge Y \in beautiful \rightarrow love(X,Y))$
$\rightarrow \forall X(X \in man \wedge X \in male \rightarrow \exists Y \in woman \rightarrow love(X,Y))$
but not
$\forall X(X \in man \wedge X \in male \rightarrow \exists Y \in woman \wedge Y \in beautiful \wedge Y \in intelligent \rightarrow love(X,Y))$

20.5.1 The Answering Procedure

The answering procedure is a tricky bit code. The problem one has to be aware of are the variables in the general statements. If one checks the statements for a particular instance the variables become bound and the original meaning of the DRS is destroyed. So one has to find a possibility to bind the variables in a way that the variable of the solution is bound to an instance but the variables in the DRS are not.

There are four cases:

1. A question about simple facts.
2. A question which formula contains a *some* term.
   (a) The solution is a particular instance.
   (b) The solution is a following of a general statement.
3. A question which formula contains a *every* or *most* term.

The actual answer procedure distinguishes between two cases. A question is either a Yes/No question or a question which asks for information:

```
answer(Q) :-
        Q:question <=> [X,QConstr],
        var(X),

        Q:drsIn    <=> [_Dom,Constr],
        Q:gl_DRSIn <=> [_Vars,Global_names],
        assert(allow_refresh),
        setof(X,
                solution(X,QConstr,Constr),
                Solutions),
        nl,
        print_solution(Solutions,QConstr,Global_names,Constr).

answer(Q) :-
        Q:question <=> [yes,QConstr],
        Q:drsIn    <=> [_Dom,Constr],
```

```
        solution(_X,QConstr,Constr),
        nl,write_out("Yes, that's right."),nl,nl.


answer(Q) :-
        Q:question <=> [yes,_QConstr],
        nl,write_out("As far as I know, no."),nl,nl.

answer(_) :-  nl,write_out("I don't know."),nl,nl.
```

The first clause of *answer/1* uses *setof* to find all solutions which are found by *solution/3* and prints them out afterwards.

The second clause of *answer* tests whether there is any solution which satisfys all constraints made and stored in *QConstr*. Which solution is found doesn't matter.

The negative answers to each kind of questions are provided by the third and fourth clause of *answer/1*.

```
refresh(Constr,FreshConstr) :-
         asserta(fresh(Constr)),
         retract(fresh(FreshConstr)).

solution(X,QConstr,Constr) :-
         refresh(Constr,FreshConstr),
         solution_(X,QConstr,FreshConstr).
```

Using assert and retract I provide a DRS with fresh variables about which I don't have to care if they get bound. Then the actual solution clause *solution_/3* is called.

```
solution_(X,[some(Y,Cond)|QConstr],Constr) :-
        find_term(some,Constr,some(Y,Scope)),
        members(Cond,Scope),
        solution_(X,QConstr,Constr).
```

This clause is meant to handle case 2.b. See Figure 7.

The solution follows from a general statement. We have to check the *some* expression of the question first, *members* instantiates variables in *QConstr* and the recursive call of *solution_(X,QConstr,Constr)* checks the antecedens after finding the verb in *QConstr*.

```
solution_(X,[some(_X,Cond)|QConstr],Constr) :-
        xappend(Cond,QConstr,NQConstr),
        solution_(X,NQConstr,Constr).
```

1
named(1,John)   male(1)
every(X,[man(X),male(X)]) some(Y,[dog(Y),neuter(Y),afraid(X) of Y]) man(1) male(1)

Figure 7: *John is a man. Every man is afraid of a dog.* Every man can be afraid of a different dog.

1
named(1,John)   male(1)
2
every(X,[man(X),male(X)]) dog(2) neuter(2) afraid(X) of 2 man(1) male(1)

Figure 8: *John is a man. Every man is afraid of a dog.* Every man is afraid of one particular dog.

This is the clause which deals with case 2.a. See Figure 8.

All restrictions in the scope of the *some* are appended to the list of the other information to be checked.

```
solution_(X,[Pred|QConstr],Constr) :-
        Pred =.. [Fun1|Args1],
        Fun1 \= some,
        find_term(Fun1,Constr,Answer),
        Answer =.. [Fun1|Args2],
        check_lists(Args1,Args2,Constr),
        solution_(X,QConstr,Constr).
```

Case 1. The question doesn't contain a *some* clause. We have though to find each simple term of the question list in the DRS and then to check whether the arguments of the term in the question list and the arguments of the term in the DRS match in some sense. This is done by *check_lists/2* and can involve antecedens tests.

```
solution_(S,[QTerm|QConstr],Constr) :-
        QTerm =.. [Quantifier,X,ScopeQ],
        quantifier(Quantifier),
        find_term(Quantifier,Constr,QTermC),
        QTermC =.. [Quantifier,X,ScopeC],
        members(ScopeC,ScopeQ),
        xappend(ScopeC,Constr,TConstr),
        solution_(S,QConstr,TConstr).
```

Case 3. The question contains an exists or an most quantifier or something of that kind. It has to be checked whether all conditions in the antecedens are full filled. If this is the case these are appended to the temporary DRS. This is necessary to make the antecedens check which will take palace later succeed.

```
check_lists([Vq|R1],[Vd|R2],Constr) :-
        (
            % Vq is either instantiated
            % then it refers to a global instance
            % or we are asking for it
            number(Vd),!,Vq=Vd
        ;
            % Vd is a term we are asking for
            % like in relative clauses
            % What did Mary say?
            var(Vq),nonvar(Vd),!,
            Vq = Vd
```

```
        ;
            % Vd is a quantified variable
            % and we have to check the
            % antecedens
            var(Vd),
            !,
            check_antecedens(Vq,Vd,Constr)
        ),
        check_lists(R1,R2,Constr).

check_lists([V1|R1],[V2|R2],Constr) :-
        V1 =.. [Fun|Args1],
        V2 =.. [Fun|Args2],
        check_lists(Args1,Args2,Constr),
        check_lists(R1,R2,Constr).
```

Because terms can be nested as in prepositional phrases (*afraid(3) of 4*) the second clause of *check_lists/2* is needed.

```
check_antecedens(Vq,Vd,Constr) :-
        member(QTerm,Constr),
        QTerm =.. [Quantifier,Vd,Cond],
        quantifier(Quantifier),
        ( var(Vd) ->
          sub(Vq,Vd,Cond,TCond)
        ;
          Vq = Vd,Cond = TCond
        ),
        check_members(TCond,Constr).
```

check_antecedens searches for an entry of a quantifier (*most* or *every*) which quantifies over *Vd*. Because of the *members/2* call in the *solution_/3* clause for case 2.b Vd can be instantiated. If it is not we have to make sure that we don't instantiate it and therefore we are substituting the variable Vd by Vq. Vq will contain the information we are eventually asking for (if we deal with a question which is not a Yes/No question).

20.6 Definite and Indefinite Articles and Discourse

Implementing definite articles I assumed that they are used in a definite way. That means that the discourse refers to objects mentioned during the conversation so far. Therefore a check has to be made if there is a object which has all properties the noun phrase preceded by the definite article has. This includes adjectives restrictive relative clauses and prepositional phrases.

20.7 Focus

I thread a focus list through the parse tree which contains references to objects mentioned in the discourse ordered in a way that the reference to the most recent mentioned object is at the very beginning of the focus list.

Every time the focus list is threaded through an entry for a noun (proper nouns included) the number which will be (or is) used to refer to the mentioned object is entered in the focus list at the beginning.

If we refer to an object using a pronoun no entry is made. The pronoun resolution takes the first member of the focus list which matches the gender of the appropriate pronoun. Therefor a new entry would not really change anything.

If we refer to an object using definite articles the entry in the focus list is made automatically by passing the noun entry. The checks made for finding the object we refer to will instantiate the variable in the focus list so that the object is reintroduced and is situated at the beginning of the focus list.

20.8 Templates for Semantic

All the things which happen while threading through a lexical entry I have hidden in semantic templates.

A lexicon entry for *woman* looks that way:

```
n(A,n(woman)) --> [woman],
  {
  A <=> aSing3Human,
  A <=> noun(woman,female)}.
```

The template for *noun* is more complex:

```
template(noun(Description,Gender),NOUN) :-
  Pred =.. [Description,X],
  Gen  =.. [Gender,X],
  NOUN:in:gl_DRS   <=> NOUN:out:gl_DRS,
  NOUN:in:scopeList<=> NOUN:out:scopeList,
  NOUN:in:focus    <=> Focus,
  NOUN:out:focus   <=> [X|Focus],
  NOUN:in:drs      <=> [Dom,Constr],
  ( % article indefinite -> new object is introduced
      NOUN:article <=> indefinite,
      % add information
      NOUN:out:drs   <=> [[X|Dom],[Pred,Gen|Constr]],
      NOUN:in:locDRS <=> [LocDom,LocConstr],
      (NOUN:quantified <=> no ->
       NOUN:out:locDRS  <=> [[X|LocDom],[Pred,Gen|LocConstr]]
```

```
      ;
       NOUN:out:locDRS  <=> [[X|LocDom],[Gen|LocConstr]]
      ),
      NOUN:restr   <=> []
  ;
    % article definite -> refer to an existing object
      NOUN:article <=> definite,
      % check information
      NOUN:out:drs    <=> NOUN:in:drs,
      NOUN:out:locDRS <=> NOUN:in:locDRS,
      NOUN:restr     <=> [check(X,[Pred,Gen],[Dom,Constr])]
  ),
  NOUN:binding    <=> (X^Pred).
```

20.9 Pronoun Resolution

In order to find an instance a pronoun can refer to the program chooses a member of the focus list and checks whether there is an entry of the appropriate gender in the local in the normal or in the global DRS.

A further check of membership in the local list is necessary to avoid *him* in *John likes him.* to refer to *John*.

```
template(pronoun(Gender),PN) :-
  Gen  =.. [Gender,X],
  PN:in:gl_DRS  <=> PN:out:gl_DRS,
  PN:in:drs     <=> PN:out:drs,
  PN:in:locDRS  <=> PN:out:locDRS,
  PN:in:gl_DRS  <=> [_Gl_Dom,Gl_Constr],
  PN:in:drs     <=> [_Dom,Constr],
  PN:in:locDRS  <=> [_LocDom,LocConstr],
  PN:in:focus   <=> Focus,
  PN:out:focus  <=> [X|Focus],
  PN:locListIn <=> LocList,
  PN:locListOut<=> [X|LocList], % reintroduced
  PN:binding    <=> X,
  PN:restr      <=> [ member(X,Focus),
                      \+ strict_member(X,LocList),
                      ( strict_member(Gen,LocConstr)
                      ; strict_member(Gen,Constr)
                      ; strict_member(Gen,Gl_Constr))
                    ].
```

The template for reflexive pronouns like *herself*, *himself* and *itself* checks whether the X which is bound to a member of the local list (which can contain at most 2 elements) is a strict member of the appropriate DRSs.

```
template(refl_pronoun(Gender),PN) :-
  Gen  =.. [Gender,X],
  PN:in:gl_DRS  <=> PN:out:gl_DRS,
  PN:in:drs     <=> PN:out:drs,
  PN:in:locDRS  <=> PN:out:locDRS,
  PN:in:focus   <=> PN:out:focus,  % instance is still in focus
  PN:in:gl_DRS  <=> Gl_DRS,
  PN:in:drs     <=> DRS,
  PN:in:locDRS  <=> LocDRS,
  PN:locListIn <=> LocList,
  PN:locListOut <=> LocList,
  PN:binding    <=> X,
  (LocList = [X|_]; LocList = [_,X]),
  PN:restr      <=> [ (strict_check(X,[Gen],LocDRS)
                       ;
                       strict_check(X,[Gen],DRS)
                       ;
                       strict_check(X,[Gen],Gl_DRS))
                    ].
```

21 Prepositional Phrases

Prepositional phrases are a hard problem for computational linguistics because of their ambiguity.

I use a procedure which asks the user for disambiguation. This procedure is called at the end of the syntactic analysis. If a preposition is found as a modifier of an NP by the syntax analysis and the NP contains a definite article this would mean that a check has to happen, whether the instance the definite article refers to has the properties of the appropriate prepositional phrase. (*The man in the house ...* has to be *in the house.*) But if the prepositional phrase relates to the VP non of these restrictions has to be tested. (*He loved the woman in the house.*)

Therefore I postpone the prepositional checks until the prepositional binding took place. Then if the preposition is bound to the VP the check will be removed from the list of remaining checks and the parsing succeeds. Otherwise it stays there and will be processed.

One could argue that it is not necessary to postpone the check because if the check for the definite article failed backtracking would chose automatically the right modifier namely the VP.

```
John owns a house.
```

```
He lives in the house with a woman.
```

This would be correct for the example given above. But it would fail for the following discourse:

```
1 John lives in a house with a garden.
2 He owns a cat.
3 This cat chases another cat in the garden.
4 John saw the cat in the garden.
```

In sentence 3 *garden* is supposed to modify *cat*.

The check would be made for *the cat in the garden* and one would not have the chance to bind the *garden* to *see*.

22 Discussion of an Example

```
| ?- tdisc(14).
Every man is afraid of a dog.

Is there only one dog ? (y/N): y
yes
Focus: [1,Z]

global DRS: []

       DRS: [1]
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1

Is Bill afraid of a dog?

As far as I know, no.

Focus: [_63313,2,1,Z]

Bill is a man.

yes
Focus: [2,_88173,1,Z]
```

```
global DRS: [2]
       named(2,Bill)
       male(2)

       DRS: [2,1]
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1

Is Bill afraid of a dog?

Yes, that's right.

Focus: [1,2,_116033,Z]

global DRS: [2]
       named(2,Bill)
       male(2)

       DRS: [2,1]
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1
```

Note that the dog we were asking for is now in focus.

```
Is Bill afraid of the dog?

Yes, that's right.

Focus: [1,2,_17731,Z]

global DRS: [2]
       named(2,Bill)
       male(2)
```

```
       DRS: [2,1]
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1

The dog is fat.

yes
Focus: [1,2,_38620,Z]

global DRS: [2]
       named(2,Bill)
       male(2)

       DRS: [2,1]
       fat(1)
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1

Who is afraid of the dog?

Bill


Focus: [1,2,_63204,Z]

global DRS: [2]
       named(2,Bill)
       male(2)

       DRS: [2,1]
       fat(1)
       man(2)
```

```
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1

Another dog chases a cat.

yes
Focus: [3,4,1,2,_94483,Z]

global DRS: [2]
       named(2,Bill)
       male(2)

       DRS: [3,4,2,1]
       cat(3)
       neuter(3)
       chase(4,3)
       dog(4)
       neuter(4)
       fat(1)
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1

John is afraid of this dog.

yes
Focus: [4,5,3,1,2,_124292,Z]

global DRS: [5,2]
       named(5,John)
       male(5)
       named(2,Bill)
       male(2)

       DRS: [3,4,2,1]
```

```
        afraid(5)of 4
        cat(3)
        neuter(3)
        chase(4,3)
        dog(4)
        neuter(4)
        fat(1)
        man(2)
        male(2)
        every(Z,[man(Z),male(Z)])
        plural(Z)
        dog(1)
        neuter(1)
        afraid(Z)of 1

Which dog is John afraid of?


afraid(5)of 4   chase(4,3)   dog(4)   neuter(4)


Focus: [5,4,3,1,2,_147255,Z]

global DRS: [5,2]
        named(5,John)
        male(5)
        named(2,Bill)
        male(2)

        DRS: [3,4,2,1]
        afraid(5)of 4
        cat(3)
        neuter(3)
        chase(4,3)
        dog(4)
        neuter(4)
        fat(1)
        man(2)
        male(2)
        every(Z,[man(Z),male(Z)])
        plural(Z)
        dog(1)
        neuter(1)
```

```
       afraid(Z)of 1

Is John afraid of the fat dog?

As far as I know, no.

Focus: [1,5,4,3,2,_180643,Z]

global DRS: [5,2]
       named(5,John)
       male(5)
       named(2,Bill)
       male(2)

       DRS: [3,4,2,1]
       afraid(5)of 4
       cat(3)
       neuter(3)
       chase(4,3)
       dog(4)
       neuter(4)
       fat(1)
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1
```

The fat dog is in focus.

```
Is John afraid of the dog which chases a cat?

Yes, that's right.

Focus: [3,4,5,1,2,_231660,Z]

global DRS: [5,2]
       named(5,John)
       male(5)
       named(2,Bill)
       male(2)
```

```
       DRS: [3,4,2,1]
       afraid(5)of 4
       cat(3)
       neuter(3)
       chase(4,3)
       dog(4)
       neuter(4)
       fat(1)
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1

John is a man.

yes
Focus: [5,3,4,1,2,_258139,Z]

global DRS: [5,2]
       named(5,John)
       male(5)
       named(2,Bill)
       male(2)

       DRS: [5,3,4,2,1]
       man(5)
       male(5)
       afraid(5)of 4
       cat(3)
       neuter(3)
       chase(4,3)
       dog(4)
       neuter(4)
       fat(1)
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
```

```
       neuter(1)
       afraid(Z)of 1
```

John is afraid of the dog every man is afraid of and of the dog which chases a cat.

```
Which dog is John afraid of?

fat(1)   dog(1)   neuter(1)   afraid(_129)of 1
afraid(5)of 4   chase(4,3)   dog(4)   neuter(4)


Focus: [5,3,4,1,2,_21184,Z]

global DRS: [5,2]
       named(5,John)
       male(5)
       named(2,Bill)
       male(2)

       DRS: [5,3,4,2,1]
       man(5)
       male(5)
       afraid(5)of 4
       cat(3)
       neuter(3)
       chase(4,3)
       dog(4)
       neuter(4)
       fat(1)
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1

Who is afraid of a dog?


Bill
John
```

```
Focus: [1,5,3,4,2,_47879,Z]

global DRS: [5,2]
       named(5,John)
       male(5)
       named(2,Bill)
       male(2)

       DRS: [5,3,4,2,1]
       man(5)
       male(5)
       afraid(5)of 4
       cat(3)
       neuter(3)
       chase(4,3)
       dog(4)
       neuter(4)
       fat(1)
       man(2)
       male(2)
       every(Z,[man(Z),male(Z)])
       plural(Z)
       dog(1)
       neuter(1)
       afraid(Z)of 1


yes
| ?- tdisc(17).
Every man who owns a big knife is a possible murderer.

yes
Focus: [Y,Z]

global DRS: []

       DRS: []
       every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
       plural(Z)
       possible murderer(Z)
```

Note that there is only the entry *possible murderer(Z)* and not *murderer(Z)*.

```
John is a man.

yes
Focus: [1,Y,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: [1]
       man(1)
       male(1)
       every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
       plural(Z)
       possible murderer(Z)

Is John a murderer?

As far as I know, no.

Focus: [1,Y,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: [1]
       man(1)
       male(1)
       every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
       plural(Z)
       possible murderer(Z)

Is John a possible murderer?

As far as I know, no.

Focus: [1,Y,Z]

global DRS: [1]
       named(1,John)
```

```
        male(1)

        DRS: [1]
        man(1)
        male(1)
        every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
        plural(Z)
        possible murderer(Z)

John owns a knife.

yes
Focus: [2,1,Y,Z]

global DRS: [1]
        named(1,John)
        male(1)

        DRS: [2,1]
        knife(2)
        neuter(2)
        own(1,2)
        man(1)
        male(1)
        every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
        plural(Z)
        possible murderer(Z)

Is John a possible murderer?

As far as I know, no.

Focus: [1,2,Y,Z]

global DRS: [1]
        named(1,John)
        male(1)

        DRS: [2,1]
        knife(2)
        neuter(2)
        own(1,2)
        man(1)
```

```
       male(1)
       every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
       plural(Z)
       possible murderer(Z)

The knife is very big.

Focus: [2,1,Y,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: [2,1]
       big(2)
       knife(2)
       neuter(2)
       own(1,2)
       man(1)
       male(1)
       every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
       plural(Z)
       possible murderer(Z)

Is John a murderer?

As far as I know, no.

Focus: [1,2,Y,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: [2,1]
       big(2)
       knife(2)
       neuter(2)
       own(1,2)
       man(1)
       male(1)
       every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
       plural(Z)
```

```
       possible murderer(Z)
```

All conditions are satisfied.

```
Is John a possible murderer?


Yes, that's right.

Focus: [1,2,Y,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: [2,1]
       big(2)
       knife(2)
       neuter(2)
       own(1,2)
       man(1)
       male(1)
       every(Z,[man(Z),male(Z),knife(Y),neuter(Y),big(Y),own(Z,Y)])
       plural(Z)
       possible murderer(Z)


yes
| ?- tdisc(38).
Every stupid woman likes John.

yes
Focus: [1,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: []
       every(Z,[woman(Z),female(Z),stupid(Z)])
       plural(Z)
       like(Z,1)
```

```
Does every woman like John?

As far as I know, no.

Focus: [1,_57258,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: []
       every(Z,[woman(Z),female(Z),stupid(Z)])
       plural(Z)
       like(Z,1)

Does every stupid woman like John?

Yes, that's right.

Focus: [1,_88298,_88300,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: []
       every(Z,[woman(Z),female(Z),stupid(Z)])
       plural(Z)
       like(Z,1)

Does every stupid fat woman like John?

Yes, that's right.

Focus: [1,_121792,_121794,_121796,Z]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: []
       every(Z,[woman(Z),female(Z),stupid(Z)])
```

```
    plural(Z)
    like(Z,1)


yes
```

(see Dialog Examples C.4).

23 Problems / Work to Be Done

23.1 Gender of *Who*

While introducing the gap for the question I make entries for each gender in the DRS. This allows questions like: *Who likes himself?* and *Who likes herself?*.

The answering procedure doesn't check the gender of the found solution so that John is an answer to *Who does like herself?*

23.2 Prepositional Phrases

The binding procedure for prepositional phrases is not very good. It asks the user for disambiguation even in cases which are not ambiguous. (*A man in a garden sleeps.*) One could have written a procedure which uses parse trees but I wanted to avoid this because I did not want to carry the tree information around while using the chart parser.

The program does not understand prepositional phrases in relative clauses or in quantified sentences.

23.3 Definite Articles

The sentence *The man I gave a book to lost it.* is not understood. This is because *a* introduces a new object in stead to allow to refer to a recently mentioned object.

23.4 Focus

.

Questions can introduce variables into the focus list which don't relate to any object.

23.5 The Determiner All

The sentences *Every man loves a woman.* and *All men love a woman.* should have the same representation but they don't. Thats due to the way the structures are build up. *men* are plural. Therefore the notation is every(X,[man(X),plural(X)]) in stead of every(X,[man(X),male(X)]). This causes problems if one wants to answer questions.

A Syntax

A.1 The Grammar

```
%-------------------------------------------------------
%
% File: grammar
%
% Stefan M"uller
%
% Large Practical
%
% Part I & III
%
% Syntax and Semantic for a fragment of English
%
% The Grammar
%
% Version 11.03.93
%
%
%-------------------------------------------------------

%-------------------------------------------------------
% sentences
%-------------------------------------------------------


s_eng(A0,S_tree) --> s_fin(A0,S_tree),mark('.',_).

s_fin(A0,S_tree) --> s(A1,nogap,nogap,S1_tree),
                     more_s_fin(A2,S1_tree,S_tree),
  {
  A1:vform <=> fin,

  A1:in:gl_DRS  <=> A0:in:gl_DRS,
  A1:in:drs     <=> A0:in:drs,
  A1:in:focus   <=> A0:in:focus,
  A1:in:locDRS  <=> [[],[]],

  A1:restr <=> A1restr,

xappend([
```

```
  find_later(A1restr,Now,Later),

  eval(Now),          % later has to be done
                      % after binding the prepositions

  A1:out:gl_DRS <=> [GlDom,GlConstr],
  A1:out:drs    <=> [Dom,A1Constr],

  bind(GlDom),            % instantiate new variables
  bind(Dom),
  A1:binding <=> Cond,    % Cond is needed for simple
                          % preposition binding
                          % (actor & patient)

  bind_prepositions(Cond,GlDom,GlConstr,
            A1Constr,A2Constr,Later,LaterRest),

  eval(LaterRest),
  retractall(answer(_,_))

          ],A2restr,A0restr),

  A2:in:gl_DRS  <=> A1:out:gl_DRS,
  A2:in:drs     <=> [Dom,A2Constr],
  A2:in:focus   <=> A1:out:focus,
  A2:restr      <=> A2restr,

  A0:out:gl_DRS <=> A2:out:gl_DRS,
  A0:out:drs    <=> A2:out:drs,
  A0:out:focus  <=> A2:out:focus,
  A0:restr      <=> A0restr

  }.

more_s_fin(A,S,S) --> [],
  {
  A:in <=> A:out}.

more_s_fin(A0,S1,S3) --> coord(and,C), s_fin(A1,S2),
            more_s_fin(A2,coordination(S1,C,S2),S3),
  {
  A1:in <=> A0:in,
  A2:in <=> A1:out,
```

```
  A0:out <=> A2:out }.

s(A0,G0,G,s(NP,VP)) --> np(A1,G0,G1,NP),vp(A2,G1,G,VP),
  {
  A1:cas   <=> nom,
  A1:agr   <=> A2:agr,
  A0:vform <=> A2:vform,

  A1:in      <=> A0:in,
  A1:locListIn <=> [],
  A2:locListIn <=> A1:locListOut,

  A1:binding <=> X,
  A2:binding <=> X^Cond,
  A0:binding <=> Cond,

  A1:restr   <=> RestrNp,
  A2:restr   <=> RestrVp,
  A0:restr   <=> RestrS,
  (A1:out:quantified    <=> no,  % normal threading

          A2:in  <=> A1:out,
          A0:out <=> A2:out,

          xappend(RestrNp,RestrVp,RestrS)
   ;
   not A1:out:quantified    <=> no,

          A0:out:gl_DRS  <=> A2:out:gl_DRS,
          A0:out:focus   <=> A2:out:focus,

          A1:in:scopeList <=> [],

          A2:in:gl_DRS   <=> A1:out:gl_DRS,
          A2:in:drs      <=> A1:out:drs,
          A2:in:locDRS   <=> A1:out:locDRS,
          A2:in:focus    <=> A1:out:focus,
          A2:locListIn  <=> A1:locListOut,
          A2:in:scopeList <=> [],

          % modify the local constraints

          A1:out:scopeList <=> A1ScopeList,
```

```
        A1:out:locDRS    <=> [_,A1LocConstrOut],

        A2:out:scopeList <=> A2ScopeList,

        A2:out:locDRS    <=> [LocDom,A2LocConstrOut],

        A0:in:drs        <=> [A0DomIn, A0ConstrIn],
        A0:out:drs       <=> [A0DomOut,A0ConstrOut],

        A0:out:locDRS    <=> [LocDom,T3],

        xappend(RestrNp,RestrVp,T),
        xappend(T,[
                scope(A1ScopeList,
                  [],A1Scoped,
          A0DomIn,SDomOut,
          A1LocConstrOut,A1NotInScope),

                del_list(A1LocConstrOut,A2LocConstrOut,
                                        ClearedConstrOut),
                scope(A2ScopeList,
          [],A2Scoped,
                  SDomOut,A0DomOut,
          ClearedConstrOut,A2NotInScope),

                  xappend(A1Scoped, A2Scoped,T1),
          xappend(T1,A1NotInScope,T2),
          xappend(T2,A2NotInScope,T3),
          xappend(T3,
                          A0ConstrIn,A0ConstrOut)],RestrS)
      )
   }.


% topicalisation, objects in first position
s(A0,nogap,nogap,s(NP,comma(',',),S)) --> np(A1,nogap,nogap,NP),
 comma(_T),
                                     s(A0,gap(np(A1)),nogap,S),
  {
  A1:cas   <=> acc,
  A0:vform <=> fin}.
```

```
%----------------------------------------------------------
% questions
%----------------------------------------------------------

question(A0,Q_tree) --> que(A1,nogap,nogap,Q_tree),mark(?,_),
  {
  A1:in <=> A0:in,

  A1:out:locDRS <=> [_Dom,Constr],

  A1:restr <=> A1restr,

  A1:out:gl_DRS <=> [GlDom,_GlConstr],

  A0:out:gl_DRS <=> A1:out:gl_DRS,   % assume that questions do
  A0:out:drs    <=> A0:in:drs,       % not contain information
                                    % eccept names of proper
                                    % nouns
  A0:out:focus <=> A1:out:focus,
  A0:restr     <=> A0restr,

  A0:question  <=> [yes,Constr],

  xappend(A1restr,[bind(GlDom)],A0restr)}.

question(A0,Q_tree) --> que(A0,Q_tree),mark(?,_).
question(A0,Q_tree) --> s(A1,nogap,nogap,Q_tree),mark(?,_),
  {
  A1:vform <=> fin,

  A1:in        <=> A0:in,
  A1:in:locDRS <=> [[],[]],

  A1:restr <=> A1restr,

  A1:out:gl_DRS <=> [GlDom,_GlConstr],

  A1:out:locDRS <=> [_Dom,Constr],

  A0:out:gl_DRS <=> A1:out:gl_DRS,   % assume that questions do
  A0:out:drs    <=> A0:in:drs,       % not contain information
  A0:out:focus  <=> A1:out:focus,    % eccept names of proper
  A0:restr      <=> A0restr,         % nouns
```

```
  A0:question <=> [yes,Constr],

  xappend(A1restr,[bind(GlDom)],A0restr)}.

que(A0,que(interr(what),Q)) --> what(_T), que(A1,gap(sub_sent(X)),nogap,Q),
  {
  A1:in <=> A0:in,

  A1:out:locDRS <=> [_Dom,LocConstr],
  A1:restr      <=> A1restr,

  A1:out:gl_DRS <=> [GlDom,_GlConstr],

  A0:question   <=> [X,LocConstr],

  A0:out:gl_DRS <=> A1:out:gl_DRS,
  A0:out:drs    <=> A0:in:drs,                 % !!
  A0:out:focus  <=> A1:out:focus,
  A0:restr      <=> A0restr,

  xappend(A1restr,[bind(GlDom)],A0restr)}.

que(A0,que(interr(T),Q)) --> who_or_what(T),
                               que(A1,gap(np(NPGap)),nogap,Q),
  {
  NPGap <=> sing3,

  NPGap:binding <=> X,

% this might seem a little bit strange
% but it allows
% Who likes himself?

  NPGap:in:gl_DRS <=> NPGap:out:gl_DRS,
  NPGap:in:locDRS <=> NPGap:out:locDRS,
  NPGap:in:drs    <=> [Dom,Constr],
  NPGap:out:drs   <=> [[X|Dom],[plural(X),male(X),female(X)|Constr]],
  NPGap:in:focus  <=> NPGap:out:focus,
  NPGap:locListIn <=> LocList,
  NPGap:locListOut<=> [X|LocList],

  A1:in <=> A0:in,
```

```
  A1:out:locDRS <=> [_LocDom,LocConstr],
  A1:restr    <=> A1restr,


  A1:out:gl_DRS <=> [GlDom,_GlConstr],

  A0:question   <=> [X,LocConstr],

  A0:out:gl_DRS <=> A1:out:gl_DRS,
  A0:out:drs    <=> A0:in:drs,            % !!
  A0:out:focus  <=> A1:out:focus,
  A0:restr      <=> A0restr,

  xappend(A1restr,[bind(GlDom)],A0restr)}.

que(A0,que(interr(who),Q)) --> who(_),  s(A1,gap(np(NPGap)),nogap,Q),
  {
  NPGap:cas  <=> nom,
  NPGap <=> sing3,
  A1:vform <=> fin,

  NPGap:binding <=> X,

% this might seem a little bit strange
% but it allows the answer Mary to the question
% Who likes himself?

  NPGap:in:gl_DRS <=> NPGap:out:gl_DRS,
  NPGap:in:locDRS <=> NPGap:out:locDRS,
  NPGap:in:drs    <=> [Dom,Constr],
  NPGap:out:drs   <=> [[X|Dom],[plural(X),male(X),female(X)|Constr]],
  NPGap:in:focus  <=> NPGap:out:focus,
  NPGap:locListIn <=> LocList,
  NPGap:locListOut<=> [X|LocList],

  A1:in:gl_DRS  <=> A0:in:gl_DRS,
  A1:in:drs     <=> A0:in:drs,
  A1:in:focus   <=> A0:in:focus,
  A1:in:locDRS  <=> [[],[]],

  A1:out:locDRS <=> [_Dom,LocConstr],
  A1:restr      <=> A1restr,
```

```
  A1:out:gl_DRS <=> [GlDom,_GlConstr],

  A0:question   <=> [X,LocConstr],
  A0:out:gl_DRS <=> A1:out:gl_DRS,
  A0:out:drs    <=> A0:in:drs,             % !!
  A0:out:focus  <=> A1:out:focus,
  A0:restr      <=> A0restr,

  xappend(A1restr,[bind(GlDom)],A0restr)
  }.

que(A0,que(interr(which),N,Q)) --> which(_T), n(Gap,N),
                                  que(A1,gap(np(NPGap)),nogap,Q),
  {
  Gap:binding  <=> X^NCond,
  NPGap:binding  <=> X,

  NPGap:in <=> NPGap:out,
  NPGap:in:focus  <=> NPGap:out:focus,
  NPGap:locListIn <=> LocList,
  NPGap:locListOut<=> [X|LocList],

  A1:in <=> A0:in,

  A1:out:locDRS <=> [_Dom,Constr],
  A1:restr    <=> A1restr,

  A1:out:gl_DRS <=> [GlDom,_GlConstr],

  A0:question <=> [X,[some(X,[NCond])|Constr]],

  A0:out:gl_DRS <=> A1:out:gl_DRS,
  A0:out:drs    <=> A0:in:drs,
  A0:out:focus  <=> A1:out:focus,
  A0:restr      <=> A0restr,

  xappend(A1restr,[bind(GlDom)],A0restr) }.


que(A0,que(pp(P,interr(which),N),Q)) --> p(A1,Pform,P), which(_T),
   n(A2,N),
                              que(_A3,gap(pp(Pform)),nogap,Q),
```

```
  {
  A1:article <=> indefinite,
  A2:article <=> indefinite,

  A0:restr <=> [write('No semantics!')] }.


que(A0,G0,G,que(Do,NP,VP)) --> vDo(A1,Do),
                          np(A2,G0,G1,NP),
                           vp(A3,G1,G,VP),
  {
  A1:vform <=> fin,
  A1:agr   <=> A2:agr,
  A3:vform <=> bse,

  A2:in:gl_DRS    <=> A0:in:gl_DRS,
  A2:in:drs       <=> A0:in:drs,
  A2:in:focus     <=> A0:in:focus,
  A2:in:locDRS    <=> [[],[]],
  A2:locListIn    <=> [],
  A2:in:scopeList <=> [],
  A2:restr        <=> RestrNp,
  A2:binding      <=> X,

  A3:in:gl_DRS    <=> A2:out:gl_DRS,
  A3:in:drs       <=> A2:out:drs,
  A3:in:locDRS    <=> A2:out:locDRS,
  A3:in:focus     <=> A2:out:focus,
  A3:locListIn    <=> A2:locListOut,
  A3:in:scopeList <=> [],                   % !!
  A3:binding      <=> X^_Cond,
  A3:restr        <=> RestrVp,

  A0:out:gl_DRS   <=> A3:out:gl_DRS,
  A0:out:focus    <=> A3:out:focus ,
  A0:restr        <=> RestrS,

  A2:out:locDRS    <=> [_A2Dom,A2LocConstr],
  A2:out:scopeList <=> A2ScopeList,
  A3:out:locDRS    <=> [_A3Dom,A3LocConstr],
  A3:out:scopeList <=> A3ScopeList,
  A0:out:locDRS    <=> [_A0Dom,A0LocConstr],
```

```
  scope_all(A2ScopeList,A2LocConstr,A2LocConstrOut),
  del_list(A2LocConstr,A3LocConstr,ClearedLocConstr),

  scope_all(A3ScopeList,ClearedLocConstr,A3LocConstrOut),

  xappend(A2LocConstrOut,A3LocConstrOut,A0LocConstr),

  xappend(RestrNp,RestrVp,RestrS)}.


que(_A0,G0,G,que(Do,NP,NEG,VP)) --> vDo(A1,Do),
                                 np(A2,G0,G1,NP),
                                 negation(NEG),
                                 vp(A3,G1,G,VP),
  {
  A1:vform <=> fin,
  A3:vform <=> bse,
  A1:agr   <=> A2:agr

  % semantic to follow

  }.

que(A0,G0,G,que(HAVE,NP,VP)) --> vhave(A1,HAVE),
                                np(A2,G0,G1,NP),
                                 vp(A3,G1,G,VP),
  {
  A1:vform <=> fin,
  A1:agr   <=> A2:agr,
  A3:vform <=> pas,

  A2:in:gl_DRS    <=> A0:in:gl_DRS,
  A2:in:drs       <=> A0:in:drs,
  A2:in:focus     <=> A0:in:focus,
  A2:in:locDRS    <=> [[],[]],
  A2:locListIn    <=> [],
  A2:in:scopeList <=> [],
  A2:restr        <=> RestrNp,
  A2:binding      <=> X,

  A3:in:gl_DRS    <=> A2:out:gl_DRS,
  A3:in:drs       <=> A2:out:drs,
  A3:in:locDRS    <=> A2:out:locDRS,
```

```
  A3:in:focus     <=> A2:out:focus,
  A3:locListIn    <=> A2:locListOut,
  A3:in:scopeList <=> [],                    % !!
  A3:binding      <=> X^_Cond,
  A3:restr        <=> RestrVp,

  A0:out:gl_DRS   <=> A3:out:gl_DRS,
  A0:out:focus    <=> A3:out:focus,
  A0:restr        <=> RestrS,

  A2:out:locDRS   <=> [_A2Dom,A2LocConstr],
  A2:out:scopeList <=> A2ScopeList,
  A3:out:locDRS   <=> [_A3Dom,A3LocConstr],
  A3:out:scopeList <=> A3ScopeList,
  A0:out:locDRS   <=> [_A0Dom,A0LocConstr],

  scope_all(A2ScopeList,A2LocConstr,A2LocConstrOut),
  del_list(A2LocConstr,A3LocConstr,ClearedLocConstr),

  scope_all(A3ScopeList,ClearedLocConstr,A3LocConstrOut),

  xappend(A2LocConstrOut,A3LocConstrOut,A0LocConstr),

  xappend(RestrNp,RestrVp,RestrS) }.


que(A0,G0,G,que(BE,NP,PCS)) --> vbe(A1,BE),
                               np(A2,G0,G1,NP),
                               pcs(A3,G1,G,PCS),
  {
  A1:vform <=> fin,
  A1:agr   <=> A2:agr,

  A2:in:gl_DRS     <=> A0:in:gl_DRS,
  A2:in:drs        <=> A0:in:drs,
  A2:in:focus      <=> A0:in:focus,
  A2:in:locDRS     <=> [[],[]],
  A2:in:scopeList  <=> [],
  A2:locListIn     <=> [],
  A2:restr         <=> RestrNp,
  A2:binding       <=> X,

  A3:in:gl_DRS    <=> A2:out:gl_DRS,
```

```
  A3:in:drs       <=> A2:out:drs,
  A3:in:locDRS    <=> A2:out:locDRS,
  A3:in:focus     <=> A2:out:focus,
  A3:locListIn    <=> A2:locListOut,
  A3:in:scopeList <=> [],                    % !!
  A3:binding      <=> X, %^_Cond,
  A3:restr        <=> RestrVp,

  A0:out:gl_DRS    <=> A3:out:gl_DRS,
  A0:out:focus     <=> A3:out:focus,
  A0:restr         <=> RestrS,

  A2:out:locDRS    <=> [_A2Dom,A2LocConstr],
  A2:out:scopeList <=> A2ScopeList,
  A3:out:locDRS    <=> [_A3Dom,A3LocConstr],
  A3:out:scopeList <=> A3ScopeList,
  A0:out:locDRS    <=> [_A0Dom,A0LocConstr],

  scope_all(A2ScopeList,A2LocConstr,A2LocConstrOut),
  del_list(A2LocConstr,A3LocConstr,ClearedLocConstr),

  scope_all(A3ScopeList,ClearedLocConstr,A3LocConstrOut),

  xappend(A2LocConstrOut, %[is(X,X)|A3LocConstrOut],A0LocConstr),
            A3LocConstrOut,A0LocConstr),

  xappend(RestrNp,RestrVp,RestrS) }.


%---------------------------------------------------------
% adjectives
%---------------------------------------------------------

adjs(A0,adj(DEG,ADJ)) --> deg(A1,DEG), adjs1(A0,ADJ),
  {
  A0:det <=> A1:det}.

adjs(A,ADJ)   --> adjs1(A,ADJ).
adjs1(A0,ADJ) --> adj(A1,ADJ1),moreadj(A2,ADJ1,ADJ),
  {
  A0:det <=> A1:det,

  A1:in  <=> A0:in,
```

```
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:binding <=> X^Cond,
  A2:binding <=> X,
  A0:binding <=> X,

  A2:adjList <=> A2adjList,
  A0:adjList <=> [Cond|A2adjList]}.

moreadj(A,ADJ,ADJ) --> [],
  {
  A:in <=> A:out,
  A:adjList <=> []}.

moreadj(A0,ADJ,adjs(ADJ,ADJ2)) --> adj(A1,ADJ1),
                                   moreadj(A2,ADJ1,ADJ2),
  {
  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:binding <=> X^Cond,
  A2:binding <=> X,
  A0:binding <=> X,

  A2:adjList <=> A2adjList,
  A0:adjList <=> [Cond|A2adjList]}.

%---------------------------------------------------
% adjective phrases
%---------------------------------------------------

adjp(A,G0,G,adjp(DEG,ADJP)) --> deg(_,DEG), adjp1(A,G0,G,ADJP).
adjp(A,G0,G,ADJP) --> adjp1(A,G0,G,ADJP).

adjp1(A0,G0,G,adjp(ADJ,PP)) --> adj(A1,pp,ADJ), pp(A2,Pform,G0,G,PP),
  {
  A1:pform <=> Pform,

  A2:article <=> indefinite,    % to avoid checks for pp

% A1:in:drs     <=> [[],[]],    % no threading
```

```
% A1:in:locDRS  <=> [[],[]],    % the entry is the instanciated
  A1:binding    <=> X^ACond,    % preposition

  A2:in  <=> A0:in,
  A0:out <=> A2:out,

  A2:locListIn  <=> A0:locListIn,

  A2:binding    <=> ACond^_PCond,    % here the entry is made
  A0:binding    <=> X, %^PCond,
  A0:restr      <=> A2:restr }.

adjp1(A0,G,G,adjp(ADJ,sub_sent(subord(that),RS))) -->
                                     adj(A1,that,ADJ),
                                               that(_T),
                                 s(A2,nogap,nogap,RS),
  {
  A2:vform <=> fin,

  A1:in <=> A0:in,
  A2:in <=> A1:out,

error

  }.

adjp1(G0,G,adjp(ADJ,VP))   --> adj(inf,ADJ), vp(A,G0,G,VP),
  { A:vform <=> inf}.

adjp1(A0,G,G,adjp(ADJ))    --> adj(A1,ADJ),
  {
  A1:in  <=> A0:in,
  A0:out <=> A1:out,

  A0:binding <=> X,
  A1:binding <=> X^_}.


%---------------------------------------------------------
% noun phrases
%---------------------------------------------------------

np(NP,gap(np(NP)),nogap,np('_')) --> [].
```

```
np(A,G,G,NP) --> np3(A,NP).

np3(A0,NP) --> np2(A1,NP1), more_np(A2,A1,Amo,NP1,NP),
  {
  A0:cas <=> A1:cas,
  A0:cas <=> A2:cas,
  A0:agr <=> Amo:agr,

  A1:in  <=> A0:in,
  A0:out <=> A1:out,

  A1:locListIn  <=> A0:locListIn,
  A0:locListOut <=> A1:locListOut,

  A0:restr     <=> A1:restr,
  A0:binding   <=> A1:binding }.

more_np(A,Ami,Amo,NP,NP) --> [],
  {
  Ami:agr <=> Amo:agr,

  A:in    <=> A:out}.

more_np(A1,_,Amo,NP,NP2) --> coord(and,C),
                              np3(A2,NP1),
                more_np(A3,Ami,Amo,coordination(NP,C,NP1),NP2),
  {
  A1:cas <=> A2:cas,
  A1:cas <=> A3:cas,
  Ami:agr:num <=> plur}.

more_np(A1,_,Amo,NP,NP2) --> coord(or,C),
                             np3(A2,NP1),
                more_np(A3,A2,Amo,coordination(NP,C,NP1),NP2),
  {
  A1:cas <=> A2:cas,
  A1:cas <=> A3:cas}.

np2(A,np(PN)) --> pronoun(A,PN).

np2(A,np(PN))    --> propernoun(A,PN).
np2(A0,np(PN,PM)) --> propernoun(A1,PN), non_restr_rel_cl(A2,PM),
```

```
  {
  A0:agr   <=> A1:agr,
  A1:agr   <=> A2:agr,
  A1:human <=> A2:human,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:locListIn  <=> A0:locListIn,
  A2:locListIn  <=> A1:locListOut,
  A0:locListOut <=> A1:locListOut,       % !!

  A0:binding <=> A1:binding,
  A1:binding <=> X,
  A2:binding <=> X^_,

  A0:restr <=> A0restr,
  A1:restr <=> A1restr,
  A2:restr <=> A2restr,

  xappend(A1restr,A2restr,A0restr) }.


% for Questions
np2(A0,np(interr(which),N)) --> which(_T),n(_A1,N),
  {
  A0:restr <=> [write('No Semantic!'),nl]}.

np2(A0,PM) --> det(A1,DETP), np1(A2,NP),
              post_modifier_np(A3,np(DETP,NP),PM),
  {
   A1:det   <=> A2:det,
   A1:agr   <=> A2:agr,
   A0:agr   <=> A2:agr,
   A0:cas   <=> A2:cas,
   A0:human <=> A2:human,
   A0:human <=> A3:human,

   A1:article <=> A2:article,
   A1:article <=> A3:article,

   A1:in:quantified <=> A0:in:quantified,
```

```
   A1:in:scopeList  <=> A0:in:scopeList,

   A2:in:gl_DRS    <=> A0:in:gl_DRS,
   A2:in:drs       <=> A0:in:drs,
   A2:in:locDRS    <=> A0:in:locDRS,
   A2:in:scopeList <=> A0:in:scopeList,
   A2:in:focus     <=> A0:in:focus,
   A2:in:npConstr  <=> [],                % !!

   A3:in  <=> A2:out,

   A3:check      <=> Description,
   A3:locListIn  <=> [X],
   A0:locListIn  <=> LocListIn,
   A0:locListOut <=> [X|LocListIn],

   A3:out:npConstr <=> Scope,

   % some(X,[man(X),fat(X)]) & man(X)

   A1:binding    <=> X^Scope^Description,
   A2:binding    <=> X^Description,
   A3:binding    <=> X^Description,
   A0:binding    <=> X,


   A0:out:gl_DRS    <=> A3:out:gl_DRS,
   A0:out:drs       <=> A3:out:drs,
   A0:out:locDRS    <=> A3:out:locDRS,
   A0:out:scopeList <=> A1:out:scopeList,      % !!
   A0:out:focus     <=> A3:out:focus,
   A0:out:quantified<=> A1:out:quantified,

   A0:restr  <=> A0restr,
   A2:restr  <=> A2restr,
   A3:restr  <=> A3restr,
   xappend(A2restr,A3restr,A0restr)
   }.

np2(A0,PM) --> np1(A1,NP), post_modifier_np(A2,NP,PM),
  {
  A1:agr:num <=> plur,
  A0:agr <=> A1:agr,
```

```
  A0:cas <=> A1:cas,

  A2:human <=> A1:human, % agreement with PM left

  A2:article <=> indefinite, % no article

  A1:in    <=> A0:in,
  A2:in    <=> A1:out,

  A1:binding   <=> X^Cond,
  A2:binding   <=> X,
  A0:binding   <=> X,

  A2:locListIn  <=> [X],
  A0:locListIn  <=> LocListIn,
  A0:locListOut <=> [X|LocListIn],

  A0:out:gl_DRS  <=> A2:out:gl_DRS,
  A0:out:drs     <=> A2:out:drs,
  A0:out:locDRS  <=> A2:out:locDRS,
  A0:out:scopeList <=> [every(X,Cond)],  % !!!!!!
  A0:out:focus   <=> A2:out:focus,

  A1:restr <=> A1restr,
  A2:restr <=> A2restr,
  A0:restr <=> A0restr,
  xappend(A1restr,A2restr,A0restr)}.

np1(A,N) --> n(A,N).
np1(A0,np(ADJ,N)) --> adjs(A1,ADJ), n(A2,N),
  {
  A0:det <=> A1:det,
  A0:agr <=> A2:agr,
  A0:cas <=> A2:cas,

  A2:article <=> A0:article,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,

  A1:binding <=> X,
  A2:binding <=> X^Cond,
  A0:binding <=> X^Cond,
```

```
  (
    A0:article <=> indefinite,
    A0:restr   <=> A2:restr,
    A0:out:gl_DRS <=> A2:out:gl_DRS,
    A2:out:drs    <=> [A2Dom,A2ConstrOut],
    A0:out:drs    <=> [A2Dom,A2ConstrOut],  % is(X,X)
    A2:out:locDRS <=> [A2LocDom,A2LocConstrOut],
    A0:out:locDRS <=> [A2LocDom,A2LocConstrOut],  % is(X,X)
    A0:out:focus  <=> A2:out:focus,
    A0:out:npConstr <=> A2:out:npConstr
  ;
    A0:article <=> definite,
    A1:adjList <=> AdjList,
    A0:out:gl_DRS <=> A0:in:gl_DRS,
    A0:out:locDRS <=> A0:in:locDRS,
    A0:out:focus  <=> A2:out:focus,         % !!
    A0:out:npConstr <=> A2:out:npConstr,

    A0:in:drs    <=> [_A0DomIN,A0ConstrIn],
    A0:out:drs <=> A0:in:drs,

    % check whether the object the dicourse refers to
    % has all mentioned properties
    A2:restr <=> A2restr,
    A0:restr <=> [members(AdjList,A0ConstrIn)|A2restr]
  )
  }.

np1(A0,np(ADJ,N)) --> adj(A1,mod,ADJ), n(A2,N),
  {
  A0:det <=> A1:det,
  A0:agr <=> A2:agr,
  A0:cas <=> A2:cas,

  A2:article <=> A0:article,

  A1:in:drs     <=> [[],[]],

  A2:in      <=> A0:in,

  A1:binding <=> Cond^ACond,
  A2:binding <=> X^Cond,
```

```
  A0:binding <=> X^Cond,

  A0:out:gl_DRS <=> A0:in:gl_DRS,
  A0:out:focus  <=> A2:out:focus,

  A0:in:drs    <=> [_A0DomIn,   A0ConstrIn],
  A0:in:locDRS <=> [_A0LocDomIn,A0LocConstrIn],
  (
    A0:article <=> indefinite,
    A0:restr   <=> A2:restr,

    A1:out:drs    <=> [Dom,   A1ConstrOut],
    A1:out:locDRS <=> [LocDom,A1LocConstrOut],
    A0:out:drs    <=> [Dom,   A0ConstrOut],
    A0:out:locDRS <=> [LocDom,A0LocConstrOut],

    xappend(A1ConstrOut,   A0ConstrIn,   A0ConstrOut),
    xappend(A1LocConstrOut,A0LocConstrIn,A0LocConstrOut)


  ;
    A0:article <=> definite,
    A0:out:drs  <=> A0:in:drs,

    % check whether the object the dicourse refers to
    % has all mentioned properties
    A2:restr <=> A2restr,
    A0:restr <=> [member(ACond,A0ConstrIn)|A2restr]
  )
  }.

post_modifier_np(A0,PM1,PM) --> pp(A1,_PName,nogap,nogap,PP),
                          post_modifier_np(A2,np(PM1,PP),PM),

  {
  A0:human <=> A2:human,
  A0:agr   <=> A2:agr,

  A1:article <=> A0:article,
  A2:article <=> A0:article,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
```

```
  A0:out <=> A2:out,

  A1:locListIn <=> A0:locListIn,
  A2:locListIn <=> A0:locListIn,

  A0:binding <=> A2:binding,

  A1:check <=> A0:check,
  A2:check <=> A0:check,

  A1:restr <=> A1restr,
  A2:restr <=> A2restr,
  A0:restr <=> A0restr,
  xappend(A1restr,A2restr,A0restr) }.

% for semantical reasons this rule has to
% be placed below the pm --> pp,pm rule
% see Documentation / Preposition Binding

post_modifier_np(A,PM,PM) --> [],
  {
  A:in <=> A:out }.

post_modifier_np(A0,PM1,PM) --> restr_rel_cl(A1,RS),
                 post_modifier_np(A2,np(PM1,RS),PM),
  {
  A0:human <=> A1:human,
  A0:human <=> A2:human,
  A0:agr   <=> A1:agr,
  A0:agr   <=> A2:agr,

  A1:article <=> A0:article,
  A2:article <=> A0:article,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:locListIn <=> A0:locListIn,
  A2:locListIn <=> A0:locListIn,

  A0:binding <=> A1:binding,
  A0:binding <=> A2:binding,
```

```
  A0:restr <=> A0restr,
  A1:restr <=> A1restr,
  A2:restr <=> A2restr,
  xappend(A1restr,A2restr,A0restr) }.

post_modifier_np(A0,PM1,PM) --> non_restr_rel_cl(A1,RS),
                     post_modifier_np(A2,np(PM1,RS),PM),
  {
  A0:human <=> A1:human,
  A0:human <=> A2:human,
  A0:agr   <=> A1:agr,
  A0:agr   <=> A2:agr,

  A2:article <=> A0:article,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:locListIn <=> A0:locListIn,
  A2:locListIn <=> A0:locListIn,

  A0:binding <=> A1:binding,
  A0:binding <=> A2:binding,

  A0:restr <=> A0restr,
  A1:restr <=> A1restr,
  A2:restr <=> A2restr,
  xappend(A1restr,A2restr,A0restr) }.

non_restr_rel_cl(A0,non_restr(comma(','),RP,S,comma(','))) -->
                     comma(_), relpron(A1,RP),
                     s(A2,gap(np(NPGap)),nogap,S),comma_or_not(_),
  {
  A1:restr <=> no,
  A1:human <=> A0:human,

  A2:vform <=> fin,
  A0:agr <=> NPGap:agr,
  % agr must be the same, but case might be different

  A2:in  <=> A0:in,
```

```
  A0:out <=> A2:out,

  A0:restr <=> A2:restr,

  A0:binding <=> X^_Description,

  A2:locListIn <=> A0:locListIn,

  NPGap:binding   <=> X,
  NPGap:in  <=> NPGap:out,
  NPGap:locListIn <=> LocList,
  NPGap:locListOut <=> [X|LocList] }.

restr_rel_cl(A0,restr(S)) -->
                       rs(A0,gap(np(NPGap)),nogap,S),
{
  NPGap:agr <=> A0:agr,
  NPGap:cas <=> acc}.

restr_rel_cl(A0,restr(RP,S)) --> relpron(A1,RP),
                        rs(A0,gap(np(NPGap)),nogap,S),
{
  A0:human <=> A1:human,
  A0:agr   <=> NPGap:agr}.
  % agr must be the same, but case might be different

rs(A0,gap(np(NPGap)),nogap,S) -->
                        s(A2,gap(np(NPGap)),nogap,S),
  {
  A2:vform <=> fin,

  A2:in:gl_DRS  <=> A0:in:gl_DRS,
  A2:in:drs     <=> A0:in:drs,
  A2:in:locDRS  <=> [[],[]],
  A2:in:focus   <=> A0:in:focus,

  A2:locListIn <=> A0:locListIn,

  A0:out:gl_DRS <=> A2:out:gl_DRS,
  A0:out:drs    <=> A2:out:drs,
  A0:out:focus  <=> A2:out:focus,

  A0:in:npConstr   <=> NPConstrIn,
```

```
  A0:out:npConstr  <=> NPConstrOut,
  A0:restr         <=> A0restr,

  A0:binding    <=> X^_Description,

  NPGap:binding    <=> X,
  NPGap:in         <=> NPGap:out,
  NPGap:locListIn  <=> LocList,
  NPGap:locListOut <=> [X|LocList],

  A2:out:locDRS <=> [_A2Dom,LocConstrS],
  A0:in:locDRS  <=> [Dom,LocConstrIn],
  A0:out:locDRS <=> [Dom,LocConstrOut],

  (A0:article   <=> indefinite,
   A2:restr     <=> T
  ;
   A0:article    <=> definite,
   A0:in:drs     <=> [_A0Dom,Constr],

   A2:restr     <=> A2restr,

   % all properties mentioned in arestrictive relative clause
   % have to be checked
   xappend(A2restr,[ members(LocConstrS,Constr)
                  ],T)
  ),

  xappend(T,[
xappend(LocConstrIn,LocConstrS,LocConstrOut),
xappend(NPConstrIn,LocConstrS,NPConstrOut)
                  ],A0restr)
 }.

%---------------------------------------------------------
% prepositional phrases
%---------------------------------------------------------

pp(A0,Pform,gap(pp(Pform)),nogap,pp('_')) --> [],
  {
  A0:in <=> A0:out}.

pp(A0,Pform,G0,G,pp(P,NP)) --> p(A1,Pform,P), np(A2,G0,G,NP),
```

```
  {
  A2:cas <=> acc,

  A1:article <=> A0:article,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:check   <=> A0:check,

  A1:binding <=> Y^X^Prep,
  A2:binding <=> Y,
  A0:binding <=> X^Prep,

  A2:locListIn  <=> A0:locListIn,

  A0:restr  <=> A0restr,
  A1:restr  <=> A1restr,
  A2:restr  <=> A2restr,
  xappend(A1restr,A2restr,A0restr)
  }.

%----------------------------------------------------------
% verb phrases
%----------------------------------------------------------

vp(A ,G0,G,VP) --> vp2(A,G0,G,VP).
vp(A0,G ,G,VP) --> vp2(A1,nogap,nogap,VP1),
                coord(_,C),
                vp2(A2,nogap,nogap,VP2),
                more_vp(A3,coordination(VP1,C,VP2),VP),
  {
  A0:vform <=> A1:vform,
  A0:vform <=> A2:vform,
  A0:vform <=> A3:vform,
  A0:agr   <=> A1:agr,
  A0:agr   <=> A2:agr,
  A0:agr   <=> A3:agr

  % semantic to follow

  }.
```

```
more_vp(A,VP,VP) --> [],
  {
  A:out <=> A:in }.

more_vp(A0,VP1,VP) --> coord(_,C),
                   vp2(A1,nogap,nogap,VP2),
                   more_vp(A2,coordination(VP1,C,VP2),VP),
  {
  A0:vform <=> A1:vform,
  A0:vform <=> A2:vform,

  A0:agr <=> A1:agr,
  A0:agr <=> A2:agr

  % semantic to follow

  }.

vp2(A0,G0,G,PM) --> vp1(A1,G0,G1,VP),
                    post_modifier_vp(A2,G1,G,VP,PM),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:locListIn <=> A0:locListIn,
  A2:locListIn <=> A1:locListOut,
% A0:locListOut<=> A0:locListIn,

  A0:binding   <=> A1:binding,
  A1:binding   <=> _X^Verb,
  A2:binding   <=> Verb,         % prepositions found
 % as VP modifiers getting
 % bound to the VP

  A0:restr     <=> A0restr,
  A1:restr     <=> A1restr,
  A2:restr     <=> A2restr,
  xappend(A1restr,A2restr,A0restr)}.
```

```
post_modifier_vp(A,G,G,P,P) --> [],
  {
  A:out <=> A:in,
  A:restr <=> []}.

post_modifier_vp(A,G,G,P,mod_vp(P,ADV)) --> adv(ADV),
  {
  A:out <=> A:in,
  A:restr <=> []}.


post_modifier_vp(A0,G0,G,PM1,PM) --> pp(A1,_PName,G0,G1,PP),
                post_modifier_vp(A2,G1,G,mod_vp(PM1,PP),PM),
  {
  A1:article <=> indefinite,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:locListIn <=> A0:locListIn,
  A2:locListIn <=> A0:locListIn,
% A0:locListOut <=> A0:locListIn,

  A0:binding <=> Verb,
  A1:binding <=> Verb^_Preposition,

  A0:restr <=> A0restr,
  A1:restr <=> A1restr,
  A2:restr <=> A2restr,
  xappend(A1restr,A2restr,A0restr) }.

%-------------------------------------------------------
% passive constructions

verbtrans(A,VP) --> verb(A,trans,VP),
   {
   A:vform <=> pas}.

verbtrans(A,VP) --> verb(A,ditrans,VP),
   {
```

```
   A:vform <=> pas}.

vp1(A0,G,G,vp(BE,VP)) --> vbe(A1,BE),  verbtrans(A2,VP),
   {
   A0:vform <=> A1:vform,
   A0:agr   <=> A1:agr,

   A2:in    <=> A0:in,
   A0:out   <=> A2:out,

   A0:locListOut <=> A0:locListIn,

   A0:restr    <=> A2:restr,

   (A2:binding <=> Y^_^_^Cond    % ditransitive
   ;A2:binding <=> Y^_X^Cond     % transitive
   ),
   A0:binding  <=> Y^Cond }.

vp1(A1,G,G,vp(BE,NEG,VP)) --> vbe(A1,BE), negation(NEG),
                              verbtrans(_A2,VP).

%-------------------------------------------------------
vp1(A0,G0,G,vp(HAVE,VP)) --> vhave(A1,HAVE), vp1(A2,G0,G,VP),
   {
   A0:vform <=> A1:vform,
   A0:agr   <=> A1:agr,
   A2:vform <=> psp,

   A2:in  <=> A0:in,
   A0:out <=> A2:out,

   A2:locListIn <=> A0:locListIn,
   A0:locListOut <=> A2:locListOut,

   A0:binding   <=> A2:binding,
   A0:restr     <=> A2:restr }.

vp1(A1,G0,G,vp(HAVE,NEG,VP)) --> vhave(A1,HAVE), negation(NEG),
                                 vp1(A2,G0,G,VP),
   {
   A2:vform <=> psp}.
```

```
%----------------------------------------------------
vp1(A0,G0,G,vp(Do,VP)) --> vDo(A1,Do), vp1(A2,G0,G,VP),
   {
   A0:vform <=> A1:vform,
   A0:agr   <=> A1:agr,
   A2:vform <=> bse,

   A2:in  <=> A0:in,
   A0:out <=> A2:out,

   A2:locListIn <=> A0:locListIn,
   A0:locListOut <=> A2:locListOut,

   A0:binding   <=> A2:binding,
   A0:restr     <=> A2:restr }.

vp1(A0,G0,G,vp(AUX,NEG,VP)) --> vDo(A1,AUX),negation(NEG),
                                vp1(A2,G0,G,VP),
   {
   A0:vform <=> A1:vform,
   A0:agr   <=> A1:agr,
   A2:vform <=> bse,

   A2:in:gl_DRS    <=> A0:in:gl_DRS,
   A2:in:drs       <=> A0:in:drs,
   A2:in:locDRS    <=> [[],[]],
   A2:in:focus     <=> A0:in:focus,
   A2:in:scopeList <=> A0:in:scopeList,


   A0:out:gl_DRS    <=> A2:out:gl_DRS,

   A2:out:locDRS    <=> LocDRS,

   A0:in :drs       <=> [A0Dom,A0Constr],
   A0:out:drs       <=> [A0Dom,[not LocDRS|A0Constr]],

   A0:in :locDRS    <=> [A0LocDomIn,A0LocConstrIn],
   A0:out:locDRS    <=> [A0LocDomIn,[not LocDRS|A0LocConstrIn]],

   A0:out:focus     <=> A2:out:focus,
   A0:out:scopeList <=> A2:out:scopeList,
```

```
   A2:locListIn <=> A0:locListIn,
   A0:locListOut <=> A2:locListOut,

   A0:binding   <=> A2:binding,
   A0:restr     <=> A2:restr }.


%---------------------------------------------------
% intransitive verbs

vp1(A,G,G,V) --> verb(A,intr,V).

%---------------------------------------------------
% copula verb phrases

vp1(A0,G0,G,vp(VBE,PCs)) --> vbe(A1,VBE), pcs(A2,G0,G,PCs),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A0:agr   <=> A2:agr,

  A2:in  <=> A0:in,

  A0:out <=> A2:out,

  A2:binding <=> X,
  A0:binding <=> X^_dummy,
  /*
  A0:out:gl_DRS <=> A2:out:gl_DRS,
  A0:out:drs    <=> A2:out:drs, %[A2Dom,[is(X,X)|A2Constr]],
  A2:out:drs    <=> [A2Dom,A2Constr],
  A0:out:locDRS <=> [A2LocDom,[is(X,X)|A2LocConstr]],
  A2:out:locDRS <=> [A2LocDom,A2LocConstr],
  A0:out:focus  <=> A2:out:focus,
  A0:out:scopeList <=> A2:out:scopeList,
  */

  A2:locListIn <=> A0:locListIn,
  A0:locListOut<=> A0:locListIn,

  A0:restr    <=> A2:restr}.

%----------------------------------------------------
```

```
% subjective predicative

pcs(A,G0,G,ADJP) --> adjp(A,G0,G,ADJP).
pcs(_,G0,G,adjp(NEG,ADJP)) --> negation(NEG), adjp(2,G0,G,ADJP).

pcs(A0,G0,G,NP) --> np(A1,G0,G,NP),
  {
  A0:agr:num <=> A1:agr:num,  % in general

  A1:in  <=> A0:in,

  A0:binding <=> A1:binding,
  A0:out:gl_DRS <=> A1:out:gl_DRS,
  A0:out:drs    <=> A1:out:drs,
  A0:out:locDRS <=> A1:out:locDRS,
  A0:out:focus  <=> A1:out:focus,
  A0:out:scopeList <=> [],      % general statement like
                                % Every man is a lunatic.
  A0:restr     <=> A1:restr }.

pcs(A,G0,G,adjp(NEG,NP)) --> negation(NEG), np(A1,G0,G,NP),
  { A:agr:num <=> A1:agr:num}. % in genaral

% problem case in NP "it was me."

%-------------------------------------------------------
% transitive verb phrases

vp1(A0,G0,G,vp(V,NP)) --> verb(A1,trans,V),
                          np(A2,G0,G,NP),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A2:cas   <=> acc,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  A1:binding <=> (Y^X^Cond),
  A2:binding <=> Y,
  A0:binding <=> (X^Cond),
```

```
  A2:locListIn <=> A0:locListIn,
  A0:locListOut<=> A2:locListOut,

  A0:restr     <=> A2:restr }.


%-------------------------------------------------------
% ditransitive verb phrases

vp1(A0,G0,G,vp(V,NP1,NP2)) -->
                 verb(A1,ditrans,V), np(A2,G0,G1,NP1),
                 np(A3,G1,G,NP2),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A2:cas   <=> acc,
  A3:cas   <=> acc,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A3:in  <=> A2:out,
  A0:out <=> A3:out,

  A1:binding   <=> Z^Y^X^Cond,
  A2:binding   <=> Y,
  A3:binding   <=> Z,
  A0:binding   <=> X^Cond,

  A2:locListIn <=> A0:locListIn,
  A3:locListIn <=> A2:locListOut,
  A0:locListOut<=> A3:locListOut,

  A0:restr     <=> A0restr,
  A2:restr     <=> A2restr,
  A3:restr     <=> A3restr,
  xappend(A2restr,A3restr,A0restr) }.


vp1(A0,G0,G,vp(V,NP)) --> verb(A1,ditrans,V), np(A2,G0,G,NP),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A2:cas   <=> acc,
```

```
  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A0:out <=> A2:out,

  % Y left unbound to be bound eventually by the
  % preposition binding procedure

  A1:binding   <=> Z^_Y^X^Cond,
  A2:binding   <=> Z,
  A0:binding   <=> X^Cond,

  A2:locListIn <=> A0:locListIn,
  A0:locListOut<=> A2:locListOut,

  A0:restr      <=> A2:restr }.


%---------------------------------------------------------
% raising and control constructions

vp1(A0,G0,G,vp(V,VP)) --> verb(A1,inf,V), vp(A2,G0,G,VP),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A2:vform <=> inf,

  A1:in     <=> A0:in,
  A2:in     <=> A1:out,

  % X want to do Y

  A1:binding   <=> Do^X^Want,
  A2:binding   <=> X^Do,
  A0:binding   <=> X^Want,

  A2:locListIn <=> A0:locListIn,
  A2:restr     <=> A2restr,

  A0:out:gl_DRS <=> A2:out:gl_DRS,
  A0:out:drs    <=> [Dom,A0Constr],
  A0:locListOut <=> A0:locListIn,
```

```
  A0:out:focus  <=> A2:out:focus,
  A0:out:locDRS <=> [LocDom,A0LocConstr],
  A0:restr      <=> A0restr,

  A2:out:drs    <=> [Dom,A2Constr],
  A2:out:locDRS <=> [LocDom,A2LocConstr],

  xappend(A2restr,[
           del(Do,A2Constr,A0Constr),
           del(Do,A2LocConstr,A0LocConstr)
                 ],A0restr)
}.

vp1(A0,G0,G,vp(V,NP,VP)) --> verb(A1,inf2,V), np(A2,G0,G1,NP),
                             vp(A3,G1,G,VP),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A2:cas   <=> acc,
  A3:vform <=> inf,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A3:in  <=> A2:out,

  % X want Y to do Z

  A1:binding   <=> DoZ^Y^X^Want,
  A2:binding   <=> Y,
  A3:binding   <=> Y^DoZ,
  A0:binding   <=> X^Want,

  A2:locListIn <=> A0:locListIn,
  A3:locListIn <=> A2:locListOut,

  A0:out:gl_DRS <=> A3:out:gl_DRS,
  A0:out:drs    <=> [Dom,A0Constr],
  A0:locListOut <=> A0:locListIn,
  A0:out:locDRS <=> [LocDom,A0LocConstr],
  A0:out:focus  <=> A3:out:focus,

  A0:restr     <=> A0restr,
  A2:restr     <=> A2restr,
```

```
  A3:restr     <=> A3restr,

  A3:out:drs    <=> [Dom,A3Constr],
  A3:out:locDRS <=> [LocDom,A3LocConstr],

  xappend(A2restr,A3restr,T2),
  xappend(T2,[
            del(DoZ,A3Constr,A0Constr),
            del(DoZ,A3LocConstr,A0LocConstr)
             ],A0restr)
}.

%-----------------------------------------------------------
sub_sent(A0,gap(sub_sent(Cond)),nogap,sub_sent('_')) --> [],
  {
  A0:out     <=> A0:in,
  A0:binding <=> Cond,
  A0:gap     <=> yes }.

sub_sent(A0,G0,G,sub_sent(subord(that),S)) --> that(_T),
 s(A0,G0,G,S),
  {
  A0:vform <=> fin,
  A0:gap <=> no}.

sub_sent(A0,G0,G,sub_sent(subord(that),S)) --> s(A0,G0,G,S),
  {
  A0:vform <=> fin,
  A0:gap <=> no}.

vp1(A0,G0,G,vp(V,RS)) --> verb(A1,that,V),
                          sub_sent(A2,G0,G,RS),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,

  A1:in     <=> A0:in,
  A2:in     <=> A1:out,

  % X said that S

  A1:binding  <=> S^X^Cond,
  A2:binding  <=> S,
```

```
  A0:binding  <=> X^Cond,

  A2:locListIn <=> [],
  A2:restr     <=> A2restr,
  A2:out:drs   <=> [Dom,A2Constr],

  A0:out:gl_DRS <=> A2:out:gl_DRS,
  A0:out:drs    <=> [Dom,A0Constr],
  A0:locListOut <=> A0:locListIn,
  A0:out:locDRS <=> A2:out:locDRS,
  A0:out:focus  <=> A2:out:focus,
  A0:restr      <=> A0restr,

  (   A2:gap <=> yes,
      % nothing to delete / the gap is because
      % we are recognising a question
      A2restr = A0restr
  ;
      A2:gap <=> no,
      % delete doubled information
      xappend(A2restr,[del(S,A2Constr,A0Constr)],A0restr)
  )}.

vp1(A0,G0,G,vp(V,NP,sub_sent(subord(that),RS))) -->
                   verb(A1,that2,V),
                   np(A2,G0,G1,NP),
                   sub_sent(A3,G1,G,RS),
  {
  A0:vform <=> A1:vform,
  A0:agr   <=> A1:agr,
  A2:cas   <=> acc,

  A1:in  <=> A0:in,
  A2:in  <=> A1:out,
  A3:in  <=> A2:out,

  % X told Y that Sent.

  A1:binding <=> Sent^Y^X^Cond,
  A2:binding <=> Y,
  A3:binding <=> Sent,
  A0:binding <=> X^Cond,
```

```
  A2:locListIn <=> A0:locListIn,
  A3:locListIn <=> [],
  A3:out:drs   <=> [Dom,A3Constr],

  A0:out:gl_DRS <=> A3:out:gl_DRS,
  A0:out:drs    <=> [Dom,A0Constr],
  A0:out:locDRS <=> A3:out:locDRS,
  A0:locListOut <=> A0:locListIn,
  A0:out:focus  <=> A3:out:focus,

  A2:restr <=> A2restr,
  A3:restr <=> A3restr,
  A0:restr <=> A0restr,

  xappend(A2restr,A3restr,T2),
  xappend(T2,[del(Sent,A3Constr,A0Constr)],A0restr)}.

vp1(A0,G0,G,vinf(to,V))   --> to(_T), vp1(A1,G0,G,V),
  {
  A0:vform <=> inf,
  A1:vform <=> bse,

  A1:in  <=> A0:in,
  A0:out <=> A1:out,

  A1:locListIn  <=> A0:locListIn,
  A0:locListOut <=> A1:locListOut,

  A0:binding <=> A1:binding,
  A0:restr   <=> A1:restr

}.

verb(A,Cat,VP) --> v(A,Cat,VP).

verb(A0,Cat,vp_conj(V1,C,V2)) -->
            v(A1,Cat,V1),coord(and,C),v(A2,Cat,V2),
  {
  A0:vform <=> A1:vform,
  A0:vform <=> A2:vform,
  A0:agr   <=> A1:agr,
  A0:agr   <=> A2:agr,
```

```
A1:in   <=> A0:in,
A2:in   <=> A1:out,
A0:out  <=> A2:out,

A1:binding <=> B1,
A2:binding <=> B2,

join(B1,B2,B0),

A0:binding <=> B0 }.
```

A.2 The Lexicon

```
%------------------------------------------------------
%
% File: lexicon
%
% Stefan M(ue)ller
%
% Large Practical
%
%
% Version 11.03.93
%
%
%---------------------------------------------------

%---------------------------------------------------
% adjectives
%---------------------------------------------------

%--------- need a prepositional phrase -------------

adj(A,pp,adj(fond))   --> [fond],
  { A:pfrom <=> of,
    A <=> adjective(fond)}.

adj(A,pp,adj(afraid)) --> [afraid],
  { A:pform <=> of,
    A <=> adjective(afraid)}.

adj(A,pp,adj(afraid)) --> [proud],
  { A:pform <=> of,
    A <=> adjective(proud)}.

%--------- need a sentence --------------------------

adj(A,that,adj(sure))   --> [sure],
  { A <=> adjective(sure)}.

%--------- need a verb phrase -----------------------

adj(A,inf,adj(able))    --> [able],
  { A <=> adjective(able)}.
```

```
adj(A,inf,adj(eager))    --> [eager],
  { A <=> adjective(eager)}.

%--------- modal adjectives ------------------------

adj(A,mod,adj(possible))    --> [possible],
  {
  A:det <=> a,
  A <=> adjective(possible)}.

%--------- normal adjectives -----------------------

adj(A,adj(big)) --> [big],
  {
  A:det <=> a,
  A <=> adjective(big)}.

adj(A,adj(anxious)) --> [anxious],
  {
  A:det <=> a,
  A <=> adjective(anxious)}.

adj(A,adj(extraordinary)) --> [extraordinary],
  {
  A:det <=> an,
  A <=> adjective(extraordinary)}.

adj(A,adj(fat))        --> [fat],
  {
  A:det <=> a,
  A <=> adjective(fat)}.

adj(A,adj(good))      --> [good],
  {
  A:det <=> a,
  A <=> adjective(good)}.


adj(A,adj(huge))      --> [huge],
  {
  A:det <=> a,
  A <=> adjective(huge)}.
```

```
adj(A,adj(large))     --> [large],
  {
  A:det <=> a,
  A <=> adjective(large)}.

adj(A,adj(lazy))     --> [lazy],
  {
  A:det <=> a,
  A <=> adjective(lazy)}.

adj(A,adj(stupid))    --> [stupid],
  {
  A:det <=> a,
  A <=> adjective(stupid)}.

adj(A,adj(valuable))  --> [valuable],
  {
  A:det <=> a,
  A <=> adjective(valuable)}.


deg(A,deg(very))    --> [very],
  {A:det <=> a}.
deg(A,deg(so))            --> [so],
  {A:det <=> a}.

%-------------------------------------------------------
% adverbs
%-------------------------------------------------------

adv(adv(well)) --> [well].
adv(adv(often)) --> [often].
adv(adv(beautifully)) --> [beautifully].
adv(adv(quickly)) --> [quickly].

coord(and,coord(and)) --> [and].
coord(or ,coord(or))  --> [or].

%-------------------------------------------------------
% determiner
%-------------------------------------------------------
```

```
det(A,det(a)) --> [a],
  {
  A:agr:num  <=> sing,
  A:det      <=> a,

  A:article        <=> indefinite,
  A:in:scopeList   <=> SList,
  A:out:scopeList  <=> [some(X,Scope) & Description|SList],
  A:out:quantified <=> no,
  A:binding        <=> X^Scope^Description}.


det(A,det(an)) --> [an],
  {
  A:agr:num <=> sing,
  A:det <=> an,

  A:article        <=> indefinite,
  A:in:scopeList   <=> SList,
  A:out:scopeList  <=> [some(X,Scope) & Description|SList],
  A:out:quantified <=> no,
  A:binding        <=> X^Scope^Description}.

det(A,det(another)) --> [another],
  {
  A:agr:num <=> sing,

  A:article       <=> indefinite,
  A:in:scopeList  <=> SList,
  A:out:scopeList <=> [some(X,Scope) & Description|SList],
  A:out:quantified<=> no,
  A:binding       <=> X^Scope^Description}.

det(A,det(a,few)) --> a(_),few(_),
  {
  A:agr:num <=> plur}.

a(_)   --> [a].    % for the chart parser
few(_) --> [few].

det(A,det(the)) --> [the],
  {
  A:article  <=> definite,
```

```
  A:in <=> A:out,
  A:out:quantified <=> no}.

det(A,det(this)) --> [this],
  {
  A:agr:num <=> sing,
  A:article <=> definite,
  A:out:quantified <=> no,
  A:in <=> A:out}.

det(A,det(all)) --> [all],
  {
  A:agr:num <=> plur,

  A:in:scopeList  <=> SList,
  A:out:scopeList <=> [every(X,Scope)|SList],
%  A:in:quantified  <=> AinQ,
  A:out:quantified <=> yes,
  A:binding <=> X^Scope^_Description,
  A:article <=> indefinite}.

det(A,det(each)) --> [each],
  {
  A:agr:num <=> sing}.

det(A,det(every)) --> [every],
  {
  A:agr:num <=> sing,

  A:in:scopeList  <=> SList,
  A:out:scopeList <=> [every(X,Scope)|SList],
  A:out:quantified <=> yes,
  A:binding <=> X^Scope^_Description,
  A:article <=> indefinite}.

det(A,det(few)) --> [few],
  {
  A:agr:num <=> plur }.

det(A,det(many)) --> [many],
  {
  A:agr:num <=> plur}.
```

```
det(A,det(most)) --> [most],
  {
  A:agr:num <=> plur,

  A:in:scopeList  <=> SList,
  A:out:scopeList <=> [most(X,Scope)|SList],
  A:out:quantified <=> yes,
  A:binding <=> X^Scope^_Description,
  A:article <=> indefinite}.

det(A,det(some)) --> [some],
  {
  A:agr:num <=> sing,
  A:article  <=> indefinite,

  A:in:scopeList  <=> SList,
  A:out:scopeList <=> [some(X,Scope) & Description|SList],
  A:out:quantified <=> no,
  A:binding <=> X^Scope^Description,
  A:article <=> indefinite}.


det(A,det(some)) --> [some],
  {
  A:agr:num <=> plur,
  A:in <=> A:out                   % !!! ??? semantics
 }.

%-------------------------------------------------------

negation(neg(not)) --> [not].

%-------------------------------------------------------
% nouns
%-------------------------------------------------------


n(A,n(bone)) --> [bone],
  {
  A <=> aSing3Unhuman,
  A <=> noun(bone,neuter)}.

n(A,n(bones)) --> [bones],
```

```
  {
  A <=> plur3unhuman,
  A <=> noun(bone,plural)}.

n(A,n(book)) --> [book],
  {
  A <=> aSing3Unhuman,
  A <=> noun(book,neuter)}.

n(A,n(books)) --> [books],
  {
  A <=> plur3unhuman,
  A <=> noun(book,plural)}.

n(A,n(boy)) --> [boy],
  {
  A <=> aSing3Human,
  A <=> noun(boy,male)}.

n(A,n(boys)) --> [boys],
  {
  A <=> plur3human,
  A <=> noun(boy,plural)}.

n(A,n(cat)) --> [cat],
  {
  A <=> aSing3Unhuman,
  A <=> noun(cat,neuter)}.

n(A,n(cats)) --> [cats],
  {
  A <=> plur3unhuman,
  A <=> noun(cat,plural)}.

n(A,n(child)) --> [child],
  {
  A <=> aSing3Human,
  A <=> noun(child) }.

n(A,n(children)) --> [children],
  {
  A <=> plur3human,
  A <=> noun(child,plural)}.
```

```
n(A,n(dog)) --> [dog],
  {
  A <=> aSing3Unhuman,
  A <=> noun(dog,neuter)}.

n(A,n(dogs)) --> [dogs],
  {
  A <=> plur3unhuman,
  A <=> noun(dog,plural)}.

n(A,n(donkey)) --> [donkey],
  {
  A <=> aSing3Unhuman,
  A <=> noun(donkey,neuter)}.

n(A,n(donkeys)) --> [donkeys],
  {
  A <=> plur3unhuman,
  A <=> noun(donkey,plural)}.

n(A,n(exercise)) --> [exercise],
  {
  A <=> anSing3Unhuman,
  A <=> noun(exercise,neuter)}.

n(A,n(exercises)) --> [exercises],
  {
  A <=> nplur3unhuman,
  A <=> noun(exercise,plural)}.

n(A,n(garden)) --> [garden],
  {
  A <=> aSing3Unhuman,
  A <=> noun(garden,neuter)}.

n(A,n(gardens)) --> [gardens],
  {
  A <=> plur3unhuman,
  A <=> noun(garden,plural)}.

n(A,n(hill)) --> [hill],
  {
```

```
  A <=> aSing3Unhuman,
  A <=> noun(hill,neuter)}.

n(A,n(hills)) --> [hills],
  {
  A <=> plur3unhuman,
  A <=> noun(hill,plural)}.

n(A,n(house)) --> [house],
  {
  A <=> aSing3Unhuman,
  A <=> noun(house,neuter)}.

n(A,n(houses)) --> [houses],
  {
  A <=> plur3unhuman,
  A <=> noun(house,plural)}.

n(A,n(knife)) --> [knife],
  {
  A <=> aSing3Unhuman,
  A <=> noun(knife,neuter)}.

n(A,n(knives)) --> [knives],
  {
  A <=> plur3unhuman,
  A <=> noun(knive,plural)}.

n(A,n(lunatic)) --> [lunatic],
  {
  A <=> aSing3Unhuman,
  A <=> noun(lunatic,male)}.  % are lunatics male?

n(A,n(lunatics)) --> [lunatics],
  {
  A <=> plur3unhuman,
  A <=> noun(lunatic,plural)}.

n(A,n(man)) --> [man],
  {
  A <=> aSing3Human,
  A <=> noun(man,male)}.
```

```
n(A,n(men)) --> [men],
  {
  A <=> plur3human,
  A <=> noun(man,plural)}.

n(A,n(man)) --> [murderer],
  {
  A <=> aSing3Human,
  A <=> noun(murderer,male)}.

n(A,n(men)) --> [murderers],
  {
  A <=> plur3human,
  A <=> noun(murderer,plural)}.

n(A,n(movie)) --> [movie],
  {
  A <=> aSing3Unhuman,
  A <=> noun(movie,neuter)}.

n(A,n(movies)) --> [movies],
  {
  A <=> plur3unhuman,
  A <=> noun(movie,plural)}.

n(A,n(paper)) --> [paper],
  {
  A <=> aSing3Unhuman,
  A <=> noun(paper,neuter)}.

n(A,n(papers)) --> [papers],
  {
  A <=> plur3unhuman,
  A <=> noun(paper,plural)}.

n(A,n(soldier)) --> [soldier],
  {
  A <=> aSing3Unhuman,
  A <=> noun(soldier,neuter)}.

n(A,n(soldiers)) --> [soldiers],
  {
  A <=> plur3unhuman,
```

```
  A <=> noun(soldier,plural)}.

n(A,n(story)) --> [story],
  {
  A <=> aSing3Unhuman,
  A <=> noun(story,neuter)}.

n(A,n(stories)) --> [stories],
  {
  A <=> plur3unhuman,
  A <=> noun(story,plural)}.

n(A,n(student)) --> [student],
  {
  A <=> aSing3Human,
  A <=> noun(student,male)}.

n(A,n(students)) --> [students],
  {
  A <=> plur3human,
  A <=> noun(student,plural)}.

n(A,n(telescope)) --> [telescope],
  {
  A <=> aSing3Unhuman,
  A <=> noun(telescope,neuter)}.

n(A,n(telescopes)) --> [telescopes],
  {
  A <=> plur3unhuman,
  A <=> noun(telescope,plural)}.

n(A,n(woman)) --> [woman],
  {
  A <=> aSing3Human,
  A <=> noun(woman,female)}.

n(A,n(women)) --> [women],
  {
  A <=> plur3human,
  A <=> noun(woman,plural)}.
```

```
%--------------------------------------------------------
% personal pronouns
%--------------------------------------------------------

pronoun(A,pronoun('I')) --> [i],
  {
  A:agr:num <=> sing,
  A:agr:per <=> 1,
  A:cas <=> nom,
  A <=> pronoun(i)}.

pronoun(A,pronoun(me)) --> [me],
  {
  A:agr:num <=> sing,
  A:agr:per <=> 1,
  A:cas <=> acc,
  A <=> pronoun(i)}.

pronoun(A,pronoun(myself)) --> [myself],
  {
  A:agr:num <=> sing,
  A:agr:per <=> 1,
  A:cas <=> acc,
  A <=> refl_pronoun(i)}.

pronoun(A,pronoun(you)) --> [you],
  {
  A:agr:num <=> sing,
  A:agr:per <=> 2,
  A <=> pronoun(you) }.

pronoun(A,pronoun(he)) --> [he],
  {
  A <=> sing3,
  A:cas <=> nom,
  A <=> pronoun(male)}.

pronoun(A,pronoun(him)) --> [him],
  {
  A <=> sing3,
  A:cas <=> acc,
  A <=> pronoun(male)}.
```

```
pronoun(A,pronoun(himself)) --> [himself],
  {
  A <=> sing3,
  A:cas <=> acc,
  A <=> refl_pronoun(male)}.

pronoun(A,pronoun(she)) --> [she],
  {
  A <=> sing3,
  A:cas <=> nom,
  A <=> pronoun(female)}.

pronoun(A,pronoun(her)) --> [her],
  {
  A <=> sing3,
  A:cas <=> acc,
  A <=> pronoun(female)}.

pronoun(A,pronoun(herself)) --> [herself],
  {
  A <=> sing3,
  A:cas <=> acc,
  A <=> refl_pronoun(female)}.

pronoun(A,pronoun(it)) --> [it],
  {
  A <=> sing3,
  A <=> pronoun(neuter)}.

pronoun(A,pronoun(itself)) --> [itself],
  {
  A <=> sing3,
  A <=> refl_pronoun(neuter)}.

pronoun(A,pronoun(we)) --> [we],
  {
  A:agr:per <=> 1,
  A:agr:num <=> plur,
  A:cas <=> nom}.

pronoun(A,pronoun(us)) --> [us],

  {
```

```
  A:agr:per <=> 1,
  A:agr:num <=> plur,
  A:cas <=> acc}.

pronoun(A,pronoun(they)) --> [they],
  {
  A:agr:per <=> 3,
  A:agr:num <=> plur,
  A:cas <=> nom,
  A <=> pronoun(plural)}.

pronoun(A,pronoun(them)) --> [them],
  {
  A:agr:per <=> 3,
  A:agr:num <=> plur,
  A:cas <=> acc,
  A <=> pronoun(plural)}.

pronoun(A,pronoun(themself)) --> [themself],
  {
  A:agr:per <=> 3,
  A:agr:num <=> plur,
  A:cas <=> acc,
  A <=> refl_pronoun(plural)}.

%-------------------------------------------------------
% prepositions
%-------------------------------------------------------


p(A,X,p(X))       --> [X],   {preposition(X),A <=> prep(X)}.

% the chart parser needs the preposition(X)
% declarations too
% therefore they are declared in a separate file


%-------------------------------------------------------
% proper nouns
%-------------------------------------------------------

propernoun(A,propern('Barbara')) --> [barbara],
  {
```

```
  A <=> sing3human,
  A <=> pn('Barbara',female)}.

propernoun(A,propern('Bill')) --> [bill],
  {
  A <=> sing3human,
  A <=> pn('Bill',male)}.

propernoun(A,propern('Computer')) --> [computer], % sory for personalising
  {                                              % a stupid machine
  A <=> sing3human,
  A <=> pn('Computer',male)}.

propernoun(A,propern('Fido')) --> [fido],
  {
  A <=> sing3human,
  A <=> pn('Fido',male)}.

propernoun(A,propern('John')) --> [john],
  {
  A <=> sing3human,
  A <=> pn('John',male)}.

propernoun(A,propern('Kanga')) --> [kanga],
  {
  A <=> sing3human,
  A <=> pn('Kanga',male)}.

propernoun(A,propern('Mary')) --> [mary],
  {
  A <=> sing3human,
  A <=> pn('Mary',female)}.

propernoun(A,propern('Piglet')) --> [piglet],
  {
  A <=> sing3human,
  A <=> pn('Piglet',male)}.

propernoun(A,propern('Roo')) --> [roo],
  {
  A <=> sing3human,
  A <=> pn('Roo',male)}.
```

```
propernoun(A,propern('Scotland')) --> [scotland],
  {
  A <=> sing3human,
  A <=> pn('Scotland',neuter)}.

propernoun(A,propern('Roo')) --> [stefan],
  {
  A <=> sing3human,
  A <=> pn('Stefan',male)}.

propernoun(A,propern('Tigger')) --> [tigger],
  {
  A <=> sing3human,
  A <=> pn('Tigger',male)}.

%----------------------------------------------------
% demonstrativ pronoun
%----------------------------------------------------

pronoun(A,pronoun(this)) --> [this],
  {
  A <=> sing3
  }.

pronoun(A,pronoun(that)) --> [that],
  {
  A <=> sing3
  }.

%-----------------------------------------------------
% relativ pronoun
%-----------------------------------------------------

relpron(A,relpron(which)) --> [which],
  {
  A:human <=> no}.

relpron(A,relpron(who)) --> [who],
  {
  A:human <=> yes}.

relpron(A,relpron(that)) --> [that],
  {
```

```
  A:restr <=> yes}.

%------------------------------------------------------
% verbs
%------------------------------------------------------

vhave(A,vhave(have)) --> [have],
  {
  A <=> general_tense}.

vhave(A,vhave(has)) --> [has],
  {
  A  <=> sing3fin}.

vhave(A,vhave(had)) --> [had],
  {
  A:vform <=> fin}.

vbe(A,vbe(be)) --> [be],
  { A:vform <=> bse}.

vbe(A,vbe(been)) --> [been],
  { A:vform <=> psp}.

vbe(A,vbe(am)) --> [am],
  {
  A:vform <=> fin,
  A:agr:num <=> sing,
  A:agr:per <=> 1}.

vbe(A,vbe(is)) --> [is],
  {
  A <=> sing3fin}.

vbe(A,vbe(are)) --> [are],
  {
  A:vform <=> fin,
  (
        A:agr:num <=> plur;
        A:agr:num <=> sing,
        A:agr:per <=> 2
   )}.
```

```
vbe(A,vbe(was)) --> [was],
  {
  A:vform <=> fin,
  A:agr:num   <=> sing,
  (
        A:agr:per <=> 1;
        A:agr:per <=> 3)}.

vbe(A,vbe(were)) --> [were],
  {
  A:vform <=> fin,
  (
        A:agr:num <=> plur;
        A:agr:num <=> sing,
        A:agr:per <=> 2
  )        }.

vDo(A,vDo(do)) --> [do],
  {
  A <=> general_tense}.

vDo(A,vDo(does)) --> [does],
  {
  A <=> sing3fin}.

vDo(A,vDo(did)) --> [did],
  {
  A:vform <=> fin}.

%-----------------------------------------------------------

v(A,intr,v(go)) --> [go],
  {
  A <=> general_tense,
  A <=> intr_verb(go)}.

v(A,intr,v(lives)) --> [lives],
  {
  A <=> sing3fin,
  A <=> intr_verb(live)}.

v(A,intr,v(live)) --> [live],
  {
```

```
  A <=> general_tense,
  A <=> intr_verb(live)}.

v(A,intr,v(purr)) --> [purr],
  {
  A <=> general_tense,
  A <=> intr_verb(purr)}.

v(A,intr,v(purrs)) --> [purrs],
  {
  A <=> sing3fin,
  A <=> intr_verb(purr)}.

v(A,intr,v(sleep)) --> [sleep],
  {
  A <=> general_tense,
  A <=> intr_verb(sleep)}.

v(A,intr,v(sleep)) --> [sleeps],
  {
  A <=> sing3fin,
  A <=> intr_verb(sleep)}.

v(A,intr,v(win)) --> [win],
  {
  A <=> general_tense,
  A <=> intr_verb(win)}.

v(A,intr,v(won)) --> [won],
  {
  A <=> finPasPsp,
  A <=> intr_verb(win)}.

%----------------------------------------------------------

v(A,trans,v(beat)) --> [beat],
  {
  A <=> general_tense,
  A <=> trans_verb(beat)}.

v(A,trans,v(beats)) --> [beats],
  {
  A <=> sing3fin,
```

```
  A <=> trans_verb(beat)}.

v(A,trans,v(bounced)) --> [bounced],
  { A:vform <=> fin,
  A <=> trans_verb(bounce)}.

v(A,trans,v(chase)) --> [chase],
  {
  A <=> general_tense,
  A <=> trans_verb(chase)}.

v(A,trans,v(chases)) --> [chases],
  {
  A <=> sing3fin,
  A <=> trans_verb(chase)}.

v(A,trans,v(chased)) --> [chased],
  {
  A <=> finPasPsp,
  A <=> trans_verb(chase)}.

v(A,trans,v(eat)) --> [eat],
  {
  A <=> general_tense,
  A <=> trans_verb(eat)}.

v(A,trans,v(finished)) --> [finished],
  {
  A <=> finPasPsp,
  A <=> trans_verb(finish)}.

v(A,trans,v(finish)) --> [finish],
  {
  A <=> general_tense,
  A <=> trans_verb(finish)}.

v(A,trans,v(frightened)) --> [frightened],
  {
  A <=> finPasPsp,
  A <=> trans_verb(frighten)}.

v(A,trans,v(help)) --> [help],
  {
```

```
  A <=> general_tense,
  A <=> trans_verb(help)}.

v(A,trans,v(impressed)) --> [impressed],
  {
  A <=> finPasPsp,
  A <=> trans_verb(impress)}.

v(A,trans,v(know)) --> [know],
  {
  A <=> general_tense,
  A <=> trans_verb(know)}.

v(A,trans,v(knows)) --> [knows],
  {
  A <=> sing3fin,
  A <=> trans_verb(know)}.

v(A,trans,v(knew)) --> [knew],
  {
  A:vform <=> fin,
  A <=> trans_verb(know)}.

v(A,trans,v(like)) --> [like],
  {
  A <=> general_tense,
  A <=> trans_verb(like)}.

v(A,trans,v(likes)) --> [likes],
  {
  A <=> sing3fin,
  A <=> trans_verb(like)}.

v(A,trans,v(lose)) --> [lose],
  {
  A <=> general_tense,
  A <=> trans_verb(lose)}.

v(A,trans,v(loses)) --> [loses],
  {
  A <=> sing3fin,
  A <=> trans_verb(lose)}.
```

```
v(A,trans,v(lost)) --> [lost],
  {
  A <=> finPasPsp,
  A <=> trans_verb(lose)}.

v(A,trans,v(loves)) --> [loves],
  {
  A <=> sing3fin,
  A <=> trans_verb(love)}.

v(A,trans,v(love)) --> [love],
  {
  A <=> general_tense,
  A <=> trans_verb(love)}.

v(A,trans,v(loved)) --> [loved],
  {
  A <=> finPasPsp,
  A <=> trans_verb(love)}.

v(A,trans,v(owns)) --> [owns],
  {
  A <=> sing3fin,
  A <=> trans_verb(own)}.

v(A,trans,v(own)) --> [own],
  {
  A <=> general_tense,
  A <=> trans_verb(own)}.

v(A,trans,v(read)) --> [read],
  {
  (A <=> general_tense;
  A <=> finPasPsp),
  A <=> trans_verb(read)}.

v(A,trans,v(reads)) --> [reads],
  {
  A <=> sing3fin,
  A <=> trans_verb(read)}.

v(A,trans,v(see)) --> [see],
  {
```

```
  A <=> general_tense,
  A <=> trans_verb(see)}.

v(A,trans,v(sees)) --> [sees],
  {
  A <=> sing3fin,
  A <=> trans_verb(see)}.

v(A,trans,v(saw)) --> [saw],
  {
  A:vform <=> fin,
  A <=> trans_verb(see)}.

v(A,trans,v(wrote)) --> [wrote],
  {
  A:vform <=> fin,
  A <=> trans_verb(write)}.

v(A,trans,v(write)) --> [write],
  {
  A <=> general_tense,
  A <=> trans_verb(write)}.

v(A,ditrans,v(gives)) --> [gives],
  {
  A <=> sing3fin,
  A <=> ditrans_verb(give)}.

v(A,ditrans,v(give)) --> [give],
  {
  A <=> general_tense,
  A <=> ditrans_verb(give)}.

v(A,ditrans,v(gave)) --> [gave],
  {
  A:vform <=> fin,
  A <=> ditrans_verb(give)}.

v(A,ditrans,v(given)) --> [given],
  {
  A:vform <=> pas,
  A <=> ditrans_verb(give)}.
```

```
v(A,ditrans,v(told)) --> [told],
  {
  A <=> finPasPsp,
  A <=> ditrans_verb(tell)}.

%----------------------------------------
% control and raising constructions
% these verbs behave semanticaly like transitive verbs

v(A,inf,v(managed)) --> [managed],
  {
  A:vform <=> fin,
  A <=> trans_verb(manage)}.

v(A,inf,v(tried)) --> [tried],
  {
  A:vform <=> fin,
  A <=> trans_verb(try)}.

v(A,inf,v(seems)) --> [seems],
  {
  A <=> sing3fin,
  A <=> trans_verb(seem)}.

v(A,inf,v(want)) --> [want],
  {
  A <=> general_tense,
  A <=> trans_verb(want)}.

v(A,inf,v(wanted)) --> [wanted],
  {
  A:vform <=> fin,
  A <=> trans_verb(want)}.

v(A,inf2,v(expect)) --> [expect],
  {
  A <=> general_tense,
  A <=> ditrans_verb(expect)}.

v(A,inf2,v(expected)) --> [expected],
  {
  A <=> finPasPsp,
  A <=> ditrans_verb(expect)}.
```

```
v(A,inf2,v(persuaded)) --> [persuaded],
  {
  A:vform <=> fin,
  A <=> ditrans_verb(persuade)}.

v(A,inf,v(wants)) --> [wants],
  {
  A <=> sing3fin,
  A <=> trans_verb(want)}.

v(A,inf,v(want)) --> [want],
  {
  A <=> general_tense,
  A <=> trans_verb(want)}.

v(A,inf2,v(wants)) --> [wants],
  {
  A <=> sing3fin,
  A <=> ditrans_verb(want)}.

v(A,inf2,v(want)) --> [want],
  {
  A <=> general_tense,
  A <=> ditrans_verb(want)}.

v(A,inf2,v(wanted)) --> [wanted],
  {
  A:vform <=> fin,
  A <=> ditrans_verb(want)}.

%-------------------------------------------

v(A,that,v(believe)) --> [believe],
  {
  A <=> general_tense,
  A <=> trans_verb(believe)}.

v(A,that,v(believes)) --> [believes],
  {
  A <=> sing3fin,
  A <=> trans_verb(believe)}.
```

```
v(A,that,v(think)) --> [think],
  {
  A <=> general_tense,
  A <=> trans_verb(think)}.

v(A,that,v(say)) --> [say],
  {
  A <=> general_tense,
  A <=> trans_verb(say)}.

v(A,that,v(said)) --> [said],
  {
  A <=> finPasPsp,
  A <=> trans_verb(say)}.

v(A,that2,v(told)) --> [told],
  {
  A <=> finPasPsp,
  A <=> ditrans_verb(tell)}.

%--------------------------------------------
% terminals for the chart parser

comma(',') --> [','].
comma_or_not(',') --> [','].
comma_or_not('_') --> [].

mark('.',_) --> ['.'].
mark('?',_) --> [?].

that(that)  --> [that].
to(to)      --> [to].

what(what)  --> [what].
which(which) --> [which].
who(who)     --> [who].
who_or_what(who) --> [who].
who_or_what(what) --> [what].
```

A.3 Syntactic Templates

```
%--------------------------------------------------------------
%
% File: syntactic_templates
%
% Stefan M"uller
%
% Version 28.02.93
%
%--------------------------------------------------------------

template(sing3,A) :-
  A:agr:num <=> sing,
  A:agr:per <=> 3.

template(sing3human,A) :-
  A:human <=> yes,
  A:agr:num <=> sing,
  A:agr:per <=> 3.

template(aSing3Human,A) :-
  A:human <=> yes,
  A:agr:num <=> sing,
  A:agr:per <=> 3,
  A:det <=> a.

template(anSing3Human,A) :-
  A:human <=> yes,
  A:agr:num <=> sing,
  A:agr:per <=> 3,
  A:det <=> an.

template(aSing3Unhuman,A) :-
  A:human <=> no,
  A:agr:num <=> sing,
  A:agr:per <=> 3,
  A:det <=> a.

template(anSing3Unhuman,A) :-
  A:human <=> no,
  A:agr:num <=> sing,
  A:agr:per <=> 3,
```

```
  A:det <=> an.

template(plur3human,A) :-
  A:human <=> yes,
  A:agr:num <=> plur,
  A:agr:per <=> 3.

template(plur3unhuman,A) :-
  A:human <=> no,
  A:agr:num <=> plur,
  A:agr:per <=> 3.

template(sing3fin,A) :-
  A:vform <=> fin,
  A:agr:num <=> sing,
  A:agr:per <=> 3.

template(general_tense,A) :-
  A:vform <=> fin,
  (
          A:agr:num <=> plur;
          A:agr:num <=> sing,
          (
                  A:agr:per <=> 1;
                  A:agr:per <=> 2)
  );
  A:vform <=> bse.

template(finPasPsp,A) :-
  A:vform <=> fin;
  A:vform <=> psp;
  A:vform <=> pas.
```

A.4 Some Test Examples for the Syntax

```
test :- nl,
write('Test Examples for Syntax'),nl,nl,
write('quantified noun phrases'), nl,nl,
        t(1),t(2),t(3),t(4),t(5),t(6),t(7),t(8),
        nl,write('personal pronouns'), nl, nl,
        t(9),t(10),t(11),
        nl,write('passive constructions'),nl,nl,
        t(12),t(13),t(14),
        nl,write('dative constructions'),nl,nl,
        t(15),t(16),
        nl,write('relative clauses'),nl,nl,
        t(17),t(18),t(19),t(20),
        nl,write('topicalized sentences'),nl,nl,
        t(21),
        nl,write(questions),nl,nl,
        t(22),t(23),t(24),t(25),t(26),
        nl,write('control constructions'),nl,nl,
        t(27),t(28),t(29),t(30),t(31),
        nl,write('raising constructions'),nl,nl,
        t(32),t(33),
        nl,write('conjunction'),nl,nl,
        t(34),t(35),t(36),
        nl,write('copula constructions'),nl,nl,
        t(37),
        nl,write('sentential complements'),nl,nl,
        t(38), %t(39),
        nl,write(negation),nl,nl,
        t(40),t(41),
        nl,write(adjectives),nl,nl,
        t(42),t(43),
        nl,write('prepositional phrases'),nl,nl,
        t(44),t(45),t(46),
        statistics(runtime,[_,T]),
        write('time = '),write(T), nl.

% measure the time for all test examples
testtime :-
        tt(1),tt(2),tt(3),tt(4),tt(5),tt(6),tt(7),tt(8),
        tt(9),tt(10),tt(11),
        tt(12),tt(13),tt(14),
        tt(15),tt(16),
```

```
        tt(17),tt(18),tt(19),tt(20),
        tt(21),
        tt(22),tt(23),tt(24),tt(25),tt(26),
        tt(27),tt(28),tt(29),tt(30),tt(31),
        tt(32),tt(33),
        tt(34),tt(35),tt(36),
        tt(37),
        tt(38),tt(39),
        tt(40),tt(41),
        tt(42),tt(43),
        tt(44),tt(45),tt(46),
          statistics(runtime,[_,T]),
          write('time = '),write(T), nl.


% measure the time for all examples without gap-threading
testtimee :-
        tt(1),tt(2),tt(3),tt(4),tt(5),tt(6),tt(7),tt(8),
        tt(9),tt(10),tt(11),
        tt(12),tt(13),tt(14),
        tt(15),tt(16),
        tt(27),tt(28),tt(29),tt(30),tt(31),
        tt(32),tt(33),
        tt(34),tt(35),tt(36),
        tt(38),tt(39),
        tt(40),tt(41),
        tt(42),tt(43),
        tt(44),tt(45),tt(46),
        statistics(runtime,[_,T]),
        write('time = '),write(T), nl.

% quantified noun phrases

d(1,"All cats purr.").
d(2,"All cats chase Tigger.").
d(3,"A cat chases Tigger.").
d(4,"Every cat chases a dog.").
d(5,"A few cats chased many dogs.").
d(6,"Few cats chased most dogs.").
d(7,"The cat chased the dog.").
d(8,"Some cat chased every dog.").

% personal pronouns
```

```
d(9,"He likes her.").
d(10,"She likes him.").
d(11,"The men think that they won.").

% passive constructions

d(12,"Few papers were read by many students.").
d(13,"Many books were lost.").
d(14,"A book was lost.").

% dative constructions

d(15,"John gave the book to Mary.").
d(16,"John gave every boy a book.").

% relative clauses

d(17,"This is the cat that Fido chased.").
d(18,"This is the cat that John persuaded to eat a bone.").
d(19,"John, who knew the story, was not impressed.").
d(20,"Every student who knew the story was impressed.").

% topicalized sentences

d(21,"John, Mary knows well.").

% questions

d(22,"What do you believe that Mary told Bill that John had said?").
d(23,"Which book did Mary write?").
d(24,"Who wrote each book?").
d(25,"Which student finished every exercise?").
d(26,"Who wrote which book?").

% control constructions

d(27,"A cat tried to chase Tigger.").
d(28,"He believes that Mary managed to read the book.").
d(29,"Roo wanted to chase Tigger.").
d(30,"Roo wanted Kanga to chase Tigger.").
d(31,"Fido persuaded a cat to chase Tigger.").
```

```
% raising constructions

d(32,"A cat seems to chase Tigger.").
d(33,"Fido expected a cat to chase Tigger.").

% conjunction

d(34,"Roo chased Tigger and Tigger bounced Roo.").
d(35,"Tigger bounced Roo and Piglet.").
d(36,"Tigger bounced Roo and frightened him.").

% copula constructions

d(37,"Tigger is the cat that Fido chased.").

% sentential complements

d(38,"He believes that Mary managed to read the book.").
d(39,"What do you believe that Mary told Bill that John had said?").

% negation

d(40,"John did not see the movie.").
d(41,"Did John not see the movie?").

% adjectives

d(42,"Tigger chases a fat cat round the garden.").
d(43,"A large fat cat in the garden chases Tigger.").

% prepositional phrases

d(44,"Tigger chased a fat cat round the garden.").
d(45,"A fat cat in the garden chased Tigger.").
d(46,"John saw the man with the telescope on the hill.").



test2 :- t(70),t(71),t(72),t(73),t(74),t(75),t(76),t(77),t(78),t(79),
         t(80),t(81),t(82),t(83),t(84),t(85),t(86),t(87),t(88),t(89),
         t(90),t(91),t(92),t(93),t(94),t(95),t(96),t(97),t(98),t(99),
         t(100),t(101),t(102),t(103),t(104),t(105),t(106),t(107),
         t(108),t(109),t(110),t(111),t(112),t(113),t(114),t(115),
```

```
        t(116),t(117),t(118),t(119),t(120),t(121),t(122),
        t(123),t(124),t(125),t(126),t(127),t(128),t(129),
        statistics(runtime,[_,T]),
        write('time = '),write(T), nl.

testtime2 :- tt(70),tt(71),tt(72),tt(73),tt(74),tt(75),
             tt(76),tt(77),tt(78),tt(79),tt(80),
             tt(81),tt(82),tt(83),tt(84),tt(85),
             tt(86),tt(87),tt(88),tt(89),tt(90),
             tt(91),tt(92),tt(93),tt(94),tt(95),
             tt(96),tt(97),tt(98),tt(99),tt(100),
             tt(101),tt(102),tt(103),tt(104),tt(105),
             tt(106),tt(107),tt(108),tt(109),tt(110),
             tt(111),tt(112),tt(113),tt(114),tt(115),
             tt(116),tt(117),tt(118),tt(119),tt(120),
             tt(121),tt(122),tt(123),tt(124),tt(125),
             tt(126),tt(127),tt(128),tt(129),
             statistics(runtime,[_,T]),
             write('time = '),write(T), nl.


% conjunction

d(70,"I want to win and to go.").
d(71,"He won and he won and he won.").
d(72,"She and Mary like John.").
d(73,"Kanga chased Tigger and bounced Roo and finished the exercise.").
d(74,"He won and she chased him and her and them and finished the exercise.").
d(75,"John and Mary and Bill like her and him and her.").
d(76,"The book has been read and lost.").
d(77,"I read and lost the book.").

% disjunction

d(77,"Mary or the students or John sees him.").
d(78,"John or Mary or the students see him.").

% conjuntion + disjunction

d(79,"Mary and the students or John see him.").
d(80,"Mary and the students or John sees him.").
```

```
% tenses

d(81,"I have lost the book.").
d(82,"He has been lost.").
d(83,"He had been lost.").
d(84,"He said that he likes her.").

% passiv

d(85,"The book was given to Mary.").
d(86,"A book was lost by him.").

% relative clauses

d(87,"The man who likes her likes him.").
d(88,"The man, who likes her, likes him.").
d(89,"He likes the man I gave the book to.").
d(90,"The man I gave the book to likes him.").
d(91,"The book that I gave to the man is valuable.").
d(92,"The book I gave to the man is valuable.").


% questions

d(93,"Which student did you give the book to?").
d(94,"To which student did you give the book?").
d(95,"Who are you afraid of?").
d(96,"Do you believe that he likes her?").
d(97,"Do you believe that Mary told Bill that she likes him?").

d(98,"Do you believe that Mary told Bill that John had said that she likes him?").
d(99,"John wrote which book?").
d(100,"Do you see him?").
d(101,"Who do you see?").
d(102,"Who does see him?").
d(103,"Who sees him?").
d(104,"Does he see him?").
d(105,"What did John say?").
d(106,"Has John said that Mary likes him?").
d(107,"What had John said?").
d(108,"What do you believe that Mary told Bill?").
d(109,"Are you students?").
d(110,"Who am I?").
```

```
d(111,"Are you afraid of the dog?").
d(112,"Who has he told that he won?").


% prepositional phrases

d(113,"The man in the garden with the book told me that he lost a telescope.").

% raising constructions

d(114,"He expected her to like them.").

% control construction + passiv

d(115,"He wanted the book to be given to John.").


% adjectives

d(116,"Very good books were lost.").
d(117,"Large books and very valuable papers were lost.").

% copula constructions

d(118,"He is sure that John told Mary the story.").
d(119,"He is eager to read the book.").
d(120,"He is afraid of the dog.").
d(121,"He is stupid.").

d(122,"John, she was very afraid of.").
d(123,"I have not been afraid of John.").
d(124,"The book, he is eager to read.").

d(123,"He expected her to be eager to read the book.").
d(125,"She was able to be eager to read a book.").

% post modifier for NPs

d(126,"The man in the garden who has read books is afraid of the dog.").

% determiner a/an

d(127,"I have finished an exercise.").
```

```
d(128,"I have finished an extraordinary exercise.").
d(129,"I read an extraordinary book.").

d(130,"John, Mary told Bill that she likes.").

% to test the counter examples just call tf(X) and be happy
% if the answer is no
tf(1) :- td("He expected she to like him.").
tf(2) :- td("He was been lost.").
tf(3) :- td("The book was read is lost.").
tf(4) :- td("What John had said?").
tf(5) :- td("Who have read the book?").
tf(6) :- td("Who he sees?").
tf(7) :- td("Who are I?").
tf(8) :- td("I finished a exercise.").
tf(9) :- td("John, I am sure that told Mary the story.").
tf(10) :- td("John or Mary or the students sees him.").
tf(11) :- td("Mary, I believe that told Bill that she likes him.").
tf(12) :- td("Who John gave the book to?").


tf(21) :- td("Many books was lost.").
tf(22) :- td("John gave the book to she.").
tf(23) :- td("Many books were lost by he.").
tf(24) :- td("The man which likes her likes him.").
tf(25) :- td("The man I gave the book to likes he.").
tf(26) :- td("Tigger bounced Roo and he.").
tf(27) :- td("Her and Tigger bounced Roo.").
tf(28) :- td("She and Mary likes John.").
tf(29) :- td("Her likes John.").
tf(30) :- td("John likes Mary and see John and likes Bill.").
tf(31) :- td("John and Marry like her and him and she.").


% the grammar doesn't handle this sentence correct

% gap-threading for rel_sent causes problems
% because it allows to pass np gaps as well

problem(1) :- td("Who he has told her that likes him?").
problem(2) :- td("Who am me?").
```

A.5 Supporting Predicates

```
%-----------------------------------------------------------
% File: test_preds
%
% Stefan M(ue)ller
%
% Large Practical
%
% Version 26.02.93
%
%-----------------------------------------------------------

t(X) :- d(X,Y),write_out(Y),nl,convert_s(Y,S),test(S,_Tree),!.
p(X) :- pr(X).
p(_).
pr(X) :- d(X,Y),write_out(Y),nl,convert_s(Y,S),!,
         test(S,Tree), drucke_baum(Tree), fail.

prall :- d(_,Y),write_out(Y),nl,get0(10),convert_s(Y,S),
         test(S,Tree), drucke_baum(Tree),fail.


td(X) :- write_out(X),nl,convert_s(X,S),test(S,Tree),write(Tree),nl.

tt(X) :- d(X,Y),write_out(Y),nl,convert_s(Y,S),test(S,_Tree),fail.
tt(_).

tdisc(X) :- disc(X,Y),test_disc(Y),
            statistics(runtime,[_,T]),
            write('time = '),write(T), nl.

go :-
       repeat,
       write('>>>   '),
       lies_satz(S),
       (end(S) -> !,fail;true),
       (switch(S) -> fail;true),
       test(S,Tree),react(Tree),
       fail.
go.

disc :- %DIn:in:gl_DRS <=> [[1],[i(1),male(1),named(1,'Stefan')]],
```

```
%DIn:in:drs    <=> [[],[]],
%DIn:in:focus  <=> [1],
init(DIn),
retractall(boun(_)),      % just in case
retractall(counter(_)),
asserta(counter(2)),
disc(DIn).

disc.

disc(DRSIn) :-
write('>>>   '),
lies_satz(S),
(end(S) -> !,fail;true),
( switch(S),disc(DRSIn);
          drs_sent(S,DRSIn,DRSOut),
  disc(DRSOut)
).

end([end,'.']).

switch([trace,'.'])   :- trace.
switch([trees,'.'])   :- trees.
switch([notrees,'.']) :- notrees.
switch([drs,'.'])     :- drs.
switch([nodrs,'.'])   :- nodrs.
switch([restr,'.'])   :- restr.
switch([norestr,'.']) :- norestr.


react(B) :- trees(on),!, drucke_baum(B).
react(B) :- nonvar(B),write(yes),nl.

:- dynamic trees/1.
:- dynamic drs/1.
:- dynamic restr/1. % uncomment for SICSTUS Prolog

trees   :- retract(trees(_)),assert(trees(on)).
notrees :- retract(trees(_)),assert(trees(off)).

drs     :- retract(drs(_)),assert(drs(on)).
nodrs   :- retract(drs(_)),assert(drs(off)).
```

```
restr   :- retract(restr(_)),assert(restr(on)).
norestr :- retract(restr(_)),assert(restr(off)).

trees(off).
drs(on).
restr(off).

init(Din) :-
       nl,
       write('Hi, I am Computer. And who are you?'),nl,
       write('Please type in your first name: '),
       lies_satz([Name,'.']),
       name(Name,[F|Rest]),
       U is F - 32,
       name(NName,[U|Rest]),
       Din:in:gl_DRS <=> [[1,2],[i(1),named(1,NName),you(2),named(2,'Computer')]],
       Din:in:drs    <=> [[],[]],
       Din:in:focus  <=> [1,2].

test(S,Tree) :-
       parse(_,S,Tree).


%------------------------------------------------------------
% File: start
%
% Stefan M(ue)ller
%
% Large Practical
%
% Version 28.01.93
%
%------------------------------------------------------------


:- [%sic,
%    quint,
    '../Prolog-Lib/hu',
    '../Prolog-Lib/lib',
    'operators.hu',
    graph,
    test_preds,
    prepositions,
```

```
    %lexicon,grammar,
    syntactic_templates,
    semantic_templates,
    discourse,lesen,output,
    dexamples,                   % semantic / discourses
    examples].                   % syntax

%:- prolog_flag(single_var_warnings,_,off).
%:- [baum].
%:- prolog_flag(single_var_warnings,_,on).

:- warn(off),
    ['../Prolog-Lib/baum.hu',
    'lexicon.pro', 'grammar.pro'    % hu - prolog
                                    % you must compile the grammar
                                    % and the lexicon
                                    % using dcgtrans
                                    % dcg grammar.
                                    % dcg lexicon.
   ],
   warn(on).

%drucke_baum(T) :- write(T),nl.

/*
% Quintus + Sicstus
parse(s_eng(DRS),S,Tree) :- s_eng(DRS,Tree,S,[]).
parse(question(DRS),S,Tree) :- question(DRS,Tree,S,[]).
*/

parse(s_eng(DRS),S,Tree) :- s_eng(S,[],DRS,Tree).
parse(question(DRS),S,Tree) :- question(S,[],DRS,Tree).


in_mode(dcg).


%------------------------------------------------------------
%
% File: lesen
%
% Stefan M(ue)ller
%
```

```
% Large Practical
%
% contains slightly modified code
% from Lehner: Prolog und Linguistik
%
%-----------------------------------------------------------

lies_satz(S):- lies_satz_x(S),
         (seeing(user) , get0(10), ! ; true ) .

lies_satz_x([W|Ws]):- repeat,
                    lies_atom(W,Trenner),
                    weiter(W,Trenner,Ws),
                    !.
/* 1. Fall: nichts wird gelesen ausser einem Ende-Zeichen */

weiter(T,T,[]):- ende(T).

/* 2. Fall: ein Wort wurde gelesen und ein Ende-Zeichen */

weiter(W,T,[T]):- nonvar(W), ende(T), !.

% mein Fall

weiter(W,',',[','|Ws]) :- nonvar(W), lies_satz_x(Ws).

/* 3. Fall: ein Wort wurde gelesen und irgendein Trenner */

weiter(W,_,Ws):- nonvar(W), lies_satz_x(Ws).


lies_atom(W,Trenner):- get_ascii_listen(L1,L2),
                       transformiere(L1,W,L2,Trenner).


get_ascii_listen(L,T):- get0(C),
                        bilde(C,L,T).

bilde(C,[],[C]):- trenner(C), !.
bilde(C,[C1|Rest],Trenner):- trans_char(C,C1),
                             !,
                             get_ascii_listen(Rest,Trenner).
```

```
transformiere([],_,L2,Trenner):- !, name(Trenner,L2).
transformiere(L,W,L2,Trenner):- name(W,L),
                                name(Trenner,L2).

trans_char(D,C):- D > 64, D < 91, !, C is D + 32.
trans_char(C,C).

comma(44).

ende(?).
ende('.').
ende('!').

trenner(32).
 /* Blank */
trenner(10).
 /* CR */
trenner(33).
 /* ! */
trenner(63).
 /* ? */
trenner(46).
 /* . */
trenner(59).
 /* ; */
trenner(58).
 /* : */
trenner(44).
 /* , */

% Version fuer Listen von Stefan M(ue)ller

convert_s(S,L):- lies_satz_x(S,L).

lies_satz_x(Sin,[W|Ws]):- repeat,
                    lies_atom(Sin,Sout,W,Trenner),
                    weiter(Sout,W,Trenner,Ws),
                    !.
/* 1. Fall: nichts wird gelesen ausser einem Ende-Zeichen */

weiter(_,T,T,[]):- ende(T).

/* 2. Fall: ein Wort wurde gelesen und ein Ende-Zeichen */
```

```
weiter(_,W,T,[T]):- nonvar(W), ende(T), !.

% mein Fall

weiter(S,W,',',[','|Ws]):-nonvar(W), lies_satz_x(S,Ws).

/* 3. Fall: ein Wort wurde gelesen und irgendein Trenner */

weiter(S,W,_,Ws):- nonvar(W), lies_satz_x(S,Ws).


lies_atom(Sin,Sout,W,Trenner):- get_ascii_listen(Sin,Sout,L1,L2),
                       transformiere(L1,W,L2,Trenner).


get_ascii_listen([C|R],Sout,L,T):-
                        bilde(R,Sout,C,L,T).

bilde(Rin,Rout,C,[],[C]):- trenner(C), !, skip_blank(Rin,Rout).
bilde(Sin,Sout,C,[C1|Rest],Trenner):- trans_char(C,C1),
                             !,
                             get_ascii_listen(Sin,Sout,Rest,Trenner).


skip_blank([32|R],Out) :- skip_blank(R,Out).
skip_blank(X,X).


%-------------------------------------------------------
%
% File: sic
%
% Stefan M(ue)ller
%
% Large Practical
%
% some supporting predicates
%
%-------------------------------------------------------


:- op(34,xfx,\=).
```

```
X \= X :- !, fail.
_ \= _.

append([],X,X).
append([X|T],Z,[X|T1]) :- append(T,Z,T1).


xappend([],X,X) :- !.
xappend([X|T],Z,[X|T1]) :- xappend(T,Z,T1), !.

member(X,[X|_]).
member(X,[_|Ys]) :- member(X,Ys).

struct(S) :- nonvar(S),
     not atomic(S),
     not numeric(S).   % selbst geschrieben, anhand spaerlicher Infos


l(X) :- listing(X).


numeric(X):- atomic(X),
     not atom(X).

del(F,[F|X],X) :- !.
del(F,[E|X],[E|Y]) :- del(F,X,Y).

last([_|R],X):- last(R,X).
last([X],Y):- X == Y.

reverse(L,RL) :- nreverse(L,[],RL),!.
nreverse([],L,L) :- !.
nreverse([H|T],R,L) :- nreverse(T,[H|R],L),!.



write_out([K|R]):- put(K), write_out(R).
write_out([]).


%-------------------------------------------------------------
%
```

```
% File: graph
%
% Stefan M(ue)ller
%
% Large Practical
%
% 20.11.92
%
% Contains code from the lecture notes
%
%-------------------------------------------------------------

X <=> Y :-
  denotes(X,Z1),
  denotes(Y,Z2),
  graph_unify(Z1,Z2).

denotes(X,X) :- var(X), !.
denotes(X,Y) :- template(X,Y). % makes templates backtrackable
denotes(X:F,V) :- !, denotes(X,V1), member(F:V2,V1), !, V = V2.
denotes(X,X) :- not template(X,_).

graph_unify(X,X) :- !.
graph_unify([A:V1|R1],F2) :-
  del(A:V2,F2,R2),
  graph_unify(V1,V2),
  graph_unify(R1,R2),!.


:- multifile template/2.  % comment for HU-Prolog

template(dummy,dummy). % if no templates are defined

out(X) :- out(X,0).

out(X,_) :- var(X),!.

out([X:Y|Rest],L) :-
atomic(Y),!,
tab(L), write(X), write(' : '), write(Y), nl,
out(Rest,L).
```

```
out([X:Y|Rest],L) :-
tab(L), write(X), write(' : '), nl,
L2 is L + 5,
out(Y,L2),
out(Rest,L).


%-----------------------------------------------------------
%
% File: baum
%
% contains a program from
% Lehner: Prolog und Linguistik
%
%-----------------------------------------------------------


atomic_laenge(A,N):- atom(A),
                     !,
                     name(A,L),
                     listen_laenge(L,N).

atomic_laenge(Nr,N):- numeric(Nr),
                      !,
                      name(Nr,L), %St. Mue. number(Nr,L),
                      listen_laenge(L,N).

listen_laenge([_|R],N):- listen_laenge(R,M),
                         !,
                         N is M + 1.
listen_laenge([],0).

leerstellen(N):- N > 0,
                 put(32),
          /* ASCII-Wert fuer Leerstelle: 32 */
                 M is N - 1,
                 leerstellen(M).
leerstellen(0).


leerstellen(N):- N =< 0.
```

```
leerstellen(N):-
             M is N,
             leerstellen(M).

leerstellen_x(N):- N > 0,
                 put(32),
          /* ASCII-Wert fuer Leerstelle: 32 */
                 M is N - 1,
                 leerstellen_x(M).
leerstellen_x(N):- N =< 0.



/* Drucken von Baeumen */



ana(Baum,[k(Baum,Pos)],L_aussen,R_aussen):-
                      atomic(Baum),
                      !,
                      atomic_laenge(Baum,N),
                      R_aussen is L_aussen + N + 2,
                      Pos is (R_aussen + L_aussen ) // 2 .

ana(Baum,[k(F,Pos),[k(Nachfolger,Pos)]],L_aussen,R_aussen):-
            Baum =.. [F,Nachfolger],
            atomic(Nachfolger),
            !,
            atomic_laenge(Nachfolger,N1),
            atomic_laenge(F,N2),
            max(N1,N2,N),
            R_aussen is L_aussen + N + 2,
            Pos is (R_aussen + L_aussen) // 2.

%ana(Baum,[k(F,Pos),L],L_aussen,R_aussen):-
ana(Baum,[k(F,Pos),Liste_nachf],L_aussen,R_aussen):-
            Baum =.. [F|Nachfolger],
            atomic_laenge(F,N),
            R_Knoten is L_aussen + N + 2,
            ana_nachfolger(Nachfolger,L_aussen,
                                   R_unt_baum,Liste_nachf),
            berechne_pos(L_aussen,Pos,Liste_nachf,
```

```
                                 R_unt_baum,R_Knoten,R_aussen).

ana_nachfolger([Baum|Rest],L_aussen,R_aussen,L):-
             Rest \= [],
             ana(Baum,L1,L_aussen,Mitte),
             ana_nachfolger(Rest,Mitte,R_aussen,L2),
             append(L1,L2,L).
ana_nachfolger([Baum],L_aussen,R_aussen,L):-
           ana(Baum,L,L_aussen,R_aussen).

berechne_pos(Links,Pos,L,R,M,R):-
                            M =< R,
                            !,
                            erster_knoten(L,Pos1),
                            letzter_knoten(L,Pos2),
                            Pos is (Pos1 + Pos2) // 2.
berechne_pos(Links,Pos,L,R,M,M):-
                            M > R,
                            Pos is (M + Links) // 2.

drucke_baum(S):- ana(S,L,0,R),
                 nl,
                 dr_baum(L)
                 ,!.                           % St. Mue.

dr_baum(L):-
          L = [_|_],
          drucke_knoten(L,0),
          nl,
          drucke_aeste(L,0),
          nl,
          drucke_zweige(L,L1,0),
          nl,
          dr_baum(L1).
dr_baum([]).

drucke_knoten([X|R],Spalte1):-
                 dr_kn(X,Spalte1,Spalte2),
                 !,
                 drucke_knoten(R,Spalte2).

drucke_knoten([_|R],Spalte):-
                 !,
```

```
                 drucke_knoten(R,Spalte).

drucke_knoten([],_).

dr_kn(k(X,Pos),S1,S2):-
             atomic_laenge(X,N),
             leerstellen(Pos - N//2 - S1),
             S2 is Pos + N//2 + N mod 2,
             write(X).

/* Blaetter */
drucke_aeste([X,Y|Rest],S):-
                Y \= [_|_],
                !,
                drucke_aeste([Y|Rest],S).

drucke_aeste([X],S):-
                !.

/* nicht-verzweigende, z.B. lexikalische Kategorien */
drucke_aeste([k(X,Pos),L|Rest],S1):-
                knoten_zahl(L,1),
                !,
                leerstellen(Pos-S1),
                write('|'),
                S2 is Pos + 1 ,
                drucke_aeste(Rest,S2).

/* normale Kategorien */
drucke_aeste([X,L|Rest],S1):-
                not knoten_zahl(L,1),
                !,
                dr_ae(L,S1,S2),
                drucke_aeste(Rest,S2).

drucke_aeste([],_).

/* dr_ae(Dominierender_Knoten,[Tochter_links,L1,Tochter_rechts,L2]  */

dr_ae(L,S1,S2) :-
               erster_knoten(L,Pos1),
               letzter_knoten(L,Pos2),
               Mitte is (Pos1 + Pos2) // 2,
```

```
               leerstellen(Pos1-S1+1),
               n_mal(Mitte-Pos1-1,'_'),
               write('|'),
               n_mal(Pos2-Mitte-2,'_'),
               S2 is Pos2 - 1.


erster_knoten([k(_,Pos)|L],Pos).

/* einen letzten Knoten gibt es nur dann, wenn
   es einen ersten Knoten gibt */

letzter_knoten([_|R],Pos):-
                        letzter_knoten(R,Pos).

/* 1. Fall: Blatt */
letzter_knoten([k(_,Pos)],Pos):- !.
/* 2. Fall: dominierender Knoten */
letzter_knoten([k(_,Pos),[_|_]],Pos).



/* Blaetter */
drucke_zweige([X,Y|Rest],Rest1,S1):-
                Y \= [_|_],
                !,
                drucke_zweige([Y|Rest],Rest1,S1).

drucke_zweige([X],[],S):-
                X \= [_|_],
                !.

/* nicht-verzweigende, z.B. lexikalische Kategorien */
drucke_zweige([k(_,Pos),X|Rest],L,S1):-
                knoten_zahl(X,1),
                !,
                leerstellen(Pos-S1),
                write('|'),
                S2 is Pos + 1,
                drucke_zweige(Rest,L1,S2),
                append(X,L1,L).

/* normale Kategorien */
```

```
drucke_zweige([X,[K|R]|Rest],L,S1):-
                !,
                dr_zw(X,[K|R],S1,S2),
                drucke_zweige(Rest,L1,S2),
                append([K|R],L1,L).
drucke_zweige([],[],_).

/* dr_zw(Dominierender_Knoten,[Tochter_links,L1,Tochter_rechts,L2]  */

dr_zw(k(X,Pos),L,S1,S4) :-
                       erster_knoten(L,Pos1),
                       leerstellen(Pos1-S1),
                       write(/),
                       S2 is Pos1+1,
                       knoten_dazwischen(L,S2,S3),
                       letzter_knoten(L,Pos2),
                       leerstellen(Pos2 - S3 - 1),
                       write('\'),
                       S4 is Pos2 .
knoten_dazwischen([k(_,_)|R],S1,S2):-
                    drucke_zweige_fuer_knoten(R,S1,S2).


drucke_zweige_fuer_knoten([k(_,_)],S,S):- !.
drucke_zweige_fuer_knoten([k(_,_),[Letzte|Nachfolger]],S,S):- !.
drucke_zweige_fuer_knoten([k(_,Pos)|R],S1,S2):-
                           !,
                           leerstellen(Pos-S1),
                           write('|'),
                           S3 is Pos + 1,
                           drucke_zweige_fuer_knoten(R,S3,S2).
drucke_zweige_fuer_knoten([_|R],S1,S2):-
                           drucke_zweige_fuer_knoten(R,S1,S2).

/* Hilfspraedikate */

knoten_zahl([k(_,_)|L],1):- not member_x(k(_,_),L).

n_mal(Arith,A):- X is Arith,
                 n_mal_x(X,A).

n_mal_x(N,A):- N > 0,
               !,
```

```
               write(A),
               M is N - 1,
               n_mal(M,A).
n_mal_x(N,_):- N =< 0.

max(X,Y,X):- X >= Y, !.
max(X,Y,Y).

member_x(X,L):- member(X,L), !.
```

B Parser

B.1 Top-Down Parser for Filler-Gap DCGs

B.1.1 The Parser

```
%---------------------------------------------------
%
% File: td_fg
%
% Stefan M(ue)ller
%
% Version 11.02.93
%
%  interpreter for FG-DCGs with restrictions
%
% The basic idea is out of Prolog and NLA
%
%---------------------------------------------------


:- op(1200,xfx,--->).
:- op(300, xfx, fills).
:- op(299, xf, island).

parse(NT island, F0-F0) --> !,
   parse(NT,F0-F0).

parse((NT fills GapType,Body2), F0-F) --> !,
   parse(NT,F0-F0),
   parse(Body2,[GapType|F0]-F).

parse((Body1,Body2),F0-F) --> !,
   parse(Body1,F0-F1),
   parse(Body2,F1-F).

parse(NT, F0-F) -->
   {NT ---> Body},
   parse(Body,F0-F).

parse([],F0-F0) --> [].

parse([Word|Rest],F0-F) -->
```

```
   [Word],
   parse(Rest,F0-F).

parse({Goals},F0-F0) --> {call(Goals)}.

parse(GapType,[GapType|F0]-F0) -->
   {gap(GapType) ---> Body},
   parse(Body,F0-F0).


test(S,Tree) :- parse(s_eng(Tree),[]-[],S,[]).



%-------------------------------------------------------
% File: startfg
%
% Stefan M(ue)ller
%
% Large Practical
%
% Version 28.01.93
%
%-------------------------------------------------------

:- ['../sic'].

:- ['../graph',td_fg,gram_fg_dcg,lex_fg_dcg,'../lesen',
    '../examples','../test_preds'].

:- prolog_flag(single_var_warnings,_,off).
:- ['../baum'].
:- prolog_flag(single_var_warnings,_,on).
```

B.1.2 The Grammar in Filler-Gap Notation

```
%-----------------------------------------------------------
%
% File: gram_fg_dcg
%
% Stefan M(ue)ller
%
% Large Practical
%
% Part I / Syntax for a fragment of English
%
% The Grammar / a FG-DCG version
%
% Version 11.02.93
%
%
%-----------------------------------------------------------

:- multifile '--->'/2.

%-----------------------------------------------------------
% sentences
%-----------------------------------------------------------

s_eng(S) ---> s_fin(S),['.'].
s_eng(S) ---> question(S),[?].

s_fin(S) ---> s(A,S1) island,more_s_fin(S1,S),
  { A:vform <=> fin}.

more_s_fin(S,S) ---> [].
more_s_fin(S1,S3) ---> coord(_,C), s_fin(S2),
      more_s_fin(coordination(S1,C,S2),S3).

s(A2,s(NP,VP)) ---> np(2,A1,NP),vp(2,A2,VP),
{
A1:cas <=> nom,
A1:agr <=> A2:agr}.

% topicalization, objects in first position
s(A,s(NP,comma(','),S)) --->
np(2,A1,NP) island fills np(2,A1,np('_')), [','],
```

```
s(A,S),
  {
  A1:cas <=> acc}.

%----------------------------------------------------------------
% questions
%----------------------------------------------------------------

question(Q) ---> que_(Q) island.
question(Q) ---> que(Q).
question(Q) ---> s(A,Q) island,
  {A:vform <=> fin}. % see documentation

que(que(interr(what),Q)) ---> [what] fills rel_sent(rel_sent('_')),
que_(Q).

que(que(interr(who),Q)) ---> [who] fills np(2,A,np('_')), que_(Q),
  {A <=> sing3}.

que(que(interr(who),Q)) ---> [who] fills np(2,An,np('_')),  s(As,Q),
  {
  An:cas  <=> nom,
  An <=> sing3,
  As:vform <=> fin}.

que(que(interr(which),N,Q)) ---> [which], n(A,N) fills np(2,A,np('_')),
que_(Q).

que(que(pp(P,interr(which),N),Q)) ---> p(Pform,P), [which], n(_,N)
fills pp(Pform,_P),
que_(Q).

que_(que(AUX,NP,VP)) ---> vaux(A,AUX), np(2,A,NP),
     vp(2,A1,VP),
  {
  A:vform <=> fin,
  A1:vform <=> bse}.
que_(que(AUX,NP,NEG,VP)) ---> vaux(A,AUX),
 np(2,A,NP),
 negation(NEG),
 vp(2,A1,VP),
  {
  A:vform <=> fin,
```

```
  A1:vform <=> bse}.
que_(que(HAVE,NP,VP)) ---> vhave(A,HAVE),
      np(2,A,NP),
      vp(2,A1,VP),
  {
  A:vform <=> fin,
  A1:vform <=> pas}.

que_(que(BE,NP,PCS)) ---> vbe(A,BE),
    np(2,A,NP),
    pcs(A,PCS),
  {
  A:vform <=> fin}.

%----------------------------------------------------------------
% adjectives
%----------------------------------------------------------------

adjs(2,A,adj(DEG,ADJ)) ---> deg(A,DEG), adjs(1,A,ADJ).
adjs(2,A,ADJ)       ---> adjs(1,A,ADJ).
adjs(1,A,ADJ)       ---> adj(0,A,ADJ1),moreadj(ADJ1,ADJ).

moreadj(ADJ,ADJ) ---> [].
moreadj(ADJ,adjs(ADJ,ADJ2)) ---> adj(0,_,ADJ1),
moreadj(ADJ1,ADJ2).


%----------------------------------------------------------------
% adjective phrases
%----------------------------------------------------------------

adjp(2,adjp(DEG,ADJP)) ---> deg(_,DEG), adjp(1,ADJP).
adjp(2,ADJP) ---> adjp(1,ADJP).
adjp(1,adjp(ADJ,PP))   ---> adj(pp,A,ADJ), pp(Pform,PP),
  { A:pform <=> Pform}.
adjp(1,adjp(ADJ,rel_sent(subord(that),RS))) ---> adj(that,ADJ),
         [that],
     s(A,RS) island,
  { A:vform <=> fin}.
adjp(1,adjp(ADJ,VP))   ---> adj(inf,ADJ), vp(2,A,VP),
  { A:vform <=> inf}.
adjp(1,adjp(ADJ))      ---> adj(0,_,ADJ).
```

```
%-------------------------------------------------------------
% noun phrases
%-------------------------------------------------------------

gap(np(2,_A,np('_'))) ---> [].
%np(2,A,gap(np(A)),nogap,np('_')) ---> [].

np(2,A,NP) ---> np(a3,A,NP).

np(a3,A1,NP) ---> np(a2,A2,NP1),more_np(A3,A2,Amo,NP1,NP),
  {
  A1:cas <=> A2:cas,
  A1:cas <=> A3:cas,
  A1:agr <=> Amo:agr}.

more_np(_,Ami,Amo,NP,NP) ---> [],
  { Ami:agr <=> Amo:agr}.

more_np(A1,_,Amo,NP,NP2) ---> coord(and,C),
  np(a3,A2,NP1),
  more_np(A3,Ami,Amo,coordination(NP,C,NP1),NP2),
  {
  A1:cas <=> A2:cas,
  A1:cas <=> A3:cas,
  Ami:agr:num <=> plur}.

more_np(A1,_,Amo,NP,NP2) ---> coord(or,C),
  np(a3,A2,NP1),
  more_np(A3,A2,Amo,coordination(NP,C,NP1),NP2),
  {
  A1:cas <=> A2:cas,
  A1:cas <=> A3:cas}.


np(a2,A,np(PN))    ---> propernoun(A,PN).
np(a2,A,np(PN,PM)) ---> propernoun(A,PN),
non_restr_rel_cl(A,PM).

np(a2,A,np(PN)) ---> pronoun(A,PN).

% for Questions
np(a2,A,np(interr(which),N)) ---> [which],n(A,N).
```

```
np(a2,A,PM) ---> det(A,DETP), np(1,A,NP),
 post_modifier_np(A,np(DETP,NP),PM).
np(a2,A,PM) ---> np(1,A,NP), post_modifier_np(A,NP,PM),
  {
  A:agr:num <=> plur}.

np(1,A,N) ---> n(A,N).
np(1,A1,np(ADJ,N)) ---> adjs(2,A2,ADJ), n(A3,N),
  {
  A1:det <=> A2:det,
  A1:agr <=> A3:agr,
  A1:cas <=> A3:cas}.

post_modifier_np(_,PM,PM) ---> [].
post_modifier_np(A,PM1,PM) ---> pp(_,PP) island,
  post_modifier_np(A,np(PM1,PP),PM).
post_modifier_np(A,PM1,PM) ---> restr_rel_cl(A,RS),
  post_modifier_np(A,np(PM1,RS),PM).

post_modifier_np(A,PM1,PM) ---> non_restr_rel_cl(A,RS),
  post_modifier_np(A,np(PM1,RS),PM).

non_restr_rel_cl(A,
non_restr(comma(','),RP,S,comma(','))) ---> [','],
relpron(A,RP) fills np(2,A2,np('_')),
s(As,S),[','],
  {
  A:restr <=> no,
  As:vform <=> fin,
  A:agr <=> A2:agr}.
  % agr must be the same, but case might be different

restr_rel_cl(A,restr(S)) ---> [] fills np(2,A2,np('_')),
      s(As,S),
  {
  As:vform <=> fin,
  A:agr    <=> A2:agr,
  A2:cas   <=> acc}.

restr_rel_cl(A1,restr(RP,S)) ---> relpron(A1,RP)
 fills np(2,A2,np('_')),
 s(As,S),
```

```
  {
  As:vform <=> fin,
  A1:agr   <=> A2:agr}.
  % agr must be the same, but case might be different

%----------------------------------------------------------------
% prepositional phrases
%----------------------------------------------------------------

gap(pp(_Pform,pp('_'))) ---> [].
%pp(Pform,gap(pp(Pform)),nogap,pp('_')) ---> [].

pp(Pform,pp(P,NP)) ---> p(Pform,P), np(2,A,NP),
  {
  A:cas <=> acc}.

%----------------------------------------------------------------
% verb phrases
%----------------------------------------------------------------


vp(2,A,VP) ---> vp(a1,A,VP).

vp(2,A,VP) ---> vp(a1,A,VP1) island,
coord(_,C),
vp(a1,A,VP2) island,
more_vp(A,coordination(VP1,C,VP2),VP).

more_vp(_,VP,VP) ---> [].
more_vp(A,VP1,VP) ---> coord(_,C),
   vp(a1,A,VP2) island,
   more_vp(A,coordination(VP1,C,VP2),VP).

vp(a1,A,PM) ---> vp(1,A,VP),
     post_modifier_vp(VP,PM).

post_modifier_vp(P,P) ---> [].
post_modifier_vp(P,mod_vp(P,ADV)) ---> adv(ADV).
post_modifier_vp(PM1,PM) ---> pp(_,PP),
  post_modifier_vp(mod_vp(PM1,PP),PM).

%----------------------------------------------------------------
vp(1,A1,vp(BE,VP)) ---> vbe(A1,BE),  vp(_,A2,VP),
```

```
   {
   A2:vform <=> pas}.          % passiv

vp(1,A1,vp(BE,NEG,VP)) ---> vbe(A1,BE), negation(NEG),
vp(_,A2,VP),
   {
   A2:vform <=> pas}.          % passiv

%-----------------------------------------------------------------
vp(1,A1,vp(HAVE,VP)) ---> vhave(A1,HAVE), vp(1,A2,VP),
   {
   A2:vform <=> psp}.

vp(1,A1,vp(HAVE,NEG,VP)) ---> vhave(A1,HAVE), negation(NEG),
  vp(1,A2,VP),
   {
   A2:vform <=> psp}.

%-----------------------------------------------------------------
vp(1,A1,vp(AUX,VP)) ---> vaux(A1,AUX), vp(1,A2,VP),
   {
   A2:vform <=> bse}.

vp(1,A1,vp(AUX,NEG,VP)) ---> vaux(A1,AUX),negation(NEG),
 vp(1,A2,VP),
   {
   A2:vform <=> bse}.


%-----------------------------------------------------------------
vp(1,A,V) ---> v(intr,A,V).

%-----------------------------------------------------------------
vp(1,A,vp(VBE,PCs)) ---> vbe(A,VBE), pcs(A,PCs).

pcs(_,ADJP) ---> adjp(2,ADJP).
pcs(_,adjp(NEG,ADJP)) ---> negation(NEG), adjp(2,ADJP).

pcs(A,NP) ---> np(2,A1,NP),
  { A:agr:num <=> A1:agr:num}. % in genaral
pcs(A,adjp(NEG,NP)) ---> negation(NEG), np(2,A1,NP),
  { A:agr:num <=> A1:agr:num}. % in genaral
```

```
% problem case in NP "it was me."

%------------------------------------------------------------
vp(1,A,vp(V,NP)) ---> vp(trans,A,V),np(2,A1,NP),
  {
  A1:cas <=> acc}.

%------------------------------------------------------------
vp(1,A,vp(V,NP1,NP2)) --->  vp(ditrans,A,V), np(2,A1,NP1),
np(2,A2,NP2),
  {
  A1:cas <=> acc,
  A2:cas <=> acc}.
vp(1,A,vp(V,NP)) ---> vp(ditrans,A,V), np(2,A1,NP),
  {
  A1:cas <=> acc}.

%------------------------------------------------------------
vp(1,A,vp(V,VP)) ---> vp(inf1,A,V), vp(2,A2,VP),
  {
  A2:vform <=> inf}.

vp(1,A,vp(V,NP,VP)) --->  vp(inf2,A,V), np(2,A1,NP),
vp(2,A2,VP),
  {
  A1:cas <=> acc,
  A2:vform <=> inf}.


%------------------------------------------------------------

gap(rel_sent(rel_sent('_'))) ---> [].

rel_sent(rel_sent(subord(that),S)) ---> [that], s(A,S),
  { A:vform <=> fin}.

vp(1,A,vp(V,RS)) ---> vp(that,A,V),
rel_sent(RS).

vp(1,A,vp(V,NP,rel_sent('_'))) ---> vp(that2,A,V),
   np(2,A2,NP) island,
   rel_sent(rel_sent('_')),
  { A2:cas <=> acc}.
```

```
vp(1,A,vp(V,NP,RS)) ---> vp(that2,A,V),
   np(2,A2,NP),
   rel_sent(RS),
  { A2:cas <=> acc}.

vp(1,A1,vinf(to,V))   ---> [to], vp(1,A2,V),
  {
  A1:vform <=> inf,
  A2:vform <=> bse}.

vp(Cat,A,VP) ---> v(Cat,A,VP).
vp(Cat,A,vp_conj(V1,C,V2)) --->v(Cat,A,V1),coord(_,C),v(Cat,A,V2).

% end of file
```

B.2 Left-Corner Recogniser

B.2.1 The Link Generator

```
%--------------------------------------------------
%
% File: generator
%
% Stefan M(ue)ller
%
% Version 19.02.93
%
% generates all links needed for the left-corner
% parser
%
%--------------------------------------------------


:- ['../sic'].


:- op(33,yfx,':').
:- op(34,yfx,'<=>').
:- op(1200,xfx,--->).

:- dynamic link/2.

make_link(Gram,Lex,File) :-

   consult(Gram),
   %------------
   % this is only for the dummy rule det ---> a,few
   % which replaced the former DCG version det(...) --> [a,few]
   consult(Lex),
   retractall(--->(_LHS,[_Word])),
   retractall(--->(_LHS,([_Word],_Restr))),
   %------------
   write('Computing links and writing to the file "'),
   write(File),write('" ...'),nl,
   % compute links
   foreach(
      (pair(LHS,FIRST), LHS \= FIRST),
      assert_once(link(FIRST,LHS))
```

```
   ),
   % compute transitive closure
   foreach(
      (links(A,B), A \= B),
       assert_once(link(A,B))
   ),
   % write links to the file
   tell(File),
   foreach(
      link(A,B),
      (write(link(A,B)),write('.'),nl
      )
   ),
   % make link relation reflexive
   write(link(A,A)),write('.'),nl,
   told.

assert_once(X) :- X,!.
assert_once(X) :- assert(X).


foreach(X,Y) :-
   X, once(Y), fail.
foreach(_,_).

once(X) :- X, !.


% compute the transitive closure

links(A,B) :- link(A,B).
links(A,B) :- link(A,C),links(C,B).


%-----------------------------------
% normal rules

pair(L,R) :-
   (LHS ---> RHS),
   conj_to_list(RHS,[First|_Rest]),
   LHS =.. [L|_],
   First =.. [R|_].
```

```
%----------------------------------
% rules in which the first element of
% the RHS is optional because it can
% be replaced by a gap

pair(L,R) :-
   (LHS ---> RHS),
   conj_to_list(RHS,[First,Second|_Rest]),
   ((First ---> []);(First ---> [],_Restr)),
   LHS =.. [L|_],
   Second =.. [R|_].

   conj_to_list((A,B),[A|List]) :- !,
   conj_to_list(B,List).
   conj_to_list(A,[A]).

:- make_link(gram_lc,lex_lc,links_lc).
:- halt.
```

B.2.2 The Parser

```
%-----------------------------------------------------
%
% File: lc
%
% Stefan M(ue)ller
%
% Version 09.02.93
%
% left corner interpreter for DCGs with restrictions
% if DAGs are used as arguments they have to occur in the
% first position
%
% The basic idea is out of the lecture notes.
%
%-----------------------------------------------------


   recognise_l([],String,String):- !.

   recognise_l([H|T]) -->
       recognise(H),
       recognise_l(T).


   recognise({Goal}) --> [],!,
        {Goal}.

   recognise([]) --> [],!.

   % consume one word of the string to parse
   recognise(Phrase) -->
       [Word],
       {
       word(Cat,[Word|Restr]),
       Cat =.. [C|_],
       Phrase =.. [P|_],
       link(C,P)},
       recognise_l(Restr),
       lc(Cat,Phrase).

   % Phrase is an ordinary term
```

```
   % or one slot for the DAG is still a variable
   % so that it unifys

   lc(Phrase,Phrase,String,String) :- !.

   % P1 & P2 are predicates which contain DAGs to be unified
   % in the first position
   lc(P1,P2,String,String) :-
     P1 =.. [Fun,DAG1|Rest],
     P2 =.. [Fun,DAG2|Rest],
     !,
     graph_unify(DAG1,DAG2).

   lc(SubPhrase,SuperPhrase) -->
       {
       rule(Phrase,[SubPhrase|Rest]),
       Phrase =.. [P|_],
       SuperPhrase =.. [S|_],
       link(P,S)},
       recognise_l(Rest),
       lc(Phrase,SuperPhrase).

% how to use the program

   test(String,S) :- recognise(s_eng(S),String,[]).
```

```
%----------------------------------------------------
%
% File: startlc
%
% Stefan M(ue)ller
%
% Large Practical
%
% Version 12.02.93
%
%-----------------------------------------------------

:- ['../sic'].

:- ['../graph',lc,links_lc, rules_lc,templates,
    '../lesen','../examples','../test_preds'].

:- prolog_flag(single_var_warnings,_,off).
:- ['../baum'].
:- prolog_flag(single_var_warnings,_,on).

%drucke_baum(X) :- write(X),nl.
```

B.3 Shift-Reduce Recogniser

```
%------------------------------------------------------------
%
% File: sr
%
% shift reduce interpreter for DCGs
%
% Stefan M(ue)ller
%
% Version 19.02.93
%
%------------------------------------------------------------




   recognise([s_eng(Tree)],Tree,[],[]).

   recognise(Stack,Tree) -->            % reduce
       {
       append_bit(Bit,Rest,Stack),

       rule(LHS,RHS2),
       match(Bit,RHS2)},
       %{write('Reduce: '),write_l([LHS|Rest]),nl},
       recognise([LHS|Rest],Tree).


   recognise(Stack,Tree) -->                % shift
       [Word],
       {
       word(Cat,[Word|Restr]),
       eval(Restr)},
       %{write('Shift : '),write_l([Cat|Stack]),nl},
       recognise([Cat|Stack],Tree).


% how to use the program

   test(String,Tree) :- recognise([],Tree,String,[]).
```

```
% to split a list into to parts of which the first must have
% at least one element

append_one([X|T],Z,[X|T1]) :- append_one(T,Z,T1).
append_one([X],R,[X|R]).

append_bit(X,Y,S) :-
  append_one(A,Y,S),
  reverse(A,X).

% first slot has to be filled by the grammar rules
% the second by the stack

match([],[]) :- !.
match([],[{Restr}|Restrictions]) :- !,
Restr,
        match([],Restrictions).

match([X|R1],[X|R2]) :- !,match(R1,R2).

match([X|R1],[Y|R2]) :-
        X =.. [Fun,DAG1|Rest],
        Y =.. [Fun,DAG2|Rest],
        !,
        graph_unify(DAG1,DAG2), match(R1,R2).


write_l([X]) :- write(X),nl,!.
write_l([X|R]) :- write(X),nl, tab(8), write_l(R).


eval([]) :- !. % no restrictions made
eval([{Restr}]) :- Restr.
```

```
%----------------------------------------------------------
%
% File: startsr
%
% Stefan M(ue)ller
%
% Large Practical
%
% Version 12.02.93
%
%----------------------------------------------------------

:- ['../sic'].

:- [sr,'../graph', rules_lc, templates,
    '../lesen','../examples','../test_preds'].


:- prolog_flag(single_var_warnings,_,off).
:- ['../baum'].
:- prolog_flag(single_var_warnings,_,on).
```

B.4 Chart Parser

B.4.1 The Rule Generator

```
%-----------------------------------------------------
%
% File: make_rules_chart
%
% Stefan M(ue)ller
%
% Large Practical
%
% Version 15.02.93
%
% generates rules for using with the chart parser
%
%-----------------------------------------------------

:- op(1200,xfx,'--->').

:- op(201,yfx,':').
:- op(202,yfx,'<=>').

:- ['../sic'].

% interface for chart-parser

make_rules_chart(Grammar,Lexicon,Rules) :-
        abolish(--->,2),
        consult(Grammar),
        consult(Lexicon),
        tell(Rules),
        foreach( word(Cat,Word,Restriction),
                (write_canonical(word(Cat,Word,Restriction)),
                 write('.'),nl)),
        foreach( rule(LHS,List,Restriction),
              (write_canonical(rule(LHS,List,Restriction)),
               write('.'),nl)),
        told.


word(Cat,Word,Restriction) :-
   (Cat_Tree ---> [Word] ,{ Restriction}),
```

```
   del_last(Cat_Tree,Cat).

word(Cat,Word,no_restr) :-
   (Cat_Tree ---> [Word]),
   del_last(Cat_Tree,Cat).

rule(LHS,[],Restriction) :-
   (LHS_Tree ---> [] ,{ Restriction}),
   del_last(LHS_Tree,LHS).

rule(LHS,[],no_restr) :-
   (LHS_Tree ---> []),
   del_last(LHS_Tree,LHS).

rule(LHS,List,Restriction) :-
   (LHS_Tree ---> RHS),
    RHS \= [],
    RHS \= ([], {_R}),
    RHS \= [_],
    RHS \= ([_], {_R}),
    del_last(LHS_Tree,LHS),
    conj_to_list(RHS,List,Restriction).

conj_to_list((A,B),[A1|List],Restr) :- !,
      del_last(A,A1),
      conj_to_list(B,List,Restr).
conj_to_list({R},[],R) :- !.
conj_to_list(A,[A1],no_restr) :-
      del_last(A,A1).


foreach(X,Y) :-
   X, once(Y), fail.
foreach(_,_).

once(X) :- X, !.

del_last(T1, T2) :-
    T1 =.. [Fun|Args1],
    del_l(Args1,Args2),
    T2 =.. [Fun|Args2].

del_l([_],[]) :- !.
```

```
del_l([X|Y],[X|Y1]) :-
    del_l(Y,Y1).

%:- make_rules_chart(gram_chart,lex_chart,rules_chart).


# shell script by Stefan M"uller
# 10.03.93
# provides grammar interface for the chart parser


cd ~stefanm/lp/chart_rules

cp new-names/grammar grammar
cp new-names/lexicon lexicon

sed s/'-->'/'--->'/ <grammar >grammar.t

cat header grammar.t > grammar.c

rm grammar.t

sed s/'-->'/'--->'/ <lexicon >lexicon.c

sicstus < start_mk_rules

cp rules_chart new-names
cd new-names
```

B.4.2 The Parser

```
%-------------------------------------------------------------
%
% File: chart
%
% Stefan M(ue)ller
%
% Version 19.02.93
%
%-------------------------------------------------------------

% a modified version of the parser of
% Gerald Gazdar & Christopher Mellish

% :- dynamic edge/6.         % uncomment for SICSTUS Prolog

parsec(V0,Vn,String) :-

    % add all epsilon-productions for each position of the chart
    foreach(rule(Category,[],Restr),
        add_edge(V,V,Category,[],['_'],Restr)),

    start_chart(V0,Vn,String).
%
add_edge(V0,V1,Category,Categories,Parse,Restr) :-
    edge(V0,V1,Category,Categories,Parse,Restr),!.

add_edge(V0,V1,Category,Categories,Parse,Restr) :-
    assert_edge(V0,V1,Category,Categories,Parse,Restr),
    foreach(new_edges(edge(V0,V1,Category,Categories,Parse,Restr)),
        true),!.

% inactive edges
new_edges(edge(V1,V2,Category1,[],_Parse,_Restr)) :-

    % add hypotheses, rules which start with found category
    foreach(rule(Category2,[Category1|Categories],Restr1),
        add_edge(V1,V2,Category2,Categories,[V2,Category1,V1],Restr1)),

    % combine inactive edge with active edges
    foreach(edge(V0,V1,Category2,[Category1|Categories],Parses,Restr2),
        add_edge(V0,V2,Category2,Categories,[V2,Category1|Parses],Restr2)).
```

```
% active edges
new_edges(edge(V0,V1,Category1,[Category2|Categories],Parses,Restr)) :-

    % combine inactive edges and active edges
    foreach(edge(V1,V2,Category2,[],_Parse,_Restr),
        add_edge(V0,V2,Category1,Categories,[V2,Category2|Parses],Restr)).


%-------predicates for chart-in-database parsers-----------
%
% start_chart
% uses add_edge to insert inactive edges for the words
% (and their respective categories) into the chart
%

start_chart(V0,V0,[]).
start_chart(V0,Vn,[Word|Words]) :-
    V1 is V0+1,
    foreach(word(Category,Word,Restr),
        add_edge(V0,V1,Category,[],[Word],Restr)),
    start_chart(V1,Vn,Words).

% how to use the program

parse(StartSymbol,String,T) :-
    retractall(edge(_,_,_,_,_,_)),
    parsec(1,Vn,String),
    % at least one parse tree is needed to succsed
    get_tree(StartSymbol,T,Vn).

% for tracing deletes all active edges
clear :-  retractall(edge(_,_,_Cat,[_Needed|_],_Parse,_Restr)).

% assert_edge

assert_edge(V1,V2,Category,Needed,Parse,Restr) :-
        assertz(edge(V1,V2,Category,Needed,Parse,Restr)).

get_tree(StartSymbol,T,ChartLength) :-
        StartSymbol =.. [S,DRS],
        ChartSymbol =.. [S,Unfilled],
```

```
        % use only the largest / latest S entry
        edge(1,ChartLength,ChartSymbol,[],Parts,Restr),

        Unfilled:in <=> DRS:in,

        get_tree(1,ChartLength,Parts,Tree),

        !,
        Restr,
        DRS:out      <=> Unfilled:out,
        DRS:question <=> Unfilled:question,
        DRS:restr    <=> Unfilled:restr,
        T =.. [S|Tree].

get_tree(V0,V1,[V1,Cat,V0],[Tree]) :-
        edge(V0,V1,Cat,[],Parts,Restr),
        get_tree(V0,V1,Parts,Tree1),
        Restr,
        Cat  =.. [Catn|_],
        Tree =.. [Catn|Tree1].

get_tree(_,_,[Term],[Term]).         % terminals

get_tree(V0,V2,Parts,Tree) :-
        get_parts(Parts,RestParts,V0,Cat,V1),
        get_tree(V0,V1,[V1,Cat,V0],Tree2),
        get_tree(V1,V2,RestParts,Tree1),
        xappend(Tree2,Tree1,Tree).



get_parts([V2,Cat2,V1,Cat,V0],[V2,Cat2,V1],V0,Cat,V1) :- !.
get_parts([Vx,Catx|Parts],[Vx,Catx|PartsRest],V0,Cat,V1) :-
         get_parts(Parts,PartsRest,V0,Cat,V1).


foreach(X,Y) :-
   X, once(Y), fail.
foreach(_,_).

%once(X) :- X, !.
```

```
no_restr. % no restrictions made
```

```
%------------------------------------------------------------
% File: startchart
%
% Stefan M(ue)ller
%
% Large Practical
%
% Version 22.03.93
%
%------------------------------------------------------------

:- [quint, %sic,
    graph,
    chart,                          % the parser
    test_preds,
    operators,
    prepositions,
    rules_chart,                    % lexicon + grammar
    syntactic_templates,
    semantic_templates,
    discourse,lesen,output,
    dexamples,
    examples].

:- prolog_flag(single_var_warnings,_,off).
:- [baum].
:- prolog_flag(single_var_warnings,_,on).


%drucke_baum(T) :- write(T),nl.

%test(S,Tree) :- s_eng(Tree,S,[]).

in_mode(chart).
```

B.5 The Compiler for ϵ-free Grammars

```
%-----------------------------------------------------
%
% File: make_rules_lc_list
%
% Stefan M(ue)ller
%
% Large Practical
%
% Version 19.02.93
%
% for lc and sr
% generates rules without e-productions in list fomrat
% rule([s,G0,G],[[np,G0,G1],[vp,G1,G]]).
%
%-----------------------------------------------------

:- op(1200,xfx,'--->').

:- op(33,yfx,':').
:- op(34,yfx,'<=>').

:- ['../sic'].

make_rules_lc(Grammar,Lexicon,Rules) :-
consult(Grammar),
consult(Lexicon),
tell(Rules),
foreach( word(Cat,Word),
(
 Cat =.. [Fun|Arg],
 write_canonical(word([Fun|Arg],Word)),
 write('.'),nl)),
retractall(--->(_,[_Word])),
retractall(--->(_,[_Word],{_Restr})),
foreach( rule(LHS,List),
              (transform(rule(LHS,List),T),
       write_canonical(T),
       write('.'),nl)),
told,
statistics(runtime,[_,T]),
write('time = '),write(T), nl.
```

```
transform(rule(RHS,LHS),rule([Fun|Args],T)) :-
  RHS =.. [Fun|Args],
  transform(LHS,T).

transform([],[]) :- !.
transform([{X}|R],[{X}|T]) :- !,transform(R,T).
transform([X|Rest],[[Fun|Args]|T]) :-
  X =.. [Fun|Args],
  transform(Rest,T).

word(Cat,[Word,Restriction]) :-
   (Cat ---> [Word], Restriction).

word(Cat,[Word]) :-
   (Cat ---> [Word]).

rule(LHS,List) :-
 (LHS ---> RHS),
    RHS \= [],
    RHS \= ([],_Restr),
  conj_to_list(RHS,L),
  process(L,List).

conj_to_list((A,B),[A|List]) :- !,
        conj_to_list(B,List).
conj_to_list(A,[A]).

% selects all combinations of elements of the input list
% and tests whether they are reducable to epsilon

process(Lin,Lout) :-
 select(Part,Lin,List),
 List \= [],
 List \= [{_Restr}],
 gaps(Part,Restr),
 xappend(List,Restr,Lout).

process(List,List).

% similar to delete but backtrackable for different solutions
```

```
rub(F,[F|X],X).
rub(F,[E|X],[E|Y]) :- rub(F,X,Y).

% selects elements out of L the rest remains in R

select([X],L,R) :- rub(X,L,R).
select([X|Other],[X|L],R) :- select(Other,L,R).

gaps([],[]).

gaps([L|R],Restrictions) :-
 gap_test(L,Restr),
 gaps(R,Rest),
 melt(Restr,Rest,Restrictions).

gaps([H|T],Restrictions) :-
 (H ---> RHS),
 conj_to_list_r(RHS,List,Restr),
 gaps(List,RestrList1),
 melt(Restr,RestrList1,RestrList2),
 gaps(T,RestrList3),
 xappend(RestrList2,RestrList3,Restrictions).

gap_test(Gap,[]) :-
 (Gap ---> []).

gap_test(Gap,Restr) :-
 (Gap ---> [],Restr).

conj_to_list_r((A,B),[A|List],Restr) :- !,
      conj_to_list_r(B,List,Restr).
conj_to_list_r({R},[],{R}) :- !.
conj_to_list_r(A,[A],[]).

foreach(X,Y) :-
   X, once(Y), fail.
foreach(_,_).

once(X) :- X, !.

melt([],X,X) :- !.
melt(X,Y,[X|Y]).
```

```
:- make_rules_lc(gram_lc,lex_lc,rules_lc_list).
:- halt.
```

C Semantic

C.1 Semantic Templates

```
%---------------------------------------------------
%
% File: semantic_templates
%
% Stefan M"uller
%
% Version 08.03.93
%
%---------------------------------------------------

%---------------------------------------------------
% adjectives

template(adjective(Description),ADJ) :-
  Pred =.. [Description,X],
  ADJ:in:gl_DRS    <=> ADJ:out:gl_DRS,
  ADJ:in:scopeList <=> ADJ:out:scopeList,
  ADJ:in:focus     <=> ADJ:out:focus,
  ADJ:in:npConstr  <=> NPConstr,
  ADJ:out:npConstr <=> [Pred|NPConstr],
  ADJ:in:drs       <=> [Dom,Constr],
  ADJ:out:drs      <=> [Dom,[Pred|Constr]],
  ADJ:in:locDRS    <=> [LocDom,LocConstr],
  ADJ:out:locDRS   <=> [LocDom,[Pred|LocConstr]],
  ADJ:binding      <=> X^Pred.

%---------------------------------------------------
% nouns

template(noun(Description,Gender),NOUN) :-
  Pred =.. [Description,X],
  Gen  =.. [Gender,X],
  NOUN:in:gl_DRS   <=> NOUN:out:gl_DRS,
  NOUN:in:scopeList<=> NOUN:out:scopeList,
  NOUN:in:focus    <=> Focus,
  NOUN:out:focus   <=> [X|Focus],
  NOUN:in:drs      <=> [Dom,Constr],
  ( % article indefinite -> new object is introduced
      NOUN:article <=> indefinite,
```

```
      % add information
      NOUN:out:drs     <=> [[X|Dom],[Pred,Gen|Constr]],
      NOUN:in:locDRS   <=> [LocDom,LocConstr],
      NOUN:out:locDRS  <=> [[X|LocDom],[Pred,Gen|LocConstr]],
      NOUN:in:npConstr <=> NPConstr,
      NOUN:out:npConstr<=> [Pred,Gen|NPConstr],
      NOUN:restr       <=> []
  ;
    % article definite -> refer to an exsisting object
      NOUN:article <=> definite,
      % check information
      NOUN:out:drs    <=> NOUN:in:drs,
      NOUN:out:locDRS <=> NOUN:in:locDRS,
      NOUN:restr     <=> [check(X,[Pred,Gen],Focus,Constr)]
  ),
  NOUN:binding    <=> X^Pred.

%-----------------------------------------------------
% prepositions

template(prep(Description),P) :-
  Prep =.. [Description,X,Y],
  P:in:gl_DRS    <=> P:out:gl_DRS,
  P:in:focus     <=> P:out:focus,
  P:in:scopeList <=> P:out:scopeList,
  P:in:npConstr  <=> NPConstr,
  P:out:npConstr <=> [Prep|NPConstr],
  P:in:drs       <=> [Dom,Constr],
  P:out:drs      <=> [Dom,[Prep|Constr]],
  P:in:locDRS    <=> [LocDom,LocConstr],
  P:out:locDRS   <=> [LocDom,[Prep|LocConstr]],
  ( % article indefinite -> new object is introduced
      P:article <=> indefinite,
      % add information
      P:restr <=> []
  ;
    % article definite -> refer to an exsisting object
      P:article <=> definite,
      % check information
      P:check   <=> Check,
      PrepCheck =.. [Description,Check,Y],
      P:restr   <=> [check_prep(PrepCheck,Constr)]
  ),
```

```
  P:binding    <=> Y^X^Prep.

%--------------------------------------------------
% proper nouns

template(pn(Name,Gender),PN) :-
  Gen =.. [Gender,X],
  PN:in:gl_DRS   <=> Gl_DRSIn,
  PN:out:gl_DRS  <=> Gl_DRSOut,
  PN:in:locDRS   <=> PN:out:locDRS,
  PN:in:drs      <=> PN:out:drs,
  PN:in:scopeList<=> PN:out:scopeList,
  PN:in:npConstr <=> PN:out:npConstr,
  PN:in:focus    <=> Focus,
  PN:out:focus   <=> [X|Focus],
  PN:locListIn   <=> LocListIn,
  PN:locListOut  <=> [X|LocListIn],
  PN:binding     <=> X,
  PN:restr       <=> [check_add(X,[named(X,Name),Gen],
     Gl_DRSIn,Gl_DRSOut)].

%--------------------------------------------------
% pronouns

template(pronoun(Gender),PN) :-
  Gen  =.. [Gender,X],
  PN:in:gl_DRS  <=> PN:out:gl_DRS,
  PN:in:drs     <=> PN:out:drs,
  PN:in:locDRS  <=> PN:out:locDRS,
  PN:in:scopeList <=> PN:out:scopeList,
  PN:in:npConstr  <=> PN:out:npConstr,
  PN:in:gl_DRS  <=> [_Gl_Dom,Gl_Constr],
  PN:in:drs     <=> [_Dom,Constr],
  PN:in:locDRS  <=> [_LocDom,LocConstr],
  PN:in:focus   <=> Focus,
  PN:out:focus  <=> [X|Focus],
  PN:locListIn  <=> LocList,
  PN:locListOut <=> [X|LocList], % reintroduced
  PN:binding    <=> X,
  PN:restr      <=> [ member(X,Focus),
                      \+ strict_member(X,LocList),
      ( strict_member(Gen,LocConstr)
      ; strict_member(Gen,Constr)
```

```
      ; strict_member(Gen,Gl_Constr)
      )
                    ].

template(refl_pronoun(Gender),PN) :-
  Gen  =.. [Gender,X],
  PN:in:gl_DRS  <=> PN:out:gl_DRS,
  PN:in:drs     <=> PN:out:drs,
  PN:in:locDRS  <=> PN:out:locDRS,
  PN:in:scopeList<=> PN:out:scopeList,
  PN:in:npConstr <=> PN:out:npConstr,
  PN:in:focus   <=> PN:out:focus,  % instance is still in focus
  PN:in:gl_DRS  <=> Gl_DRS,
  PN:in:drs     <=> DRS,
  PN:in:locDRS  <=> LocDRS,
  PN:locListIn <=> LocList,
  PN:locListOut <=> LocList,
  PN:binding    <=> X,
  (LocList = [X|_]; LocList = [_,X]),
  PN:restr      <=> [ (strict_check(X,[Gen],LocDRS)
                      ;
                      strict_check(X,[Gen],DRS)
                      ;
                      strict_check(X,[Gen],Gl_DRS))
    ].

% there is need for a plural feature

%----------------------------------------------------------
% verbs

%ditransitive

template(ditrans_verb(Description),TV) :-
  Pred =.. [Description,X,Y,Z],
  TV:in:gl_DRS <=> TV:out:gl_DRS,
  TV:in:focus  <=> TV:out:focus,
  TV:in:scopeList <=> TV:out:scopeList,
  TV:in:npConstr  <=> NPConstr,
  TV:out:npConstr <=> [Pred|NPConstr],
  TV:in:drs    <=> [Dom,Constr],
  TV:out:drs   <=> [Dom,[Pred|Constr]],
  TV:in:locDRS <=> [LocDom,LocConstr],
```

```
  TV:out:locDRS <=> [LocDom,[Pred|LocConstr]],
  TV:binding  <=> Z^Y^X^Pred.

%transitive

template(trans_verb(Description),TV) :-
  Pred =.. [Description,X,Y],
  TV:in:gl_DRS <=> TV:out:gl_DRS,
  TV:in:focus  <=> TV:out:focus,
  TV:in:scopeList <=> TV:out:scopeList,
  TV:in:npConstr  <=> NPConstr,
  TV:out:npConstr <=> [Pred|NPConstr],
  TV:in:drs    <=> [Dom,Constr],
  TV:out:drs   <=> [Dom,[Pred|Constr]],
  TV:in:locDRS <=> [LocDom,LocConstr],
  TV:out:locDRS <=> [LocDom,[Pred|LocConstr]],
  TV:binding  <=> Y^X^Pred.

% intransitive

template(intr_verb(Description),IV) :-
  Pred =.. [Description,X],
  IV:in:gl_DRS <=> IV:out:gl_DRS,
  IV:in:focus  <=> IV:out:focus,
  IV:in:scopeList <=> IV:out:scopeList,
  IV:in:npConstr  <=> NPConstr,
  IV:out:npConstr <=> [Pred|NPConstr],
  IV:in:drs    <=> [Dom,Constr],
  IV:out:drs   <=> [Dom,[Pred|Constr]],
  IV:in:locDRS <=> [LocDom,LocConstr],
  IV:out:locDRS <=> [LocDom,[Pred|LocConstr]],
  IV:binding  <=> X^Pred.
```

C.2 Scoping, Question Answering, Discourse

```
%------------------------------------------------------------
%
% File: discourse
%
% Large Practical
%
% Stefan M"uller
%
% Version 09.03.93
%
%------------------------------------------------------------


drs_sent(Sent,DiscIn,DiscOut) :-

        (
            (last(Sent,'.'),
             once(parse(s_eng(DiscIn),Sent,Tree)),

             DiscIn:restr <=> Restr,
             eval(Restr),

             react(Tree)
            ;
             last(Sent,?),
             once(parse(question(DiscIn),Sent,Tree)),
             DiscIn:restr <=> Restr,
             eval(Restr),

             react(Tree),

             answer(DiscIn)
            ),
            DiscIn:out:drs <=> [Dom,InConstr],
            clear(InConstr,OutConstr), % get rid of multiple entries
            DiscIn:out:focus <=> InFocus,
            clear(InFocus,OutFocus),
            DiscOut:in:gl_DRS <=> DiscIn:out:gl_DRS,
            DiscOut:in:drs    <=> [Dom,OutConstr],
            DiscOut:in:focus  <=> OutFocus,
            output(DiscOut)
```

```
        ;
            nl,write_out("I don't understand!"),nl,nl,
            % nothing has changed
            DiscOut:in <=> DiscIn:in
        ),
        !.


test_disc(Disc) :-
        A:in:gl_DRS <=> [[],[]],
        A:in:drs    <=> [[],[]],
        A:in:focus  <=> [],
        retractall(counter(_)),
        asserta(counter(0)),
        drs(Disc,A).

drs([],S) :-
   S:out <=> S:in.

drs([Sent|Disc],S1) :-
        write_out(Sent),nl,nl,
        convert_s(Sent,SentL),
        drs_sent(SentL,S1,S2),
        drs(Disc,S2).


%-----------------------------------------------
% returns a list without multiple entries

% the list has to be reversed because the
% order of the focus list must not be destroied
% so if there are two entries of the same instnace
% the most recent will survive

clear(X,Z) :- reverse(X,Y),
      clear_(Y,U),
      reverse(U,Z).

clear_([],[]) :- !.

clear_([Constr|R],Cleared) :-
        strict_member(Constr,R),
        clear_(R,Cleared).
```

```
clear_([Constr|R],[Constr|Cleared]) :-
        clear_(R,Cleared).

%-----------------------------------------------


answer(Q) :-
        Q:question <=> [X,QConstr],
        var(X),

        (restr(on) ->
         write('Question: '), write((X,QConstr)),nl
        ;true),

        Q:in:drs    <=> [_Dom,Constr],
        Q:in:gl_DRS <=> [_Vars,Global_names],
        setof(X,
                solution(X,QConstr,Constr),
                Solutions),
        nl,
        print_solution(Solutions,Global_names,Constr).

answer(Q) :-
       Q:question <=> [X,_QConstr],
       number(X),   % solution is found already by instantiating variables
       Q:in:drs    <=> [_Dom,Constr],
       Q:in:gl_DRS <=> [_Vars,Global_names],
       print_solution([X],Global_names,Constr).


answer(Q) :-
        Q:question <=> [yes,QConstr],
        Q:in:drs   <=> [_Dom,Constr],

        (restr(on) ->
         write('Question: '), write((yes^QConstr)),nl
        ;true),

        solution(_X,QConstr,Constr),
        nl,write_out("Yes, that's right."),nl,nl.
```

```
answer(Q) :-
        Q:question <=> [X,_QConstr],
        nonvar(X),
        nl,write_out("As far as I know, no."),nl,nl.

answer(_) :-  nl,write_out("I don't know."),nl,nl.


:- dynamic fresh/1.

fresh(Constr,FreshConstr) :-
         asserta(fresh(Constr)),
         retract(fresh(FreshConstr)).

solution(X,QConstr,Constr) :-
         fresh(Constr,FreshConstr),
         solution_(X,QConstr,FreshConstr).

% all checked
solution_(_,[],_) :- !.

% if entries exist, they have to be instanciated
% in the same way
% case 2.b. the solution follows from a
% general statement we have to check the
% 'some' expression of the question first
% and then the antecedens
solution_(X,[some(Y,Cond)|QConstr],Constr) :-
        find_term(some,Constr,some(Y,Scope)),
        members(Cond,Scope),
        solution_(X,QConstr,Constr).

% case 2.a. one particular instance is meant
solution_(X,[some(_X,Cond)|QConstr],Constr) :-
        xappend(Cond,QConstr,NQConstr),
        solution_(X,NQConstr,Constr).

% case 3
% does every woman love John?
% does john love every woman?
solution_(S,[QTerm|QConstr],Constr) :-
        QTerm =.. [Quantifier,X,ScopeQ],
quantifier(Quantifier),
```

```
find_term(Quantifier,Constr,QTermC),
        QTermC =.. [Quantifier,X,ScopeC],
members(ScopeC,ScopeQ),
xappend(ScopeC,Constr,TConstr),
solution_(S,QConstr,TConstr).

quantifier(every).
quantifier(most).



% cases 1
solution_(X,[Pred|QConstr],Constr) :-
        Pred =.. [Fun1|Args1],
        Fun1 \= some,
        find_term(Fun1,Constr,Answer),
        Answer =.. [Fun1|Args2],
        check_lists(Args1,Args2,Constr),
        solution_(X,QConstr,Constr).

check_lists([],[],_Constr) :- !.

check_lists([Vq|R1],[Vd|R2],Constr) :-
        (
            % Vq is either instanciated
            % then it refers to a global instance
            % or we are asking for it
            number(Vd),!,Vq=Vd
        ;
            % Vd is a term we are asking for
            % like in relative clauses
            % What did mary say?
            var(Vq),nonvar(Vd),!,
            Vq = Vd
        ;
            % Vd is a quantified variable
            % and we have to check the
            % antecedens
            var(Vd),
            !,
            check_antecedens(Vq,Vd,Constr)
        ),
        check_lists(R1,R2,Constr).
```

```
check_lists([TermQ|R1],[TermD|R2],Constr) :-
        TermQ =.. [Fun|ArgsQ],
        TermD =.. [Fun|ArgsD],
        check_lists(ArgsQ,ArgsD,Constr),
        check_lists(R1,R2,Constr).

check_antecedens(Vq,Vd,Constr) :-
        member(QTerm,Constr),
QTerm =.. [Quantifier,Vd,Cond],
quantifier(Quantifier),
( var(Vd) ->
          sub(Vq,Vd,Cond,TCond)
        ;
          Vq = Vd,Cond = TCond
        ),
        members(TCond,Constr).


%--------------------------------------------------
% predicates for various anaphora checks
%--------------------------------------------------

check(X,Cs,Focus,Constr) :-
(var(X),!,member(X,Focus)
;
 % X is initialized by adjective check
         strict_member(X,Focus)),
        strict_members(Cs,Constr).

%--------------------------------------------------
% for proper nouns

check_add(X,Cs,[Dom,Constr],[Dom,Constr]) :-
member(X,Dom),
        strict_members(Cs,Constr),!.

check_add(X,Cs,[Dom,Constr],[[X|Dom],NewConstr]) :-
        append(Cs,Constr,NewConstr).

% for prepositions

check_prep(P,Constr) :-
```

```
        strict_member(P,Constr).

strict_check(X,Cs,[Dom,Constr]) :-
        strict_member(X,Dom),
        strict_members(Cs,Constr).

/*
strict_member(X,[Y|_]) :-
        X == Y.
strict_member(X,[_|L]) :-
        strict_member(X,L).
*/

strict_members([],_).
strict_members([X|L],L1) :-
        strict_member(X,L1),
        strict_members(L,L1).


members([],_) :- !.
members([X|Rest],Constr) :-
        member(X,Constr),
        members(Rest,Constr).

%--------------------------------------------------
% postpone the evaluation of
% prepositional checks


find_later([],[],[]) :- !.

find_later([check_prep(P,Constr)|R1],Now,[check_prep(P,Constr)|R2]) :-
     (restr(on) -> write('Postponed: '),
            write(check_prep(P,Constr)),nl
            ;true),
      find_later(R1,Now,R2),!.

find_later([Restr|R1],[Restr|Now],R2) :-
      find_later(R1,Now,R2).


eval([]) :- !.
eval([Restr|Rest]) :-
```

```
       (restr(on) -> write('R: '),write(Restr),nl;true),
       Restr,
       (restr(on) -> write(ok),nl;true),
       eval(Rest).

%-------------------------------------------------
% to join the binding of verbs
% conected with coordinators

join(X^Y^Z^P1,X^Y^Z^P2,X^Y^Z^(P1,P2)) :- !.
join(X^Y^P1,X^Y^P2,X^Y^(P1,P2)) :- !.
join(X^P1,X^P2,X^(P1,P2)) :- !.


%--------------------------------------------------
% binds all unbound variables in a list
% stops if one variable which is bound already
% turns up

:- dynamic counter/1.

bind([]) :- !.
bind([V|Rest]) :- var(V),!,retract(counter(Y)),
           V is Y + 1,
           asserta(counter(V)),
           bind(Rest).

bind(_L) :- !. % L = [V|Rest],nonvar(V)
               % all variables in rest are bound
               % already

:- dynamic answer/2.


bind_prepositions(Cond,GlDom,GlConstr,Cin,Cout,Later,LaterRest) :-
    ( bind_actor(Cond,GlDom,Cin,C1,Later,L1)
    ; C1=Cin,L1 = Later
    ),
    ( bind_patient(Cond,GlDom,C1,C2,L1,L2)
    ;
      C2=C1,L2 = L1
    ),
    bind_ask(C2,C2,Cout,Cin,GlConstr,L2,LaterRest),
```

```
    asserta(bound(Cout,LaterRest)).


bind_actor(Cond,_GlDom,Cin,Cout,Lin,Lout) :-
    member(by(V,X),Cin),
%   var(V),
%   strict_member(X,GlDom),  % this would be sure
    Cond =.. [_Fun,X|_],
    del(by(V,X),Cin,Cout),
    (del(prep_check(by(V,X),_),Lin,Lout),!
    ; true).

bind_patient(Cond,_GlDom,Cin,Cout,Lin,Lout) :-
    member(to(V,Y),Cin),
%   var(V),
%   strict_member(Y,GlDom),
    Cond =.. [_Fun,_X,Y,_Z],
    del(to(V,Y),Cin,Cout),
    (del(prep_check(to(V,Y),_),Lin,Lout),!
    ; true).

bind_ask([Prep|_],C,C,_,_,L,L) :-
    Prep =.. [P,X,_],
    preposition(P),
    nonvar(X),!.      % found a prep already bound
                      % -> the following will be bound too

bind_ask([Prep|Rest],Cin,Cout,Constr,GlConstr,Later,LaterRest) :-
    Prep =.. [P,X,_],
    preposition(P),
    ask(P,Answer),
    (find_number_for_name(Answer,GlConstr,X);
     find_term(Answer,Constr,X)
     % ask only if no term found
    ;
     \+ find_term(Answer,Constr,_),
     retract(Prep,Answer),
     write('What?'),nl,fail),

    % we don't need the anaphora check if the preposition
    % modifies a verb
    (verb(Answer) ->
       (del(check_prep(Prep,_TConstr),Later,L1),!     % ???? !!!
```

```
    % this might cause
    % problems
    % if there are more then
    % one entries
    % of the same prep
        ;  Later = L1     % delete it if it is there
),
C2 = Cin

    % if it modifies a noun
    % we have to remove the doubled entry
    ; ( member(check_prep(Prep,_),Later) -> del(Prep,Cin,C2)
       ;
      C2 = Cin),
      L1 = Later
    ),
    bind_ask(Rest,C2,Cout,Constr,GlConstr,L1,LaterRest).

bind_ask([_|Rest],Cin,Cout,Constr,GlConstr,Later,LaterRest) :-
    bind_ask(Rest,Cin,Cout,Constr,GlConstr,Later,LaterRest).

bind_ask([],C,C,_,_,L,L) :- !.

% ask only once

ask(Prep,Answer) :-
    retract(answer(Prep,Answer)),
    asserta(answer(Prep,Answer)), !.

ask(Prep,Answer) :-
    nl,
    write_out("Please type in the infinitiv of the verb or the noun to which '"),
    write(Prep),
    write_out("' is related: "),
    lies_satz([Answer,'.']),
    asserta(answer(Prep,Answer)).

% generates new answers, fails never
ask(Prep,Answer) :-
    ask(Prep,Answer).

verb(Verb) :-
    in_mode(dcg),!,
```

```
    v(_DAG,_Cat,_Tree,[Verb|X],X).

verb(Verb) :-
    word(Cat,[Verb],_Restr),
    Cat =.. [v|_].

find_number_for_name(Person,GlConstr,Answer) :-
    name(Person,[X|Y]),
    U is X - 32,
    name(Name,[U|Y]),
    find_number(Name,GlConstr,Answer),!.

% finds a term which has the funtor name
% FunctorName
% backtracking provides the next term
% with this functor name

find_term(_FunctorName,[],no_answer) :-
    !,fail.
find_term(FunctorName,[Answer|_],Answer) :-
    Answer =.. [FunctorName|_].
find_term(FunctorName,[_|T],Answer) :-
    find_term(FunctorName,T,Answer).

find_number(Name,[named(No,Name1)|_],No) :- !,Name == Name1.

find_number(Name,[_|R],No) :-
     find_number(Name,R,No).



% searches any occurance of X in a list
% finds even such which are hidden in terms

dig_args(X,[H|T]) :-
            nonvar(H),
            H =.. [_Fun|Args],
            dig_args(X,Args);
            dig_args(X,T).

dig_args(X,[Y|_]) :- X == Y.
dig_args(X,[_|T]) :- dig_args(X,T).
```

```
%--------------------------------------------
% quantifier scoping

scope([],S,S,D,D,C,C) :- !.

scope([some(X,Y) & Description|Scopes],Sin,Sout,DomIn,DomOut,Cin,Cout) :-
    ask_scope(some(X,Y) & Description,Sin,S1,DomIn,D1,Cin,C1),
    scope(Scopes,S1,Sout,D1,DomOut,C1,Cout).

scope([QTerm|Scopes],Sin,[QTerm|Sout],
       DomIn,DomOut,Cin,[plural(Var)|Cout]) :-
    QTerm =.. [_Quantifier,Var,Scope],

    % use the scoped QTerm in stead of the
    % unscoped entries in Cin
    % Cin is a part of a local constraint
    % which contains all information (unscoped)

    del_list(Scope,Cin,C1),
    scope(Scopes,Sin,Sout,DomIn,DomOut,C1,Cout).

ask_scope(some(X,Y) & Description,Sin,Sout,DomIn,DomOut,Cin,Cout) :-
    Description =..[Desc,_],
    write('Is there only one '),
    write(Desc),
    write(' ? (y/N): '),
    get0(C),
    (C = 10 -> true;get0(_)),
    (C =121 -> Cin = Cout,  DomOut = [X|DomIn],
       Sin = Sout
           ;  del_list(Y,Cin,Cout),
       Sout = [some(X,Y)|Sin],
              DomOut = DomIn).


scope_all([],C,C) :- !.

scope_all([QTerm|Scopes],Cin,[QTerm|Cout]) :-
    QTerm =.. [_Quantifier,_Var,Scope],
    del_list(Scope,Cin,C1),
    scope_all(Scopes,C1,Cout).
```

```
scope_all([some(X,Scope) & _Descr|Scopes],Cin,
          [some(X,Scope)|Cout]) :-
    del_list(Scope,Cin,C1),
    scope_all(Scopes,C1,Cout).

del_list([],L,L) :- !.
del_list([H|T],Lin,Lout) :-
    strict_del(H,Lin,L1),
    del_list(T,L1,Lout).

/*
strict_del(X,[Y|R],R) :- X == Y,!.
strict_del(X,[H|R],[H|R1]) :-
    strict_del(X,R,R1).

once(X) :- X,!.
*/

%-----------------------------------------------
% Authors:  Guy Barry and David Beaver

% sub(Val,Var,InTerm,OutTerm) :-
%       OutTerm is a copy of InTerm where the variable Var has been
%       replaced by the term or variable Val and all other variables
%       remain unchanged (given that Var does not occur in Val).
sub(Val,Var,InTerm,OutTerm) :-
        var(Var),
        bagof(  Term,
                Var^(Var=Val,Term=InTerm),
                [OutTerm] ).
```

C.3 Output

```
%------------------------------------------------------------
%
% File: output
%
% Large Practical
%
```

```
% Stefan M"uller
%
% Version 06.3.93
%
%-------------------------------------------------------------


output(A) :-
          drs(on),
          A:in:gl_DRS <=> [Gl_Dom,Gl_Constr],
          A:in:drs    <=> [Dom,Constr],
          A:in:focus  <=> Focus,

  fresh((Focus,Constr),(FreshFocus,FreshConstr)),
  bind_vars(FreshConstr),

          write('Focus: '),      write(FreshFocus), nl,nl,
          write('global DRS: '), write(Gl_Dom),
                write_l(7,Gl_Constr),nl,
          write('       DRS: '), write(Dom),
                write_drs(FreshConstr),nl,nl,!.

output(_).

bind_vars(Constr) :- bind_vars(Constr,90,_).

bind_vars([],C,C) :- !.

bind_vars([H|T],CounterI,CounterO) :-
   (
      var(H),
      !,
      name(H,[CounterI]),
      CounterT is CounterI - 1
           ;
              H =.. [_Fun|Args],
      bind_vars(Args,CounterI,CounterT)
   ),
   bind_vars(T,CounterT,CounterO).



/* a pretty printer for oldfashioned DRSs
```

```
write_drs([X|_]) :- var(X),write(X). % error
write_drs([HiddenDRS|Rest]) :-
          HiddenDRS =.. [Quantifier,[AntDom,AntConstr],
                                    [ConDom,ConConstr]],
          !,
          nl,tab(12),
          write(AntDom),tab(10),
          write(Quantifier),tab(5),write(ConDom),
          length(AntDom,LengthA),
          name(Quantifier,Q),
          length(Q,LengthQ),
          Distance is LengthQ - LengthA + 9 + 10,
          write_pairs(12,Distance,AntConstr,ConConstr),
          write_drs(Rest).

write_pairs([],[]).
write_pairs(In,_Distance,X,[]) :- write_l(In,X).
write_pairs(In,Distance,[],Y) :- Tabs is In + Distance, write_l(Tabs,Y).
write_pairs(In,Distance,[X|R1],[Y|R2]) :- nl,tab(In),
          write(X),
          term_length(X,Length),
          Tabs is Distance - Length,
          tab(Tabs),
          write(Y),
          write_pairs(In,Distance,R1,R2).

term_length(X,4) :- var(X). % just a guess
term_length(X,L) :-
        X =.. [Fun,Arg1|Args],
        !,
        bit_length(Args,L1),
        term_length(Arg1,L2),
        term_length(Fun,L3),
        L is L1 + L2 +L3 +2.

term_length(X,L) :-
        atomic(X),
        name(X,List),
        length(List,L).

bit_length([H|T],L) :-
        term_length(H,L1),
```

```
        bit_length(T,L2),
        L is L1 + L2 + 1. % 1 = comma in terms

bit_length([],0).

*/

write_drs([X|Rest]) :- nl,tab(7),write(X),write_drs(Rest).
write_drs([]) :- !.

write_l(_,[]) :- nl,!.
write_l(Tabs,[H|T]) :- nl, tab(Tabs),write(H),write_l(Tabs,T).

write_line([]) :- nl,!.
write_line([H|T]) :- write(H),tab(3),write_line(T).

conj_to_list((A,B),[A|List]) :- !,
        conj_to_list(B,List).
conj_to_list(A,[A]).




%---------------------------------------------------
% print_solution is used to print
% answers to questions


print_solution([],_,_) :- nl,nl,!.

print_solution([X|Rest],Global_names,Constr) :-
        (    var(X),
     nl,write('This is not specified!'),nl,
     write('May be you made a general statement about it.'),nl,
     nl
          ;
     find_name(X,Global_names,Name),
             write(Name),tab(7),
     (  find_infos(X,Constr,Infos),
        write_line(Infos)
     ;
        true),
     nl
```

```
         ;
             find_infos(X,Constr,Infos),
             write_line(Infos)
%         ;
%              \+ number(X),write(X)
        ),
        print_solution(Rest,Global_names,Constr).

find_name(X,[named(X,Name)|_],Name) :- !.
find_name(X,[_|Rest],Name) :- find_name(X,Rest,Name).

% find at least one info

find_infos(X,Constr,[H|T]) :- find_info(X,Constr,[H|T]).

find_info(_,[],[]) :- !.

find_info(X,[Info|T],[Info|Infos]) :-
          Info =.. [_Fun|Args],
          dig_args(X,Args),
          !,
          find_info(X,T,Infos).

find_info(X,[_|T],Infos) :-
         find_info(X,T,Infos).
```

C.4 Discourse Examples

```
%---------------------------------------------------------
%
% File: dexamples
%
% Large Practical
%
% Stefan M"uller
%
% Version 15.03.93
%
% This file contains some discourse examples
% discussed in the documentation.
%
%---------------------------------------------------------
```

```
% testdisc calls examples which don't interact
% with the user

testdisc :- restr_off, drs_off,
    tdisc(1), tdisc(2), tdisc(3),
            tdisc(4),
            tdisc(5),
            tdisc(8), tdisc(12),
    tdisc(16),tdisc(17),tdisc(25),
    tdisc(29),
     statistics(runtime,[T,_]),
     write('time = '),write(T), nl.

disc(1,[
 "A man who likes a donkey which likes Mary likes John."
       ]).

disc(2,[
 "John is a man.",
 "He is fat.",
 "Mary likes the fat man.",
 "Who does Mary like?"
       ]).

% nested phrases
disc(3,[
 "John expected Mary to read a book.",
 "The book is large.",
 "Did Mary read the book?"
       ]).

%   nested phrases:     expect(j,m,want(m,b,like(b,b)))
% + reflexive pronoun

disc(4,[
 "John expected Mary to want Bill to like himself.",
 "Did John expect Mary to want Bill to like himself?"
       ]).

%   nested phrases: say(j,like(j,j))
% + reflexive pronoun
```

```
disc(5,[
 "John said he likes himself."
       ]).

disc(7,[
 "Every man likes a book which is stupid."
       ]).

%   questions

disc(8,[
 "Mary is afraid of John.",
 "Who is afraid of John?",
 "Who is Mary afraid of?"
       ]).

%   subjective predicative
% + reflexive pronouns

disc(9,[
 "John, who is proud of himself, told Mary that he won."
       ]).

%   quantifier
% + questions
% + focus in questions

disc(10,[
 "Every man is afraid of a dog.",   % yes or no
 "John is a man.",
 "Is John afraid of a dog?",
 "Bill is a man.",
 "Is Bill afraid of a dog?",
 "Is Bill afraid of the dog?",     % if you typed 'no' above
   % the program will not
   % understand the question
   % because there is no
   % single dog.
 "Mary is a woman.",
 "Is Mary afraid of a dog?",
 "Mary is afraid of a fat dog.",
 "John is afraid of another dog which sleeps.",
 "Is Mary afraid of a dog?",       % focus
```

```
 "Is this dog fat?",               % yes
 "Which dogs is John afraid of?"
         ]).

/* doesn't work

%   prepositional phrases
% + definite articles
% + nested phrases

disc(11,
["John lives in a garden.",
 "A man in the garden likes Mary.",
 "Another man likes Bill.",
 "This man is very fat.",
 "John told Mary that the fat man has said that the man in the garden likes her."]).
*/

%   prepositional phrases
% + definite articles
% + nested phrases
% like 11 but without interaction

disc(12,
["John lives in a garden.",
 "A stupid man likes Mary.",
 "Another man likes Bill.",
 "This man is very fat.",
 "John told Mary that the fat man has said that the stupid man likes her."]).

/*
Focus: [3,4,6,1,5,2]

global DRS: [5,3,1]
       named(5,Bill)
       male(5)
       named(3,Mary)
       female(3)
       named(1,John)
       male(1)

       DRS: [6,4,2]
       tell(1,3,say(6,like(4,3)))
```

```
       6 is 6 fat(6) like(6,5) man(6) male(6)
       like(4,3)
       4 is 4 man(4) male(4) stupid(4)
       garden(2) neuter(2)
       live(1)in 2
       live(1)

*/

%   prepositional phrases
% + definite articles

disc(13,[
 "John lives in a house with a garden.",              % house
 "A dog chases a fat cat round the garden.",          % chase
 "Another cat in the house purrs.",                   % cat
 "This cat is lazy.",
 "The fat cat likes the cat in the house.",           % type in cat
 "The fat cat had loved the lazy cat in the garden.", % type in love
 "Who does the fat cat like?",               % cat(6) lazy(6)
 "Who did the fat cat love?"                          % cat(6) in house(3)
        ]).
/*
Focus: [4,2,6,3,5,1]

global DRS: [1]
       named(1,John)
       male(1)

       DRS: [6,4,5,2,3]
       love(4,6)in 2
       love(4,6)
       cat(6)in 3
       like(4,6)
       lazy(6) purr(6) cat(6) neuter(6)
       chase(5,4)round 2
       cat(4) neuter(4) fat(4)
       chase(5,4)
       dog(5) neuter(5)
       garden(2) neuter(2)
       house(3)with 2
       house(3) neuter(3)
       live(1)in 3
```

```
      live(1)
*/
%   quantifier
% + questions

disc(14,[
 "Every man is afraid of a dog.",                      % type y
 "Is Bill afraid of a dog?",                           % no
 "Bill is a man.",
 "Is Bill afraid of a dog?",                           % yes
 "Is Bill afraid of the dog?",                         % yes
 "The dog is fat.",
 "Who is afraid of the dog?",                          % Bill
 "Another dog chases a cat.",
 "John is afraid of this dog.",
 "Which dog is John afraid of?",           % dog(X) chase(X,Y)
 "Is John afraid of the fat dog?",                    % no
 "Is John afraid of the dog which chases a cat?", % yes
 "John is a man.",
 "Which dog is John afraid of?",                      % ... both dogs
 "Who is afraid of a dog?"                            % Bill John
        ]).

%   quantifier

disc(15,[
 "Every fat man gave a stupid woman a valuable book."
       ]).

%   quantifier
% + modal adjectives

disc(16,[
 "Every soldier is a possible murderer.",
 "John is a soldier.",
 "Is John a murderer?",
 "Is John a possible murderer?"
       ]).

%   quantifier
% + modal adjectives

disc(17,[
```

```
 "Every man who owns a big knife is a possible murderer.",
 "John is a man.",
 "Is John a murderer?",               % no
 "Is John a possible murderer?",      % no
 "John owns a knife.",
 "Is John a possible murderer?",      % no
 "The knife is very big.",
 "Is John a murderer?",               % no
 "Is John a possible murderer?"]).    % yes

%   quantifier
%   compare with the old box model

disc(18,[
 "Every woman likes John, who is fat.",
 "Is John fat?"                       % yes
        ]).

%   quantifier
% + nested phrases

disc(19,[
 "John said that every woman likes a man."
        ]).

%   quantifier
% + questions

disc(20,[
 "Every man loves a woman.",
 "John is a man.",
 "Bill is a man.",
 "Stefan is a man.",
 "Tigger is a cat.",
 "Who does love a woman?",
 "Does John love a woman?"
        ]).

% donkey sentences

% if you type in yes here you are the one
% who causes trouble
```

```
disc(21,[
 "Every man owns a donkey which loves him."
        ]).

disc(22,[
 "Every man who owns a donkey beats it."
        ]).


disc(23,[
 "Every man is afraid of a dog.",       % yes or no
 "John is a man.",
 "John is afraid of a huge dog.",
 "Which dog is John afraid of?"         % a special or general
% and the huge one
        ]).

disc(24,[
 "John is a fat man.",
 "Every woman likes him.",
 "Does every woman like him?",
 "Does every woman like John?",
 "Barbara is a woman.",
 "Does Barbara like him?"
        ]).

%   nested phrases

disc(25,[
 "John said that he wants to help Scotland."
        ]).

disc(26,[
 "Every man is afraid of a dog.",
 "They are stupid.",
 "John is a man.",
 "Is John stupid?",                 % yes
 "Is John afraid of a dog?"         % yes
        ]).

%   focus
% + definite articles
```

```
disc(27,[
 "John lives in a house with a garden.",        % house
 "A fat cat chases Tigger round the garden.",   % this is not ambiguous
% because my grammar
% doesn't allow PPs
% to modify
% proper nouns
 "Another cat which is very large sleeps.",
 "John likes the fat cat.",
 "It is lazy.",
 "Which cat is lazy?"
        ]).
/*
Focus: [5,1,6,2,4,3]

global DRS: [4,1]
       named(4,Tigger) male(4)
       named(1,John) male(1)

       DRS: [6,5,2,3]
       lazy(5)
       like(1,5)
       sleep(6) large(6) cat(6) neuter(6)
       chase(5,4)round 2
       chase(5,4)
       cat(5) neuter(5) fat(5)
       garden(2) neuter(2)
       house(3)with 2
       house(3) neuter(3)
       live(1)in 3
       live(1)
*/

disc(28,[
 "All men that own knives are possible murderers."
        ]).

disc(29,[
 "Every woman likes herself.",
 "Mary is a woman.",
 "John likes himself.",
 "Who does like himself?"             % John Mary (!?)
        ]).
```

```
% This might cause trouble with feminists.


/* doesn't work yet
   the answering procedure has to be augmented

   sentence conjuntion causes problems too

%   sentence conjunction
% + quantifier
% + question answering with some kind of drawing conclusions

disc(31,[
 "Every man is a lunatic and every lunatic loves a woman.",
 "John is a man.",
 "Mary is a woman.",
 "Does John love Mary?"
        ]).

*/

% doesn't work yet

disc(32,[
 "A man was sure that John had said that he wants to help Scotland."
]).

%   prepositional phrases

disc(33,[
 "John lives in a house with a garden.",  % house
 "He owns a cat.",
 "John saw the cat in the garden."       % only saw is accepted
]).

%   prepositional phrases

disc(34,[
 "John lives in a house with a garden.",        % house
 "He owns a cat.",
 "This cat chases another cat in the garden.",  % type in cat
 "John saw the cat in the garden."              % cat or saw
        ]).
```

```
% prepositional phrases

disc(35,[
 "John lives in a house with a garden.",        % house
 "He lives in this house with a woman."         % live
% house fails
]).

disc(36,[
 "Every man who is afraid of a dog is anxious."
]).

% doesn't work yet
disc(37,[
 "A woman loves every man."
]).

%   quantifiers
% + questions

disc(38,[
 "Every stupid woman likes John.",
 "Does every woman like John?",
 "Does every stupid woman like John?",
 "Does every stupid fat woman like John?"
]).

disc(39,[
 "Most men sleep.",
 "Do most men sleep?"
]).

%   definite articles
% + relative clauses

disc(40,[
 "A man likes Mary.",
 "A man likes Bill.",
 "The man who likes Mary likes John."
]).
```

References

[1] C Mellish, P Whitelock *Computational Syntax - AI3 Lecture Notes* Edinburgh University 1992

[2] Fernando C.N. Pereira, Stuart M. Shieber *Prolog and Natural-Language Analysis*

[3] Barbara J Groz *The Representation and Use of Focus in a System for Understanding Dialogs*

[4] Candace L Sidner *Focusing in the Comprehension of Definite Anaphora*

[5] Alan Black *Computational Semantics - AI3 Lecture Notes* Edinburgh University 1993

[6] Christoph Lehner *Prolog und Linguistik*

[7] Gerald Gazdar, Christopher Mellish *Natural Language Processing in Prolog*

[8] Jerry Hobbs, Stuart Shieber *An Algorithm for Generating Quantifier Scope* Computational Linguistics, Vol. 13, Nos 1-2, pp 47-63 (1987)

[9] Rodney Huddleston *English Grammar an outline*