Contents

A The Code

This appendix contains the code for the described implementation.[26] Since it seems easier to understand the code, if the internal format is known, TROLL's way of representing feature graphs internally is described here. A feature graph is a triple, consisting of the feature tree, the reentrancy list and the dependency list, i.e. `fs(FT,RL,DL)`. The feature tree is encoded by having nodes with the types and features, i.e. `node(Type,Features)`. The features are a prolog list with arguments consisting of a feature and a node, i.e. `[feat(Feat1,Node1),feat(Feat2,Node2),...]`. The features are ordered alphabetically on the feature list. Reentrancies in the feature tree are wrapped, so that they are easily recognizable, i.e. `rnum(Num)`. The reentrancy list consists of entries of a variable and the appropriate node, i.e. `[re(Num1,Node1),re(Num2,Node2),...]`. The reentrancy list may contain chains. The dependency list entries are of the format `group(Group,Template,Cases)`. Group is a variable which identifies the group. Template is a list of variables which enable to identify the possible values belonging to a particular variable, i.e. which values occur at a particular node. Cases is a list of lists which contains the columns which belong in some sense together, i.e. the single elements of this set of feature graphs. The format for the LP rules is `lp(FS1,FS2)`. Lexical entries are `lex_entry(Word,FS)` and rules are `rule(FS)`.

Some predicates do occur in the code, but are not defined since they were not written by me or are self explanatory. The most prominent of those are `unify_fs/3` and `subsumes/2`. The unification is non destructive which means that the result is in the third argument whereas the two input feature graphs remain unchanged in the first and second argument. Subsumption has to have the more general feature graph in the first argument, the specific one in the second. Auxiliary predicates are

`make_restrictor/2` with input a list of paths and output the needed format for `restrict/3`,

`restrict/3` with input restrictor and a feature graph and output the restricted version of the input feature graph,

`reentrancy_member/2` is just as the standard `member/2` predicate only for reentrancy lists,

`remove_reentrancy/3` removes an entry from the reentrancy list and returns the rest of the list,

`clean_up/2` cleans the reentrancy and dependency lists from chains of entries and entries which are no longer necessary,

`cpu_time/2` takes a goal and returns the time which was necessary to execute it.

[26] The code given in this section is somewhat pretty printed so that the predicate names correspond to the intuitions. In the real implementation this was not done because of compatibility reasons with other parsers.

`write_sentence/2` prints the sentence to the screen with the edge numbering and returns the length of the input string.

`get_chart_length/1` counts the edges on the chart.

`get_lp_rules/1` collects all the LP rules in a list.

`get_restrictor/1` returns the restrictor as a list of paths.

`get_lp_store/2` extracts the LP store from a PS rule.

`extract_lhs/2` extracts the left hand side from a PS rule.

`embed_lhs/2` embeds a feature structure so that it becomes the left hand side of a PS rule.

`add_lp_store/3` adds an LP store to a PS rule.

`get_length_rhs/2` calculates the length of the right hand side of a PS rule.

`empty_rhs/1` checks whether the right hand side of a PS rule is empty.

`append_fs/3` appends a FS at the beginning of a list of signs.

`copy_fs/4` copies a feature graph thereby removing all but the internal reentrancies.

`report/5` prints the cpu time used, the resulting number of edges on chart and agenda and the number of solutions or the solutions themselves.

A.1 The Parser

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                            %
%    Top-down ID/LP Earley parser for Troll that solves      %
%    the problem of non-local backward feature passing       %
%                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % In principle this is a modified version of the top-down parser found
 % in Gazdar & Mellish (1989) augmented to handle feature structures and
 % ID/LP grammars. It utilizes a chart and an agenda. Several
 % improvements have been added to make it more efficient. The
 % improvements involve the use of a restrictor to ensure termination
 % and several techniques to save on the prediction and the addition of
 % edges to the chart (Gerdemann 1991).

:- ensure_loaded(lp_acceptable).

 % parse(+String)
 % top level goal of the parser, resets the dynamic predicates in the
 % database and initializes the Restrictor, the StartRule, the LPRules,
```

```
 % the Chart and the Agenda. Additionally some output is given, namely
 % the input sentence with the respective vertex numbers and the
 % resulting output of the parser, i.e. timings and the counted edges.
 % Note that both Chart and Agenda are asserted in the database because
 % of Quintus memory problems.

parse(String):-
        reset_database,
        write_sentence(String,End),
        get_restrictor(Restrictor),
        make_start_rule(StartRule),
        get_lp_rules(LPRules),
        predict(StartRule,0,Restrictor,_Flag),
        scan(String),
        cpu_time(extend_edges(1,Restrictor,End,LPRules,Num_Edges),Time),
        report(parse,Time,End,Num_Edges,_Chart).


 % scan(+String)
 % takes the string and calls start_chart/2, starting with vertex 0.

scan(String):-
        start_chart(String,0).


 % start_chart(+String,+Vertex)
 % initializes the chart by asserting edges for all lexical entries
 % found for each word, and for all positions the empty categories are
 % added in the form of passive edges as well. This is done in a findall
 % loop to be able to discover easily whether there is a lexical entry
 % for a word at all. The edges get a 'passive' wrapper as to find them
 % easier in the database. Each lexical entry is turned into a ps_rule
 % with empty goals, rhs, lp store and rec features.

start_chart([],V0):-
        findall(edge(V0,V0,FT),
                (lex_entry([],fs(FT,RL,DL)),
                assertz(passive(V0,V0,fs(node(ps_rule,
                                 [feat(goals,node(e_list,[])),
                                 feat(lhs,FT),
                                 feat(lp_store,node(eset,[])),
                                 feat(rec,node(e_list,[])),
                                 feat(rhs,node(e_list,[]))])
                            ,RL,DL)))).
               _Chart).
```

```
start_chart([Word|Words],V0):-
        findall(edge(V0,V0,FT),
                (lex_entry([],fs(FT,RL,DL)),
                assertz(passive(V0,V0,fs(node(ps_rule,
                                 [feat(goals,node(e_list,[])),
                                 feat(lhs,FT),
                                 feat(lp_store,node(eset,[])),
                                 feat(rec,node(e_list,[])),
                                 feat(rhs,node(e_list,[]))])
                            ,RL,DL)))),
                _Chart),
        V1 is V0 +1,
        findall(edge(V0,V1,FT),
                (lex_entry(Word,fs(FT,RL,DL)),
                assertz(passive(V0,V1,fs(node(ps_rule,
                                 [feat(goals,node(e_list,[])),
                                 feat(lhs,FT),
                                 feat(lp_store,node(eset,[])),
                                 feat(rec,node(e_list,[])),
                                 feat(rhs,node(e_list,[]))])
                            ,RL,DL)))),
                Chart1),
        ( Chart1 = [] ->
            write('No lexical entry found for word >>'),
            write(Word),
            write(' <<'),nl,nl
        ;   true),
        start_chart(Words,V1).


 % extend_edges(+Counter,+Restrictor,+StringLength,+LPRules,-NumOfEdges)
 % takes an edge from the agenda, checks whether it is already subsumed
 % by an edge in the chart and starts the appropriate further operations
 % with the edge (predicate new_edges/5). The edge is asserted to
 % the chart just in case it was either a passive edge or an active edge
 % that was used in a successful prediction. It also counts the edges
 % taken from the agenda. The Restrictor, the StringLength and the
 % LPRules are just passed on.

extend_edges(N,Restrictor,End,LPRules,No):-
        retract(agenda(Edge)),!,
        (subsumes_edge(Edge) ->
            true
        ;   (new_edges(Edge,Restrictor,End,LPRules,Flag),
             (Flag == good ->
                 assertz(Edge)
```

```
          ;   true))),
        N1 is N+1,
        extend_edges(N1,Restrictor,End,LPRules,No).
extend_edges(N,_Restrictor,_End,_LPRules,NumOfEdges):-
        NumOfEdges is N-1.


 % new_edges(+Edge,+Restrictor,+StringLength,+LPRules,-Flag)
 % takes the edge and calls the appropriate complete and prediction
 % steps. The Restrictor, the StringLength and the LPRules are just
 % passed on. The Flag markes whether the edge has to be asserted to the
 % chart.

new_edges(passive(V1,V2,Rule),_Restrictor,End,LPRules,good) :-
        complete_passive(V1,V2,Rule,End,LPRules).
new_edges(active(V1,V2,Rule),Restrictor,End,LPRules,Good) :-
        predict(Rule,V2,Restrictor,Good),
        complete_active(V1,V2,Rule,End,LPRules).


 % complete_passive(+From,+To,+Rule,+StringLength,+LPRules)
 % does the actual completion step. Since the parser deals with
 % non-local backward feature passing, the process is a bit more
 % complicated than usual though the principle stays the same.
 % The Left Hand Side of the input Rule is extracted to check whether
 % there exists an active edge in the chart where one of its elements on
 % the right hand side unifies with this LHS. This is done in a fail
 % loop. To be able to unify the structures properly, the length of the
 % right hand side is computed and LHS is embedded nondeterministically
 % on all possible positions on the righthandside. If this unification
 % was successful, the LP stores are extracted, unified, checked for
 % acceptance and added to the resulting new rule. Only if this does not
 % lead to a violation the new edge is constructed. This construction
 % (change_fs/4) contains the check for LP acceptability - including the
 % construction of a new LP store - and the moving of the recognized
 % category from the right hand side of the rule to the recognized
 % sequence of FSs. If the righthandside is empty, the goals are
 % executed and a cleaned version of the edge is asserted. Otherwise,
 % the resulting cleaned edge is asserted.

complete_passive(V1,V2,Rule,End,LPRules):-
        extract_lhs(Rule,LHS),
        active(V0,V1,Rule1),
        get_length_rhs(Rule1,Length),
        embed_fs(LHS,Length,RuleLHS,Pos),
        unify_fs(RuleLHS,Rule1,Rule2),
```

```
        get_lp_store(Rule,LPStore),
        get_lp_store(Rule2,LPStore2),
        union(LPStore,LPStore2,OutLP_Store1),
        check_lp_store(OutLP_Store1,LPRules,OutLP_Store),
        add_lp_store(Rule2,OutLP_Store,Rule3),
        change_fs(Rule3,Pos,LPRules,Rule4),
        ( empty_rhs(Rule4) ->
            execute_goals(Rule4,OutRule4),
            clean_up(OutRule4,OutRule),
            assertz(agenda(passive(V0,V2,OutRule)))
        ;   clean_up(Rule4,OutRule),
            assertz(agenda(active(V0,V2,OutRule)))),
        fail.
complete_passive(_,_,_,_,_).


 % complete_active(+From,+To,+Rule,+StringLength,+LPRules)
 % does the actual completion step. An appropriate passive edge is
 % taken from the chart, its left hand side taken and embedded
 % nondeterministically according to the computed length of the right
 % hand side of the active rule to be able to test unifiability. If this
 % unification was successful, the LP stores are extracted, unified,
 % checked for acceptance and added to the resulting new rule. Only if
 % this does not lead to a violation the new edge is constructed. This
 % construction contains the check for LP acceptability - including the
 % construction of a new LP store - and the moving of the recognized
 % category from the right hand side of the rule to the recognized
 % sequence of FSs. If the right hand side is empty, the goals are
 % executed and a cleaned version of the edge is asserted. Otherwise,
 % the resulting cleaned edge is asserted.

complete_active(V1,V2,Rule,End,LPRules):-
        get_length_rhs(Rule,Length),
        passive(V2,V3,Rule1),
        extract_lhs(Rule1,LHS1),
        embed_fs(LHS1,Length,RuleLHS1,Pos),
        unify_fs(RuleLHS1,Rule,Rule2),
        get_lp_store(Rule1,LPStore1),
        get_lp_store(Rule2,LPStore2),
        union(LPStore1,LPStore2,OutLP_Store1),
        check_lp_store(OutLP_Store1,LPRules,OutLP_Store),
        add_lp_store(Rule2,OutLP_Store,Rule3),
        change_fs(Rule3,Pos,LPRules,Rule4),
        ( empty_rhs(Rule4) ->
            execute_goals(Rule4,OutRule4),
            clean_up(OutRule4,OutRule),
```

```
            assertz(agenda(passive(V1,V3,OutRule)))
        ;   clean_up(Rule4,OutRule),
            assertz(agenda(active(V1,V3,OutRule)))),
        fail.
complete_active(_,_,_,_,_).


 % predict(+Rule,+Vertex,+Restrictor,-Flag)
 % constitutes the prediction step of the algorithm. The Rule is taken
 % and via fail loop all FSs on the right hand side are extracted and
 % tried for prediction. To ensure termination only the restricted
 % versions of the FSs are used. For greater efficiency only those
 % predictions are actually made that haven't been done before. A Flag
 % markes if there have been any predictions at all. If there are none
 % or have been none the FS does not need to be added to the chart.
 % This is the version that does not check lp acceptability.

predict(InRule,Num,Restrictor,_Flag):-
        get_next_fs(InRule,FS),
        restrict(Restrictor,FS,OutFS),
        (already_predicted(Num,OutFS,Flag) ->
            (Flag = good ->
                assert(flag(good))
            ;   true)
        ;   predict_aux(Num,OutFS),
            (flag(good) ->
                assertz(predicted(Num,OutFS,good))
            ;   assertz(predicted(Num,OutFS,bad)))),
        fail.
predict(_,_,_,good):-
        flag(good),
        retractall(flag(_)).
predict(_,_,_,bad):-
        \+flag(good).


 % predict_aux(+Vertex,+FS)
 % does the actual prediction. The FS is embedded under the left hand
 % side and then tried to unify with any rule. An active edge is
 % constructed and asserted to the agenda.

predict_aux(Num,OutFS):-
        embed_lhs(OutFS,RuleFS),
        rule(NewRule),
        unify_fs(NewRule,RuleFS,Rule),
        clean_up(Rule,OutRule),
```

```
        assertz(agenda(active(Num,Num,OutRule))),
        assertz(flag(good)),
        fail.
predict_aux(_,_).
```

A.2 LP Acceptability

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    LP acceptability                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % poss_lp_accept(+SequenceOfFS,+RL,+DL,+LPRules,+InFlag,-OutFlag)
 % fails if the SequenceOfFS is not LP acceptable, unpacks the first
 % level of the sequence, recurses down the rest, returns yes in OutFlag
 % if the sequence was possibly LP violated.

poss_lp_accept(rnum(Num),RL,DL,LPRules,InFlag,OutFlag):-
        clean_up(fs(rnum(Num),RL,DL),fs(NewFS,RL0,DL0)),
        poss_lp_accept(NewFS,RL0,DL0,LPRules,InFlag,OutFlag).
poss_lp_accept(node(ne_list_sign,[feat(hd,_),
                                  feat(tl,node(e_list,[]))]),_,_,
              _LPRules,Flag,Flag):- !.
poss_lp_accept(node(ne_list_sign,[feat(hd,HD),
                                  feat(tl,TL)]),RL,DL,
              LPRules,InFlag,OutFlag):-
        poss_lp_acceptable(LPRules,fs(HD,RL,DL),fs(TL,RL,DL),no,Flag),
        (Flag == no ->
            InFlag1 = InFlag
        ;   InFlag1 = yes),
        poss_lp_accept(TL,RL,DL,LPRules,InFlag1,OutFlag).


 % poss_lp_acceptable(+LP_Rules,+FirstOfRec,+RestOfRec,+InFlag,-OutFlag)
 % unpacks the FSs from the lp wrapper, fails if First is not allowed
 % after Rest, descends recursively through all LP rules. This is the
 % version that cares for the cases of possible lp violation as well as
 % lp acceptance, the resulting lp situation is marked in OutFlag.

poss_lp_acceptable([],_First,_Rest,Flag,Flag).
poss_lp_acceptable([lp(FS1,FS2)|T],First,fs(Rest,RL,DL),InFlag,OutFlag):-
        poss_lp_aux_1(Rest,RL,DL,First,FS1,FS2,Flag),
        ( Flag == yes ->
            InFlag1 = yes
        ;   InFlag1 = InFlag),
        poss_lp_acceptable(T,First,fs(Rest,RL,DL),InFlag1,OutFlag).
```

```
 % poss_lp_aux_1(+RestList,+RL,+DL,+First,+FS1,+FS2,-Flag)
 % checks whether First is subsumed by the second FS1 of the lp rule, if
 % this is not the case, a test unification is performed to determine
 % whether an lp rule might apply, if that is not the case as well, it
 % is not necessary to check lp acceptability for this constellation;
 % if any of this is the case, it remains to be checked whether one of
 % the elements from the RestList is subsumed by FS2 -> violation of the
 % lp rule if we had a subsumption for FS1 as well, otherwise the test
 % is for possible lp violation. The result is stored in Flag.

poss_lp_aux_1(rnum(Num),RL,DL,First,FS1,FS2,Flag):-
        clean_up(fs(rnum(Num),RL,DL),fs(NewFS,NewRL,NewDL)),
        poss_lp_aux_1(NewFS,NewRL,NewDL,First,FS1,FS2,Flag).
poss_lp_aux_1(node(e_list,[]),_RL,_DL,_First,_FS1,_FS2,_Flag):- !.
poss_lp_aux_1(node(ne_list_sign,List),RL,DL,First,FS1,FS2,Flag):-
        (subsumes(FS1,First) ->
            poss_lp_aux(node(ne_list_sign,List),RL,DL,FS2,no,Flag)
        ;   (unify_fs(FS1,First,_) ->
                poss_lp_aux(node(ne_list_sign,List),RL,DL,FS2,yes,Flag)
            ;   true)).


 % poss_lp_aux(+SubcatList,+FS,+RL,+DL,+InFlag,-OutFlag)
 % checks whether an element of List is subsumed by FS -> fails in
 % the case that the InFlag signals that there was already a subsumption
 % with the other lp element, a violation of an LP rule has occurred. In
 % the other cases a test for possible lp violation is done. if none of
 % these occur, the structure is lp acceptable.

poss_lp_aux(node(e_list,[]),_,_,_FS,Flag,Flag):- !.
poss_lp_aux(node(ne_list_sign,[feat(hd,Next),feat(tl,T)]),RL,DL,
                 FS,InFlag,OutFlag):-
        ( subsumes(FS,fs(Next,RL,DL)) ->
            ( InFlag == no ->
                fail
            ;   InFlag1 = yes)
        ;   (unify_fs(FS,fs(Next,RL,DL),_) ->
                InFlag1 = yes
            ;   InFlag1 = InFlag)),
        poss_lp_aux(T,RL,DL,FS,InFlag1,OutFlag).


 % lp_acceptable_rest(+LP_Rules,+FirstOfRec,+RestOfRec)
 % unpacks the FSs from the lp wrapper, fails if First is not allowed
```

```
 % before Rest, descends recursively through all LP rules.

lp_acceptable_rest([],_First,_Rest).
lp_acceptable_rest([lp(FS1,FS2)|T],First,fs(Rest,RL,DL)):-
        lp_aux_rest_1(Rest,RL,DL,First,FS2,FS1),
        lp_acceptable_rest(T,First,fs(Rest,RL,DL)).


 % lp_aux_rest_1(+RestList,+RL,+DL,+First,+FS2,+FS1)
 % checks whether First is subsumed by the second FS1 of the lp rule, if
 % this is not the case, it is not necessary to check lp acceptability
 % for this constellation; if it is the case, it remains to be checked
 % whether one of the elements from the RestList is subsumed by FS2 ->
 % violation of the lp rule.

lp_aux_rest_1(rnum(Num),RL,DL,First,FS2,FS1):-
        clean_up(fs(rnum(Num),RL,DL),NewFS),
        lp_aux_rest_1(NewFS,RL,DL,First,FS2,FS1).
lp_aux_rest_1(node(e_list,[]),_,_,_,_,_):- !.
lp_aux_rest_1(node(ne_list_sign,List),RL,DL,First,FS2,FS1):-
        (user:subsumes(FS2,First) ->
            lp_aux_rest(node(ne_list_sign,List),RL,DL,FS1)
        ;   true).

 % lp_aux_rest(+List,+RL,+DL,+FS)
 % checks whether an element of List is subsumed by FS -> fails in
 % this case since a violation of an LP rule has occurred.

lp_aux_rest(node(e_list,[]),_,_,_FS):- !.
lp_aux_rest(node(ne_list_sign,[feat(hd,Next),feat(tl,T)]),RL,DL,FS):-
        \+ user:subsumes(FS,fs(Next,RL,DL)),
        lp_aux_rest(T,RL,DL,FS).
```

A.3 Auxiliary Procedures

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          Edge Subsumption routines                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % subsumes_edge(+Edge)
 % tries to find an edge in the database that is identical to Edge with
 % respect to th vertices and where the rule subsumes the rule of the
 % input edge.
```

```
subsumes_edge(active(V0,V1,Rule)):-
        active(V0,V1,Rule1),
        subsumes(Rule1,Rule),
        !.
subsumes_edge(passive(V0,V1,Rule)):-
        passive(V0,V1,Rule1),
        subsumes(Rule1,Rule),
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                        Initialization routines               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % make_start_rule(-StartRule)
 % constructs a ps rule with the appropriately instantiated features and
 % values from the initial FS.

make_start_rule(fs(node(ps_rule,[feat(goals,node(e_list,[])),
                                 feat(lhs,node(dummy,[])),
                                 feat(lp_store,node(eset,[])),
                                 feat(rec,node(e_list,[])),
                                 feat(rhs,node(ne_list_sign,
                                           [feat(hd,Symbol),
                                            feat(tl,node(e_list,[]))]))]))
                        ,RL,DL)):-
        initial(fs(Symbol,RL,DL)).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               extracting of substructures from a FS         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % get_next_fs(+PSRule,-FS)
 % takes the input ps rule and returns a member of the right hand side
 % by calling member_rhs/4

get_next_fs(fs(node(ps_rule,[_Goals,_LHS,_LPStore,_Rec,
                        feat(rhs,RHS)]),RL,DL),fs(FS,RL,DL)):-
        member_rhs(RHS,RL,FS).


 % member_rhs(+RHS,+RL,-FS)
 % returns a FS from RHS.
```

```
member_rhs(rnum(Num),RL,FS):-
        reentrancy_member(re(Num,FT),RL),
        member_rhs(FT,RL,FS).
member_rhs(node(ne_list_sign,[feat(hd,FS),_]),_RL,FS).
member_rhs(node(ne_list_sign,[_,feat(tl,RestFS)]),RL,FS):-
        member_rhs(RestFS,RL,FS).


 % pop_rhs(+RHS,+RL,-RHS,-RL,+Acc,+Pos,-FS)
 % pops the FS at position Pos from the right hand side, returns the FS,
 % the resulting right hand side and the resulting reentrancy list.

pop_rhs(rnum(RNum),RL0,RNum,RL,Acc,Pos,FS):-
        remove_reentrancy(re(RNum,FT),RL0,RL1),
        pop_rhs(FT,RL1,PoppedFT,RL2,Acc,Pos,FS),
        RL = [re(RNum,PoppedFT)|RL2].
pop_rhs(node(ne_list_sign,
             [feat(hd,FS),
              feat(tl,Rest)]),RL,Rest,RL,Pos,Pos,FS):- !.
pop_rhs(node(ne_list_sign,
             [feat(hd,HD),
              feat(tl,Rest)]),RLIn,
        node(ne_list_sign,
             [feat(hd,HD),
              feat(tl,Out)]),RLOut,Acc,Pos,FS):-
        Acc1 is Acc+1,
        pop_rhs(Rest,RLIn,Out,RLOut,Acc1,Pos,FS).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               changing of feature structures                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % change_fs(+PSRule,+Pos,+LPRules,-PSRule)
 % The FS appropriate according to Pos is extracted from the right hand
 % side of the PSRule, a test is performed whether the feature graph in
 % question may precede all the remaining elements in the right hand
 % side (multi) set, then the test for possible lp violation is
 % performed. If the Flag markes no possible violation of LP rules,
 % the result is the feature structure as before.
 % If there is a possible violation of an LP rule, a new entry for the
 % LP store is created by producing a cleaned copy of the sequence in
 % question for the second element and a reentrancy is introduced that
 % links the first element of the pair to the recognized feature
 % structure. This new element is unified by union_aux2/6 into the
 % existing LP store.
```

```
change_fs(fs(node(ps_rule,
               [feat(Goal,Goals),
                feat(lhs,LHS),
                feat(lp_store,LPStore),
                feat(rec,Rec),
                feat(rhs,RHS)]),RL0,DL),
           Pos,
           LPRules,
           fs(node(ps_rule,
               [feat(Goal,Goals),
                feat(lhs,LHS),
                feat(lp_store,NewLPStore),
                feat(rec,NewRec),
                feat(rhs,RestRHS)]),RL,DL)):-
        pop_rhs(RHS,RL0,RestRHS,RL1,1,Pos,FS),
        lp_acceptable_rest(LPRules,fs(FS,RL0,DL),fs(RestRHS,RL1,DL)),
        poss_lp_acceptable(LPRules,fs(FS,RL0,DL),fs(Rec,RL0,DL),no,Flag),
        append_fs(Rec,FS,NewRec0),
        ( Flag == no ->
            NewLPStore = LPStore,
            RL = RL1,
            NewRec = NewRec0
        ;   copy_fs(NewRec0,RL0,DL,fs(CopyFS,RL4,_DL1)),
            union_aux2(LPStore,RL0,DL,
                  node(pair,[feat(first,rnum(A)),
                             feat(second,CopyFS)]),[re(A,NewRec0)|RL4],
                  NewLPStore,_RL3,DL3),
            NewRec = rnum(A),
            RL = [re(A,NewRec0)|RL0]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               adding of information to Feature structures    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % embed_fs(+FS,+MaxEmbedding,-PSRule,-Pos)
 % The intention is to embed a FS on the right hand side of a rule. Here
 % it has to be possible to embed on every depth up to MaxEmbedding
 % since we are dealing only with ID rules. The depth on which the
 % embedding took place is returned in Pos.

embed_fs(fs(FT,RL,DL),Max,fs(node(ps_rule,[feat(rhs,RHS)]),RL,DL),Pos):-
        embed_aux(FT,Max,RHS,1,Pos).
```

```
 % embed_aux(+FS,+MaxEmbedding,-RHS,+Acc,-Acc)
 % takes FS and embeds it in a hpsg type of list as long as the
 % accumulator is smaller or equal to the MaxEmbedding.

embed_aux(FS,_,node(ne_list_sign,[feat(hd,FS)]),Pos,Pos).
embed_aux(FS,Max,node(ne_list_sign,[feat(hd,node(bot,[])),feat(tl,TL)]),
              Acc,Pos):-
        Max > Acc,
        Acc1 is Acc +1,
        embed_aux(FS,Max,TL,Acc1,Pos).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Comparison of feature structures               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % diff(+FS1,+FS2)
 % succeeds if FS1 and FS2 do not subsume each other.

diff(FS1,FS2):-
        \+identical(FS1,FS2).


 % identical(+FS1,+FS2)
 % succeeds if FS1 and FS2 do subsume each other.

identical(FS1,FS2):-
        subsumes(FS1,FS2),
        subsumes(FS2,FS1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Operations on lp stores                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % check_lp_store(+LPStore,+LPRules,-LPStore)
 % extracts the pairs from the LP store, where the elements differ and
 % checks whether they are lp acceptable. Returns an LP store that
 % contains only the elements which are still in doubt.

check_lp_store(fs(node(eset,[]),RL,DL),_,fs(node(eset,[]),RL,DL)):- !.
check_lp_store(fs(InLP_Store,RL,DL),LPRules,fs(OutLP_Store,RL0,DL0)):-
        dif(InLP_Store,RL,DL,LPRules,OutLP_Store,RL0,DL0).
```

```
 % dif(+LPStore,+RL,+DL,+LPRules,-LPStore,-RL,-DL)
 % finds the elements in LPStore where the cleaned first and second
 % elements of a pair are different and tests possible LP violation
 % on them. If they are LP acceptable, they are removed from the LP
 % store.

dif(node(eset,[]),RL,DL,_,node(eset,[]),RL,DL):- !.
dif(node(neset_pair,[feat(elt,node(pair,[feat(first,FS1),
                                         feat(second,FS2)])),
                     feat(els,Rest)]),RL,DL,
    LPRules,
    OutFS,RL0,DL0):-
        clean_up(fs(FS1,RL,DL),FS1Out),
        diff(FS1Out,fs(FS2,RL,DL)),
        !,
        poss_lp_accept(FS1,RL,DL,LPRules,no,Flag),
        ( Flag == yes ->
            OutFS = node(neset_pair,
                          [feat(elt,node(pair,
                                [feat(first,FS1),
                                 feat(second,FS2)])),
                           feat(els,RestFS)])
        ;   OutFS = RestFS),
        dif(Rest,RL,DL,LPRules,RestFS,RL0,DL0).
dif(node(neset_pair,[feat(elt,node(pair,Pair)),
                     feat(els,Rest)]),RL,DL,
    LPRules,
    node(neset_pair,[feat(elt,node(pair,Pair)),
                     feat(els,OutFS)]),RL0,DL0):-
        dif(Rest,RL,DL,LPRules,OutFS,RL0,DL0).


 % union(+LPStore1,+LPStore2,-LPStore)
 % It is necessary to unify the LP stores in the parser but this can't
 % be done with the ordinary unification algorithm since set unification
 % is essentially the same as list unification in Troll. union/3 is the
 % top level predicate of this process, it catches the easy cases and
 % calls union_aux1/6 to do the work. The proper set unification can be
 % achieved here only for this special case because of the format of the
 % data. If you want this in general for Troll, you need another logic.

union(fs(node(eset,[]),_,_),LPStore,LPStore):- !.
union(LPStore,fs(node(eset,[]),_,_),LPStore):- !.
union(fs(LPStore1,RL1,DL1),fs(LPStore2,RL2,DL2),fs(OutLPStore,RL,DL)):-
        union_aux1(LPStore1,RL1,DL1,LPStore2,RL2,DL2,OutLPStore,RL,DL).
```

```
 % union_aux1(+LPStore1,+RL1,+DL1,+LPStore2,+RL2,+DL2,-LPStore,-RL,-DL)
 % recurses through the first LP store and extracts the relevant pairs
 % and calls union_aux2/6 to unify this pair into the second LP store.

union_aux1(node(eset,[]),_,_,LPStore,RL,DL,LPStore,RL,DL):- !.
union_aux1(node(neset_pair,[feat(elt,Pair),feat(els,Rest)]),RL1,DL1,
      LPStore,RL2,DL2,
      OutLPStore,RL,DL):-
        union_aux2(LPStore,RL2,DL2,Pair,RL1,DL1,NewLPStore,RL0,DL0),
        union_aux1(Rest,RL1,DL1,NewLPStore,RL0,DL0,OutLPStore,RL,DL).


 % union_aux2(+LPStore,+RL,+DL,+Pair,+RL,+DL,-LPStore,-RL,-DL)
 % the pair can be unified into the input LP store just in case it is
 % not already there, or we find another pair whose second element is
 % literally identical to the second element of the input pair. In this
 % case the first elements of those pairs have to be unified. The
 % predicate recurses through the LP store.

union_aux2(node(eset,[]),RL1,DL1,
          Pair,RL,DL,
          node(neset_pair,[feat(elt,Pair),
                           feat(els,node(eset,[]))]),OutRL,OutDL):-
        !,
        append(RL,RL1,OutRL),
        append(DL,DL1,OutDL).
union_aux2(node(neset_pair,[feat(elt,node(pair,[feat(first,First1),
                                               feat(second,Second1)])),
                           feat(els,Rest)]),RL1,DL1,
          node(pair,[feat(first,First2),
                     feat(second,Second2)]),RL2,DL2,
          node(neset_pair,[feat(elt,node(pair,[feat(first,First),
                                               feat(second,Second)])),
                           feat(els,Rest)]),RL,DL):-
        identical(fs(Second1,RL1,DL1),fs(Second2,RL2,DL2)),
        !,
        user:unify_fs(fs(First1,RL1,DL1),fs(First2,RL2,DL2),fs(First,RL,DL)),
        clean_up(fs(First,RL,DL),fs(Second,_RL3,_DL3)).
union_aux2(node(neset_pair,[feat(elt,Pair1),feat(els,InRest)]),RL1,DL1,
          Pair,RL2,DL2,
          node(neset_pair,[feat(elt,Pair1),feat(els,OutRest)]),RL,DL):-
        union_aux2(InRest,RL1,DL1,Pair,RL2,DL2,OutRest,RL,DL).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                         Auxiliary predicates                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % already_predicted(+Num,+FS,-Flag)
 % checks if the prediction to be made has already been done with a FS
 % that subsumes the input FS. If this is successful the Flag indicates
 % whether any predictions were made from it.

already_predicted(Num,FS,Flag):-
        predicted(Num,FS1,Flag),
        subsumes(FS1,FS),
        !.
```