An Alternative Approach to ID/LP: Precompilation of LP rules into the Lexicon

Frank Morawietz*
Seminar für Sprachwissenschaft
Eberhard–Karls–Universität Tübingen
Kleine Wilhelmstraße 113
72074 Tübingen
Germany

December 8, 1993

The program which is the topic of this write–up is a result of work for the seminar 'Advanced Prolog Techniques' taught by PhD. D. Gerdemann in the SS 1993 at the University of Tübingen. The literature used as background consisted of Richard A. O'Keefe's book 'The Craft of Prolog'[1], 'Techniques of Prolog Programming' by Tu Van Le[2] and 'Advanced Prolog, techniques and examples' by Peter Ross[3].

1 The Idea

HPSG employs the ID/LP format as a tool to deal with relations of dominance and linear precedence. Since parsing of this format is problematic because of the possible nonlocal flow of information[4] and generally because the semantics of LP rules are difficult to define, alternative ways of dealing with this format are desirable.

This program tries to precompile the information on linearization. Since the information which categories may appear on the righthandsides of ID schemata is connected with the lexical head employed in the relevant schema, it is possible to put the information on linearization at the same place. Naturally this demands additional changes. Originally a schema matched with a lexical head to produce an instance of the schema according to the head's subcat list, which was then treated as an ID schema, i.e. allowed permutations of the righthandsides of this instance of the schema which are acceptable by the LP rules. Now we have to assume that the abstract schema matches with the lexical head to produce an instance of the schema, but since we already enforced the order of the elements the head subcategorizes for, we have to treat the order of the complements as fixed, i.e. the schema is not longer an ID schema. It is immediately clear that this produces new problems. How do we position the head in

*Email: `frank@sfs.nphil.uni-tuebingen.de`

[1]Richard A. O'Keefe (1990) *The Craft of Prolog* MIT Press.

[2]Tu Van Le (1993) *Techniques of Prolog Programming* John Wiley & Sons.

[3]Peter Ross (1989) *Advanced Prolog, techniques and examples* Addison–Wesley.

[4]For a detailed discussion of the problems involved see R. Seiffert (1988) *Chart Parsing of Unification ID/LP Grammars* and F. Morawietz (ms) master's thesis.

relation to the complements? What happens if we don't have a lexical head in the schema? What happens in the case of argument raising? The first two can be overcome by having a fixed order of the daughters in the schemata itself. Argument raising is not that easy, but one might think of some kind of shuffling operation according to the LP rules. This wouldn't be a precompilation, but would have to be done during runtime. Nevertheless, the solution of those problems is not the issue here. One severe problem remains anyway. The information on the categories on the subcat list may not be fully instantiated, i.e. they may be partial. Thus it might happen that the information to determine LP acceptability is not available, i.e. the LP rules would not apply as desired. The consequence of this is, that one would have to check LP acceptability during processing again since the subcat list of a lexical entry may be invalid according to linearization[5]. The program is not concerned with this check during parsing but deals solely with the precompilation of the information on linear precedence into the list of complements.

Additionally the transitive closure of the LP rules is precomputed as well, though it is theoretically not clear whether this is necessary. It is done using a slight variation of a standard algorithm for such computations, namely Warshall's algorithm taken from O'Keefe's book (p. 171). At the moment this is done under the assumption that transitivity applies when two LP rules contain feature terms that unify with each other. This seems linguistically not desirable. Better would be to test for identity of the denotations of the two feature terms. Identity could be implemented as vice versa subsumption. This could be altered easily but since this is only a prototype no further effort went into it[6].

The program has been hooked up to an older version of an already existing system named TROLL (Gerdemann (1993)) to guarantee a full functioning prototype.

Practically this precompilation means to take a lexicon and a set of LP rules and produce a new lexicon. In more detail it consists of taking each lexical entry, looking for its subcat list (or subject and comps feature in newer versions of HPSG), generating all LP acceptable permutations and constructing new lexical entries, in this prototype one for each possible permutation of the subcat list. Naturally this might lead to a high number of entries for each item and through that to a high level of ambiguity for the parser. To reduce at least the space needed for storage of those lexical items, one might consider using dependent disjunctions to compact several entries back into one entry with a dependent disjunction of the elements of its subcat list. But since TROLL handles only atomic types as dependent disjuncts this would work only for a limited number of the entries namely those that differ only in their atomic types. To give a simplified example, the compaction would work for subcat lists like $\langle NP_{nom}, NP_{acc}, NP_{dat}\rangle$ and $\langle NP_{nom}, NP_{dat}, NP_{acc}\rangle$, but not for entries with subcat lists like $\langle NP_{nom}, VP, NP_{acc}\rangle$ and $\langle NP_{nom}, NP_{acc}, VP\rangle$.

2 Usage

The program consists of three main parts. Three versions of the actual permutation program, called `compile_version_N.pl`; and three versions of the outer frame that provides

[5] This is wrong – use unification to construct even more entries and everything should work. Is there a paralellism between these HPSGs and Unification Categorial Grammar?

[6] It hasn't been done because at the time when the program was written subsumption was not available. To change it now would mean that all the timings would have to be redone and I think this not worth the effort because the improvement in the code can be seen with the other older version as well.

the necessary auxiliary predicates, called `precompile_lp_version_N.pl`. And a file called `warshall.pl` that contains code for computing the transitive closure of the LP rules. Additionally there exist procedures to read a lexicon and a file with LP rules. No effort was made to optimize them (`process_lex.pl`, `process_lp.pl`).

If you want to test this program, load `troll.pl` into Quintus, this includes the hpsg type hierarchy which is in the same directory. Call `precompile(hpsg_lp,hpsg_lex)`. The output consists in the durations for the compilation and the new lexicon in pretty printed version.

Currently the setup is to use `precompile_lp_version_3.pl` and `compile_version_1.pl`. If you want to change this, edit `troll.pl` for a change of the `precompile_lp_version_N.pl` files and them for a change of the `compile_version_N.pl` files. Otherwise everything works as originally designed in this old TROLL version.

3 The Code

All the code has been written for the HPSG type hierarchy (`hpsg_th`) which is in the same directory. Some of the prolog predicates rely on the structure imposed by the hierarchy, so if you change the hierarchy make sure you don't need to alter the program as well. What is important is the path to the subcat list, because that is the information we need to precompile the information on linear precedence into.

3.1 Auxiliary files

There exist four files that are not directly needed for the precompilation, but were necessary to get the prototype running. No effort was made to optimize the code in these files. All of them are commented and you can look there if you want to know more details for example about the syntax of the lexical entries or the LP rules.

Firstly there is a lexicon with seven entries, four verbs and three nouns. The file is called `hpsg_lex`. The verbs are: walks, sees, sells, and an imaginary five place relation verb called test. The nouns are: she, kim and book.

Secondly there is a file containing a set of LP rules, called `hpsg_lp`. The simplified LP rules are:

- $NP_{nom} \prec NP_{acc}$
- $NP \prec VP_{fin\&\neg inverted\&\neg aux}$
- $VP_{fin\&inverted\&\neg aux} \prec NP$
- $bot \prec PP_{to}$
- $VP_{aux} \prec VP_{\neg aux}$

Those LP rules are compiled to fifteen new LP rules by the warshall algorithm by employing the transitive closure operation described above.

The other two files are variations of the `process_fs.pl` file. They are called `process_lp.pl` and `process_lex.pl`. Both contain a main goal (`process_lex(File)` or `process_lp(File)`) which reads the file, creates a new file with the entries in prolog format and prints the resulting structures to the screen. Additionally both contain goals that just process the file without printing the results (`read_lex(File)` or `read_lp(File,LP_List)`). The LP version also returns the resulting LP rules in a list for further processing.

3.2 warshall.pl

This file has as main goal `closure(LP_List,File)`. The only part that has been changed in relation to O'Keefe's version is that the memberchk/2 test in warshall/4 has been substituted by a test whether there is a possible unification so that transitivity applies[7].

Since the algorithm utilizes another format of the data some extra predicates have been written to convert the respective formats into each other.

3.3 `precompile_lp_version_N.pl`

This family of files contains the code for all operations except the actual compilation itself. The main goal `precompile(LP_File,Lex_File)` takes a file with LP rules and a lexicon in the usual TROLL format, transforms both to prolog (thereby generating the predicates `lp(FS1,FS2)` and `lex_entry(Word,FS)`), computes the transitive closure for the lp rules, writes them into a file in prolog format, then constructs a new lexicon by using the lp rules on the subcat lists of the lexical entries. The new lexicon is written into a file too.

The optimization level of the program increases with the increasing number of the version.

The changes made from the first to the second version affects the predicates process/3 and extract_member/4. The changes were all concerned with making the code deterministic as far as possible.

In the first version process/3 contained a check whether the subcat list needed precompilation at all. This was done by having a three way disjunction listing the possibilities without the usage of first argument indexing. This disjunction has been eliminated by introducing a new predicate necessary/1 which uses first argument indexing to be deterministic.

The other change does not make the code entirely deterministic, but improves on the first version nevertheless. The in version #1 not deterministic predicate extract_member/4 is used in both directions. In the one direction it is used to find a feature in a list. This can not deterministic since the features don't have to be there since TROLL does not demand total well–typing. But by changing the order of the arguments, it was possible to make it deterministic when used for construction of the new entries.

The changes from the second to the third version consist of alterations to the predicates process/3, convert_to_prolog/2 and make_entries/3.

The change of the first two predicates was necessary because both utilized some adapted version of the path_val/3 predicate taken from UNICORN[8]. In Troll it is somewhat dubious what it should return in case the feature structure does not contain the feature that is looked for. Both have now specific procedures that deal appropriately with the structures and are deterministic as far as possible. In the case of process/3 a whole new procedure has been introduced, namely extract_subcat/3.

[7] Here would the change to identity be appropriate.

[8] Hinrichs & Gerdemann (1989).

The predicate make_entries/3 has been changed in the sense that the family of predicates make_entry_N/3 has been partially evaluated to yield just a single predicate make_entry/3.

3.4 `compile_version_N.pl`

The predicates in this family of files do the actual permutation of the elements of the subcat list according to the LP rules. The files differ in the principal approach to the solution of the problem. The first does create a permutation and checks whether this is lp acceptable before trying to generate the next one. The second generates all permutations for a given subcat list and generates via O'Keefe's sublist test the sublist that passes the test for LP acceptability. The third approach interleaves the generation of a permutation and the test for lp acceptability.

The first file contains already several versions of the code. The first is written without any attempts to optimize it. It has been improved in several steps. First by using the technique of lagging to use first argument indexing. Secondly by getting rid of unnecessary wrappers and introducing an additional check whether there is the possibility for a violation of the LP rule. Thirdly by case analysis and further lagging.

The code in the second file proceeds parallel to the development of the same problem in O'Keefe's book (p. 204). The only change is that the filter is now the test for LP acceptability.

The third file consists mainly of variations of the library predicates permutation/2 and insert/3 from the library(lists). An element is only added to the current permutation of the subcat list in creation if it passes LP acceptability. The code for LP acceptability is derived from the one in the first file.

4 Some Timings

Some runs of the program with different versions have been measured to be able to judge whether any optimization is an improvement. All timings have been done on HiSun, i.e. a SUN Spark station, using the versions of the `cpu_time` predicate from O'Keefe (p. 81) and the lexicon, type hierarchy and LP rules which are in the same directory. The five LP rules are precompiled to fifteen LP rules by Warshall's algorithm. The lexicon contains four verbs and three nouns, but timings for the difference in the `compile_version_N.pl` files are just given for those entries that actually demand a precompilation. Recall that there is an entry for *sees* as an example for a two place relation, *sells* for a three place and *test* as an imaginary five place relation. The timings for the differences in the `precompile_lp_version_N.pl` files was done using `compile_version_1.pl` in all three cases. Since the time intervals needed for single computations are fairly small, all tests have been done a bigger number of times as well for greater accuracy.

Timings for the `precompile_lp_version_N.pl` files:

	1×	5×	20×
version 1	1.45	7.67	31.55
version 2	1.45	7.70	30.80
version 3	1.45	7.23	29.20

Timings for the `compile_version_N.pl` files:

	sees		sells		test	
	1×	100×	1×	100×	1×	100×
version 1	0.033	2.367	0.117	9.900	0.750	68.550
version 2	0.033	2.418	0.133	10.383	1.250	98.816
version 3	0.034	2.401	0.134	13.450	0.367	37.666

5 Conclusions

To summarize it can be said that even if this program is part of a precompilation step and therefore speed is not all important, it can nevertheless be seen in the timings above that some desired improvements on the speed of the program could be made by applying different techniques such as lagging, first argument indexing and case analysis and by avoiding as many choicepoints as possible, i.e. keep the code as deterministic as possible.

What strikes as odd is the fact that the technique of interleaving the generator and the tester in the third version of the actual compilation program seems to be slower in the cases of short lists and only in the case of the five place list there is a gain; even if this leads to a result that is almost smaller by half. A simple explanation for this might be that the statistics/2 predicate which is used in the cpu_time predicates is too imprecise to measure correctly. But I don't think this is the case since the goals were tried a hundred times each and uncertainties should be averaged out. But at the moment the only other explanation I can come up with is that in the case of the five place relation it is really worth trying to eliminate violations as early as possible whereas in the other cases it does not really matter whether the violation is discovered sooner or later. It might be interesting to check whether these facts hold for other instances of the generate and test paradigm as well or if this effect solely appears in this program. In that case the explanation would have to be found in the code of the program itself.

A Appendix: The Code

A.1 precompile_lp_version_1.pl

```
:- ensure_loaded(process_lex).
:- ensure_loaded(process_lp).
:- ensure_loaded(path_val).
:- ensure_loaded(compile_version_1).
:- ensure_loaded(warshall).

 % precompile(+LP_File,+Lex_File)
 % takes a file with LP rules and a lexicon (all predicates are
 % hardwired for the current troll hpsg type hierarchy) in indentation
 % format, transforms both to prolog format (predicates lp(FS1,FS2) and
 % lex_entry(Word,FS)), computes the transitive closure for the lp
 % rules, writes them into a file in prolog format, then constructs a
 % new lexicon by using the lp rules on the subcat lists of the lexical
 % entries. That means that in the end we have lexical entries whose
 % subcat lists reflects the order of their complements. The new lexicon
 % is written into a file too.

precompile(LP_File,Lex_File):-
        read_lp(LP_File,_LP),
        read_lex(Lex_File),
        closure(NewLP,LP_File),
        cpu_time_fail(1,precompile_aux(Lex_File,NewLP),Duration),
        write(user_output,'Compilation time: '),
        write(user_output,Duration),
        write(user_output,'
                                 '),
        fail.
precompile(_,Lex_File):-
        told,
        rename_file(Lex_File,New_lex_file),
        abolish(lex_entry/2),
        reconsult(New_lex_file),
        write_lex.

precompile_aux(Lex_File,NewLP):-
        rename_file(Lex_File,New_lex_file),
        tell(New_lex_file),
        loop(NewLP).



 % loop(+LP)
 % takes a list of lp rules, finds all lexical entries via a fail loop
 % and processes them to yield new lexical entries.
```

```
loop(LP):-
        !,
        lex_entry(Word,FS),
        do(process(Word,FS,LP)),
        fail.

 % do(+Goal)
 % ensures that in a fail loop no backtracking in the argument goal is
 % possible.

do(X):-
        X,!.

 % process(+Word,+FS,+LPrules)
 % does the actual work. First gets the subcat list from the input FS.
 % we have to catch three cases in which the new entry is identical to
 % the old one: an empty subcat list or one with just one element. Since
 % a list with only one element can have either type ne_list_synsem or
 % list_synsem we have to alternatives for the second case.
 % If we have a subcat list bigger than one with just one element we
 % convert it into prolog list format, compute the possible valid
 % permutations, convert it back to hpsg format and create the new
 % entries.
 % Finally the new entries are written (to a file).
 % The use of naive_path_val/3 may be a problem but I assume that every
 % lex_entry has got a subcat list. we could substitute this with a
 % specific procedure that finds the subcat list.

process(Word,Lex_Entry,LP):-
        naive_path_val([synsem,loc,category,subcat],Lex_Entry,Subcat),
        ( (Subcat = fs(node(e_list,[]),_,_)
                 ;
           Subcat =
                 fs(node(ne_list_synsem,[_,feat(tl,node(e_list,[]))]),_,_)
                 ;
           Subcat =
                 fs(node(list_synsem,[_,feat(tl,node(e_list,[]))]),_,_)
           )
         ->
            New_Entries = [lex_entry(Word,Lex_Entry)]
         ;
           (write(user_output,'Compilation time for  '),
            write(user_output,Word),
            write(user_output,'  :'),
            convert_to_prolog(Subcat,Prolog_Subcat),
            cpu_time(1,compile_lp(Prolog_Subcat,LP,Prolog_Subcats),Dur),
```

```
            write(user_output,Dur),nl(user_output),nl(user_output),
            (Prolog_Subcats = [] ->
                (write(user_output,'The lexical entry for  >>'),
                 write(user_output,Word),
                 write(user_output,'<<  is not compatible with the given
                                     lp rules'),nl,nl,
                 fail)
             ;
                 (turn_to_hpsg_list(Prolog_Subcats,New_Subcats),
%                write_subcats(New_Subcats),
                 make_entries(New_Subcats,lex_entry(Word,Lex_Entry),
                              New_Entries)))
            )),
        write_lex_entries(New_Entries).


 % write_lex_entries(+ListOfEntries)

write_lex_entries([]).
write_lex_entries([H|T]):-
        write(H),write('.'),nl,nl,
        write_lex_entries(T).



 % make_entries(+ListOfValidSubcats,+OldEntry,-ListOfNewEntries)

make_entries([],_,_).
make_entries([H|T],lex_entry(Word,Entry),[lex_entry(Word,NewEntry)|Rest]):-
        make_entry0(Entry,H,NewEntry),
        make_entries(T,lex_entry(Word,Entry),Rest).


 % make_entryN(+OldFS,+Subcat,-NewFS)
 % takes the old feature structure, walks through it via the features to
 % the subcat list and replaces it by the new value and outputs a new
 % feature structure.

make_entry0(fs(node(Type,Feat),RL,DL),fs(Subcat,_,_),
                 fs(node(Type,OutFeat),RL,DL)):-
        extract_member(feat(synsem,Fs),Feat,Before,After),
        make_entry1(Fs,Subcat,NewFeat),
        append(Before,[feat(synsem,NewFeat)|After],OutFeat).
make_entry1(node(Type,Feat),NewSubcat,node(Type,OutFeat)):-
        extract_member(feat(loc,Fs),Feat,Before,After),
        make_entry2(Fs,NewSubcat,NewFeat),
        append(Before,[feat(loc,NewFeat)|After],OutFeat).
```

```
make_entry2(node(Type,Feat),NewSubcat,node(Type,OutFeat)):-
        extract_member(feat(category,Fs),Feat,Before,After),
        make_entry3(Fs,NewSubcat,NewFeat),
        append(Before,[feat(category,NewFeat)|After],OutFeat).
make_entry3(node(Type,Feat),Subcat,node(Type,OutFeat)):-
        extract_member(feat(subcat,_Fs),Feat,Before,After),
        append(Before,[feat(subcat,Subcat)|After],OutFeat).


 % extract_member(+Feat,+FS,-Before,-After)
 % just like member only that it also returns everything before and
 % after the item.

extract_member(X,[X|R],[],R).
extract_member(X,[Y|R],[Y|R1],After) :-
        extract_member(X,R,R1,After).




 % turn_to_prolog_list(+ListOfHpsgLists,-ListOfPrologLists)

turn_to_prolog_list([],[]).
turn_to_prolog_list([H|T],[Subcat|Rest]):-
        convert_to_prolog(H,Subcat),
        turn_to_prolog_list(T,Rest).

 % convert_to_prolog(+FS,-PrologList)
 % takes the HD and makes it into an element of the prolog list,
 % recurses down through the TL.
 % The use of naive_path_val/3 may be a problem but I assume that every
 % lex_entry has got a hd and a tl. we could substitute this with a
 % specific procedure.

convert_to_prolog(fs(node(e_list,_),_,_),[]).
convert_to_prolog(FS,[FS1|Rest]):-
        naive_path_val([hd],FS,FS1),
        naive_path_val([tl],FS,FS2),
        convert_to_prolog(FS2,Rest).



 % turn_to_hpsg_list(+ListOfPrologLists,-ListOfHpsgLists)

turn_to_hpsg_list([],[]).
turn_to_hpsg_list([H|T],[fs(FS,RL,DL)|Rest]):-
        convert_to_hpsg(H,FS,RL,DL),
        turn_to_hpsg_list(T,Rest).
```

```
 % convert_to_hpsg(+PrologList,-HPSGList)

convert_to_hpsg([],node(e_list,[]),_,_).
convert_to_hpsg([fs(FS,RL,DL)|FSs],
                node(ne_list_synsem,[feat(hd,FS),feat(tl,FS1)]),
                RL,DL):-
        convert_to_hpsg(FSs,FS1,_,_).


 % write_subcats(+ListOfHpsgSubcats)

write_subcats([]).
write_subcats([H|T]):-
        pretty_print(H),
        write_subcats(T).


% write_lex(+ListOfLexEntries)

write_lex([]).
%write_lex([new_lex_entry(Word,FS)|T]):-
%       write('word: '),
%       write(Word),nl,
%       pretty_print(FS),nl,nl,
%       write_lex(T).
write_lex([lex_entry(Word,FS)|T]):-
        write('word: '),
        write(Word),nl,
        pretty_print(FS),nl,nl,
        write_lex(T).


 % write_lex
 % writes all lex entries via a fail loop

write_lex:-
        lex_entry(Word,FS),
        do((write('word: '),
        write(Word),nl,
        pretty_print(FS),nl,nl)),
        fail.
write_lex.
```

A.2 precompile_lp_version_2.pl

```
:- ensure_loaded(process_lex).
:- ensure_loaded(process_lp).
:- ensure_loaded(path_val).
:- ensure_loaded(compile_version_1).
:- ensure_loaded(warshall).

 % precompile(+LP_File,+Lex_File)
 % takes a file with LP rules and a lexicon (all predicates are
 % hardwired for the current troll hpsg type hierarchy) in indentation
 % format, transforms both to prolog format (predicates lp(FS1,FS2) and
 % lex_entry(Word,FS)), computes the transitive closure for the lp
 % rules, writes them into a file in prolog format, then constructs a
 % new lexicon by using the lp rules on the subcat lists of the lexical
 % entries. That means that in the end we have lexical entries whose
 % subcat lists reflects the order of their complements. The new lexicon
 % is wriverbatimen into a file too.

precompile(LP_File,Lex_File):-
        read_lp(LP_File,_LP),
        read_lex(Lex_File),
        closure(NewLP,LP_File),
        cpu_time_fail(20,precompile_aux(Lex_File,NewLP),Duration),
        write(user_output,'Compilation time: '),
        write(user_output,Duration),
        write(user_output,'
                  '),fail.
precompile(_,Lex_File):-
        told,
        rename_file(Lex_File,New_lex_file),
        abolish(lex_entry/2),
        reconsult(New_lex_file),
        write_lex.

precompile_aux(Lex_File,NewLP):-
        rename_file(Lex_File,New_lex_file),
        tell(New_lex_file),
        loop(NewLP).




 % loop(+LP)
 % takes a list of lp rules, finds all lexical entries via a fail loop
 % and processes them to yield new lexical entries.
```

```
loop(LP):-
        !,
        lex_entry(Word,FS),
        do(process(Word,FS,LP)),
        fail.


 % do(+Goal)
 % ensures that in a fail loop no backtracking in the argument goal is
 % possible.

do(X):-
        X,!.


 % process(+Word,+FS,+LPrules)
 % does the actual work. First gets the subcat list from the input FS.
 % we have to catch three cases in which the new entry is identical to
 % the old one: an empty subcat list or one with just one element. Since
 % a list with only one element can have either type ne_list_synsem or
 % list_synsem we have two alternatives for the second case.
 % If we have a subcat list bigger than one with just one element we
 % convert it into prolog list format, compute the possible valid
 % permutations, convert it back to hpsg format and create the new
 % entries.
 % Finally the new entries are written (to a file).
 % The use of naive_path_val/3 may be a problem but I assume that every
 % lex_entry has got a subcat list. we could substitute this with a
 % specific procedure that finds the subcat list.

process(Word,FS,LP):-
        naive_path_val([synsem,loc,category,subcat],FS,Subcat),
        (   necessary(Subcat)
           ->
            New_Entries = [lex_entry(Word,FS)]
         ;
           (write(user_output,'Compilation time for  '),
            write(user_output,Word),
            write(user_output,'  :'),
            convert_to_prolog(Subcat,Prolog_Subcat),
            cpu_time(1,compile_lp(Prolog_Subcat,LP,Prolog_Subcats),Dur),
            write(user_output,Dur),nl(user_output),nl(user_output),
            (Prolog_Subcats = [] ->
                (write(user_output,'The lexical entry for >>'),
                 write(user_output,Word),
                 write(user_output,'<<  is not compatible with the given
                                        lp rules'),nl,nl,
                 fail)
```

```
             ;
                 (turn_to_hpsg_list(Prolog_Subcats,New_Subcats),
%                write_subcats(New_Subcats),
                 make_entries(New_Subcats,lex_entry(Word,FS),
                              New_Entries)))
             )),
        write_lex_entries(New_Entries).

 % necessary(+Subcat)
 % takes a feature structure in form of a hpsg (subcat) list and fails
 % if the list has more than one element.

necessary(fs(node(Type,List),_,_)):-
        necessary_aux(Type,List).
necessary_aux(e_list,[]).
necessary_aux(list_synsem,_).
necessary_aux(ne_list_synsem,[_,feat(tl,node(e_list,[]))]).


 % make_entries(+ListOfValidSubcats,+OldEntry,-ListOfNewEntries)

make_entries([],_,[]).
make_entries([H|T],lex_entry(Word,Entry),[lex_entry(Word,NewEntry)|Rest]):-
        make_entry0(Entry,H,NewEntry),
        make_entries(T,lex_entry(Word,Entry),Rest).


 % make_entryN(+OldFS,+Subcat,-NewFS)
 % takes the old feature structure, walks through it via the features to
 % the subcat list and replaces it by the new value and outputs a new
 % feature structure.

make_entry0(fs(node(Type,Feat),RL,DL),fs(Subcat,_,_),
                 fs(node(Type,OutFeat),RL,DL)):-
        extract_member(Before,feat(synsem,Fs),Feat,After),
        make_entry1(Fs,Subcat,NewFeat),
        extract_member(Before,feat(synsem,NewFeat),OutFeat,After).
make_entry1(node(Type,Feat),NewSubcat,node(Type,OutFeat)):-
        extract_member(Before,feat(loc,Fs),Feat,After),
        make_entry2(Fs,NewSubcat,NewFeat),
        extract_member(Before,feat(loc,NewFeat),OutFeat,After).
make_entry2(node(Type,Feat),NewSubcat,node(Type,OutFeat)):-
        extract_member(Before,feat(category,Fs),Feat,After),
        make_entry3(Fs,NewSubcat,NewFeat),
        extract_member(Before,feat(category,NewFeat),OutFeat,After).
make_entry3(node(Type,Feat),Subcat,node(Type,OutFeat)):-
        extract_member(Before,feat(subcat,_Fs),Feat,After),
        extract_member(Before,feat(subcat,Subcat),OutFeat,After).
```

```
 % extract_member(?Before,+Feat,?FS,?After)
 % just like member only that it also returns everything before and
 % after the item.

extract_member([],X,[X|R],R):-!.
extract_member([Y|R1],X,[Y|R],After) :-
        extract_member(R1,X,R,After).


 % turn_to_prolog_list(+ListOfHpsgLists,-ListOfPrologLists)

turn_to_prolog_list([],[]).
turn_to_prolog_list([H|T],[Subcat|Rest]):-
        convert_to_prolog(H,Subcat),
        turn_to_prolog_list(T,Rest).


% convert_to_prolog(+FS,-PrologList)
 % takes the HD and makes it into an element of the prolog list,
 % recurses down through the TL.
 % The use of naive_path_val/3 may be a problem but I assume that every
 % lex_entry has got a hd and a tl. we could substitute this with a
 % specific procedure.

convert_to_prolog(fs(node(e_list,_),_,_),[]).
convert_to_prolog(FS,[FS1|Rest]):-
        naive_path_val([hd],FS,FS1),
        naive_path_val([tl],FS,FS2),
        convert_to_prolog(FS2,Rest).

 % turn_to_hpsg_list(+ListOfPrologLists,-ListOfHpsgLists)

turn_to_hpsg_list([],[]).
turn_to_hpsg_list([H|T],[fs(FS,RL,DL)|Rest]):-
        convert_to_hpsg(H,FS,RL,DL),
        turn_to_hpsg_list(T,Rest).


 % convert_to_hpsg(+PrologList,-HPSGList)

convert_to_hpsg([],node(e_list,[]),_,_).
convert_to_hpsg([fs(FS,RL,DL)|FSs],
                node(ne_list_synsem,[feat(hd,FS),feat(tl,FS1)]),
                RL,DL):-
        convert_to_hpsg(FSs,FS1,_,_).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% some predicates used for writing

% write_subcats(+ListOfHpsgSubcats)

write_subcats([]).
write_subcats([H|T]):-
        pretty_print(H),
        write_subcats(T).


% write_lex_entries(+ListOfEntries)

write_lex_entries([]).
write_lex_entries([H|T]):-
        write(H),write('.'),nl,nl,
        write_lex_entries(T).


% write_lex(+ListOfLexEntries)

write_lex([]).
% write_lex([new_lex_entry(Word,FS)|T]):-
%       write('word: '),
%       write(Word),nl,
%       pretty_print(FS),nl,nl,
%       write_lex(T).
write_lex([lex_entry(Word,FS)|T]):-
        write('word: '),
        write(Word),nl,
        pretty_print(FS),nl,nl,
        write_lex(T).


% write_lex
% writes all lex entries via a fail loop

write_lex:-
        lex_entry(Word,FS),
        do((write('word: '),
        write(Word),nl,
        pretty_print(FS),nl,nl)),
        fail.
write_lex.
```

A.3 precompile_lp_version_3.pl

```
:- ensure_loaded(process_lex).
:- ensure_loaded(process_lp).
:- ensure_loaded(path_val).
:- ensure_loaded(compile_version_1).
:- ensure_loaded(warshall).
:- ensure_loaded(type_union4).

 % precompile(+LP_File,+Lex_File)
 % takes a file with LP rules and a lexicon (all predicates are
 % hardwired for the current troll hpsg type hierarchy) in indentation
 % format, transforms both to prolog format (predicates lp(FS1,FS2) and
 % lex_entry(Word,FS)), computes the transitive closure for the lp
 % rules, writes them into a file in prolog format, then constructs a
 % new lexicon by using the lp rules on the subcat lists of the lexical
 % entries. That means that in the end we have lexical entries whose
 % subcat lists reflects the order of their complements. The new lexicon
 % is written into a file too.

precompile(LP_File,Lex_File):-
        read_lp(LP_File,_LP),
        read_lex(Lex_File),
        closure(NewLP,LP_File),
        cpu_time_fail(1,precompile_aux(Lex_File,NewLP),Duration),
        write(user_output,'Compilation time: '),
        write(user_output,Duration),
        write(user_output,'
                  '),fail.
precompile(_,Lex_File):-
        told,
        rename_file(Lex_File,New_lex_file),
        abolish(lex_entry/2),
        reconsult(New_lex_file)
%       ,write_lex
        .

precompile_aux(Lex_File,NewLP):-
        rename_file(Lex_File,New_lex_file),
        tell(New_lex_file),
        loop(NewLP).



 % loop(+LP)
 % takes a list of lp rules, finds all lexical entries via a fail loop
 % and processes them to yield new lexical entries.
```

```
loop(LP):-
        !,
        lex_entry(Word,FS),
        do(process(Word,FS,LP)),
        fail.


 % do(+Goal)
 % ensures that in a fail loop no backtracking in the argument goal is
 % possible.

do(X):-
        X,!.


 % process(+Word,+FS,+LPrules)
 % does the actual work. First gets the subcat list from the input FS.
 % we have to catch three cases in which the new entry is identical to
 % the old one: an empty subcat list or one with just one element. Since
 % a list with only one element can have either type ne_list_synsem or
 % list_synsem we have to alternatives for the second case.
 % If we have a subcat list bigger than one with just one element we
 % convert it into prolog list format, compute the possible valid
 % permutations, convert it back to hpsg format and create the new
 % entries.
 % Finally the new entries are written (to a file).

process(Word,FS,LP):-
     (extract_subcat(FS,Subcat,OutFS) ->
        (   necessary(Subcat)
           ->
            New_Entries = [lex_entry(Word,OutFS)]
         ;
           (write(user_output,'Compilation time for  '),
            write(user_output,Word),
            write(user_output,'  :'),
            convert_to_prolog(Subcat,Prolog_Subcat),
            cpu_time(1,compile_lp(Prolog_Subcat,LP,Prolog_Subcats),Dur),
            write(user_output,Dur),nl(user_output),nl(user_output),
            (Prolog_Subcats = [] ->
                (write(user_output,'The lexical entry for >>'),
                 write(user_output,Word),
                 write(user_output,'<<  is not compatible with the given
                                       lp rules'),nl,nl,
                 fail)
             ;
                 (turn_to_hpsg_list(Prolog_Subcats,New_Subcats),
%                write_subcats(New_Subcats),
```

```
                make_entries(New_Subcats,lex_entry(Word,OutFS),
                             New_Entries)))
           ))

    ;   write(user_output,'No subcat list found for '),
        write(user_output,Word),nl,nl,
        New_Entries = [lex_entry(Word,FS)]),
    write_lex_entries(New_Entries).


 % extract_subcat(InFS,Subcat,OutFS)
 % runs through InFS to find the Subcat list thereby possibly
 % dereferencing chains and producing a different RL changing InFS to
 % OutFS. That the Subcat list is treated as a feature structure on its
 % own is connected with convert_to_prolog/2 which needs the RL as well
 % as the Subcat list.

extract_subcat(fs(FS,RL,DL),fs(Subcat,OutRL,DL),fs(FS,OutRL,DL)):-
        extract_synsem_aux(FS,RL,OutRL,Subcat).

 % extract_Feature_aux(Node,InRL,OutRL,Subcat)
 % finds the Feature, dereferences it if possible and continues with the
 % next feature

extract_synsem_aux(node(_,Feat),RL,OutRL,Subcat):-
        find_synsem(Feat,Synsem1),
        deref(Synsem1,RL,RL1,Val),
        ( Val = rnum(_) ->
            fail
        ;   ( Val = re(_,Node) ->
                extract_loc_aux(Node,RL1,OutRL,Subcat)
            ;   extract_loc_aux(Val,RL1,OutRL,Subcat))).
extract_loc_aux(node(_,Feat),RL,OutRL,Subcat):-
        find_loc(Feat,Loc1),
        deref(Loc1,RL,RL1,Val),
        ( Val = rnum(_) ->
            fail
        ;   ( Val = re(_,Node) ->
                extract_category_aux(Node,RL1,OutRL,Subcat)
            ;   extract_category_aux(Val,RL1,OutRL,Subcat))).
extract_category_aux(node(_,Feat),RL,OutRL,Subcat):-
        find_category(Feat,Category1),
        deref(Category1,RL,RL1,Val),
        ( Val = rnum(_) ->
            fail
        ;   ( Val = re(_,Node) ->
                extract_subcat_aux(Node,RL1,OutRL,Subcat)
            ;   extract_subcat_aux(Val,RL1,OutRL,Subcat))).
```

```
extract_subcat_aux(node(_,Feat),RL,OutRL,Subcat):-
        find_subcat(Feat,Subcat1),
        deref(Subcat1,RL,OutRL,Val),
        ( Val = rnum(_) ->
            fail
        ;   ( Val = re(_,Node) ->
                Subcat = Node
            ;   Subcat = Val)).


 % find_Feature(FeatList,Node)
 % looks for Feature in FeatList, returns the relevant node; fails if
 % the feature can't be found.

find_synsem([feat(synsem,Node)|_],Node).
find_synsem([feat(Feat,_)|Rest],Node):-
        (Feat @< synsem ->
            find_synsem(Rest,Node)
        ;   fail).
find_loc([feat(loc,Node)|_],Node).
find_loc([feat(Feat,_)|Rest],Node):-
        (Feat @< loc ->
            find_loc(Rest,Node)
        ;   fail).
find_category([feat(category,Node)|_],Node).
find_category([feat(Feat,_)|Rest],Node):-
        (Feat @< category ->
            find_category(Rest,Node)
        ;   fail).
find_subcat([feat(subcat,Node)|_],Node).
find_subcat([feat(Feat,_)|Rest],Node):-
        (Feat @< subcat ->
            find_subcat(Rest,Node)
        ;   fail).


 % necessary(+Subcat)
 % takes a feature structure in form of a hpsg (subcat) list and fails
 % if the list has more than one element.

necessary(fs(node(Type,List),_,_)):-
        necessary_aux(Type,List).
necessary_aux(e_list,[]).
necessary_aux(list_synsem,_).
necessary_aux(ne_list_synsem,[_,feat(tl,node(e_list,[]))]).
```

```
 % make_entries(+ListOfValidSubcats,+OldEntry,-ListOfNewEntries)

make_entries([],_,[]).
make_entries([H|T],lex_entry(Word,Entry),[lex_entry(Word,NewEntry)|Rest]):-
        make_entry(Entry,H,NewEntry),
        make_entries(T,lex_entry(Word,Entry),Rest).


 % make_entry(+OldFS,+Subcat,-NewFS)
 % takes the old feature structure, walks through it via the features to
 % the subcat list and replaces it by the new value and outputs a new
 % feature structure.
 % partially evaluated version of make_entryN/3

make_entry(fs(node(Type,Feat),RL,DL),fs(Subcat,_,_),
                 fs(node(Type,OutFeat1),RL,DL)):-
        extract_member(Before1,feat(synsem,node(Type1,Feat1)),Feat,After1),
        extract_member(Before2,feat(loc,node(Type2,Feat2)),Feat1,After2),
        extract_member(Before3,feat(category,node(Type3,Feat3)),Feat2,After3),
        extract_member(Before4,feat(subcat,_Fs),Feat3,After4),
        extract_member(Before4,feat(subcat,Subcat),OutFeat4,After4),
        extract_member(Before3,feat(category,node(Type3,OutFeat4)),OutFeat3,
                       After3),
        extract_member(Before2,feat(loc,node(Type2,OutFeat3)),OutFeat2,After2),
        extract_member(Before1,feat(synsem,node(Type1,OutFeat2)),OutFeat1,
                       After1).


 % extract_member(?Before,+Feat,?FS,?After)
 % just like member only that it also returns everything before and
 % after the item.

extract_member([],X,[X|R],R):-!.
extract_member([Y|R1],X,[Y|R],After) :-
        extract_member(R1,X,R,After).


 % turn_to_prolog_list(+ListOfHpsgLists,-ListOfPrologLists)

turn_to_prolog_list([],[]).
turn_to_prolog_list([H|T],[Subcat|Rest]):-
        convert_to_prolog(H,Subcat),
        turn_to_prolog_list(T,Rest).
```

```
 % convert_to_prolog(+HPSGList,-PrologList)

convert_to_prolog(fs(node(_Type,FeatList),RL,DL),Result):-
        convert_to_prolog_aux(FeatList,RL,DL,Result).


 % convert_to_prolog_aux(+ListOfHD/TLFeat,+RL,+DL,-PrologList)
 % takes the HD and makes it into an element of the prolog list,
 % recurses down through the rest. Works without naive_path_val/3.

convert_to_prolog_aux([],_,_,[]).
convert_to_prolog_aux([feat(hd,FS1),feat(tl,node(_Type,FeatList))],
                      RL,DL,[fs(FS1,RL,DL)|Rest]):-
        convert_to_prolog_aux(FeatList,RL,DL,Rest).


 % turn_to_hpsg_list(+ListOfPrologLists,-ListOfHpsgLists)

turn_to_hpsg_list([],[]).
turn_to_hpsg_list([H|T],[fs(FS,RL,DL)|Rest]):-
        convert_to_hpsg(H,FS,RL,DL),
        turn_to_hpsg_list(T,Rest).


 % convert_to_hpsg(+PrologList,-HPSGList)

convert_to_hpsg([],node(e_list,[]),_,_).
convert_to_hpsg([fs(FS,RL,DL)|FSs],
                node(ne_list_synsem,[feat(hd,FS),feat(tl,FS1)]),
                RL,DL):-
        convert_to_hpsg(FSs,FS1,_,_).



 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 % some predicates used for writing

 % write_subcats(+ListOfHpsgSubcats)

write_subcats([]).
write_subcats([H|T]):-
        pretty_print(H),
        write_subcats(T).


 % write_lex_entries(+ListOfEntries)

write_lex_entries([]).
```

```
write_lex_entries([H|T]):-
        write(H),write('.'),nl,nl,
        write_lex_entries(T).


 % write_lex(+ListOfLexEntries)

write_lex([]).
 % write_lex([new_lex_entry(Word,FS)|T]):-
 %      write('word: '),
 %      write(Word),nl,
 %      pretty_print(FS),nl,nl,
 %      write_lex(T).
write_lex([lex_entry(Word,FS)|T]):-
        write('word: '),
        write(Word),nl,
        pretty_print(FS),nl,nl,
        write_lex(T).


 % write_lex
 % writes all lex entries via a fail loop

write_lex:-
        lex_entry(Word,FS),
        do((write('word: '),
        write(Word),nl,
        pretty_print(FS),nl,nl)),
        fail.
write_lex.
```

A.4 compile_version_1.pl

```
:- use_module(library(lists),[permutation/2,permutation/3,insert/3]).
:- ensure_loaded(type_union4).

 % compile_lp(+SubcatList,+LPrules,-ListOfSubcatLists)
 % finds all permutations of the subcat list that are lp acceptable

compile_lp([],_LP,[]).
compile_lp(Subcat,LP,Result):-
        (findall(NewSubcat,
                (permutation(Subcat,NewSubcat),
                 lp_acceptable(LP,NewSubcat)),
                Result) ->
            true
            ;
            Result = []).
```

```
% lp_acceptable(+ListOfLPRules,+SubcatList)
% recurses down the lp list
%
% lp_acceptable([],_Subcat).
% lp_acceptable([LPrule|T],Subcat):-
%      lp_aux(Subcat,LPrule),
%      lp_acceptable(T,Subcat).
%
%
% lp_aux(+SubcatList,+LPRule)
% recurses through the subcat list
%
% lp_aux([_],_).
% lp_aux([H|T],LP):-
%      lp_aux1(T,H,LP),
%      lp_aux(T,LP).
%
%
% lp_aux1(+RestList,+FirstOfList,+LPrule)
% checks whether First and some elemnt from Rest unify with the
% elements of the lp rule. If both do in a certain way, we have a
% violation of this lp rule.
%
% lp_aux1([],_,_LP).
% lp_aux1([Next|T],H,lp(Fs1,Fs2)):-
%      \+ (unify_fs(Fs1,Next,_),
%          unify_fs(Fs2,H,_)),
%      lp_aux1(T,H,lp(Fs1,Fs2)).
%

% Improved by lagging to use first argument indexing since the old
% lp_aux/2 was not deterministic
%
% lp_aux([H|T],LPrule):-
%      lp_aux_0(T,H,LPrule).
%
% lp_aux_0([],_Last,_LP).
% lp_aux_0([Second|T],First,LP):-
%      lp_aux_1(T,First,Second,LP),
%      lp_aux_0(T,Second,LP).
%
% lp_aux_1([],First,Second,lp(Fs1,Fs2)):-
%      \+ (unify_fs(Fs1,Second,_),
%          unify_fs(Fs2,First,_)).
```

```
% lp_aux_1([Next|T],First,Second,lp(Fs1,Fs2)):-
%      \+ (unify_fs(Fs1,Second,_),
%          unify_fs(Fs2,First,_)),
%      lp_aux_1(T,First,Next,lp(Fs1,Fs2)).
%


% Improved by unpacking the lp rule and testing whether it is worth
% trying to find a violation at all.
%
% lp_acceptable([],_Subcat).
% lp_acceptable([lp(FS1,FS2)|T],Subcat):-
%      lp_aux(Subcat,FS1,FS2),
%      lp_acceptable(T,Subcat).
%
% lp_aux([],_,_).
% lp_aux([H|T],FS1,FS2):-
%      (unify_fs(H,FS2,_) ->
%          lp_aux_0(T,H,FS1,FS2)
%      ;
%          lp_aux(T,FS1,FS2)
%      ).
%
% lp_aux_0([],_Last,_LP).
% lp_aux_0([Second|T],First,LP):-
%      lp_aux_1(T,First,Second,LP),
%      lp_aux_0(T,Second,LP).
%
% lp_aux_1([],First,Second,lp(Fs1,Fs2)):-
%      \+ (unify_fs(Fs1,Second,_),
%          unify_fs(Fs2,First,_)).
% lp_aux_1([Next|T],First,Second,lp(Fs1,Fs2)):-
%      \+ (unify_fs(Fs1,Second,_),
%          unify_fs(Fs2,First,_)),
%      lp_aux_1(T,First,Next,lp(Fs1,Fs2)).
%
%

% further improvement by case analysis. Only if the first element of
% the list matches the second part of the lp statement, we have to
% check whether nothing else in the list does.
% Also improved by further lagging to stop lp_aux/4 as early as
% possible (only one element left).

lp_acceptable([],_Subcat).
lp_acceptable(LP,[H|T]):-
        lp_acceptable_0(LP,H,T).
```

```
lp_acceptable_0([],_First,_Rest).
lp_acceptable_0([lp(FS1,FS2)|T],First,Rest):-
        lp_aux(Rest,First,FS1,FS2),
        lp_acceptable_0(T,First,Rest).


lp_aux([],_Last,_,_).
lp_aux([H|T],First,FS1,FS2):-
        (unify_fs(First,FS2,_) ->
            lp_aux_0(T,H,FS1)
        ;
            lp_aux(T,H,FS1,FS2)
        ).

lp_aux_0([],H,FS1):-
        \+ unify_fs(H,FS1,_).
lp_aux_0([Second|T],First,FS1):-
        \+ unify_fs(First,FS1,_),
        lp_aux_0(T,Second,FS1).
```

A.5 compile_version_2.pl

```
:- use_module(library(lists),[permutation/2,permutation/3,insert/3]).
:- ensure_loaded(type_union4).
:- ensure_loaded(utilities).


 % O'Keefe p.204: filtering the elements of a list that pass a certain
 % test, in this case LP acceptability.

 % compile_lp(+SubcatList,+LPrules,-ListOfSubcatLists)
 % finds all permutations of the subcat list and checks afterwards which
 % are lp acceptable

compile_lp([],_LP,[]).
compile_lp(Subcat,LP,ValidSubcat):-
        findall(NewSubcat,
                permutation(Subcat,NewSubcat),
                Result),
        lp_acceptable_1(Result,LP,ValidSubcat).

% first intermediate step
% lp_acceptable1(List,Test_LP,Sublist):-
%       sublist(List,Test_LP,X,X,Sublist,[]).
%
```

```
% sublist([],_LP,R0,R,R0,R).
% sublist([H|T],Test_LP,L0,L,R0,R):-
%       filter(Test_LP,H,L0,L,M0,M),
%       sublist(T,Test_LP,M0,M,R0,R).
%
% filter(Test_LP,H,,M0,[H|M],M0,M):-
%       lp_acceptable(Test_LP,H),!.
% filter(_,_,M0,M,M0,M).

% second intermediate step
% lp_acceptable1(List,Test_LP,Sublist):-
%       sublist(List,Test_LP,Sublist).
%
% sublist([],_LP,[]).
% sublist([H|T],Test_LP,Sublist0):-
%       filter(Test_LP,H,Sublist0,Sublist1),
%       sublist(T,Test_LP,Sublist1).
%
% filter(Test_LP,H,,[H|M],M):-
%       lp_acceptable(Test_LP,H),!.
% filter(_,_,M,M).


lp_acceptable_1(List,Test_LP,Sublist):-
        sublist(List,Test_LP,Sublist).

sublist([],_LP,[]).
sublist([H|T],Test_LP,Sublist0):-
        lp_acceptable(Test_LP,H),
        !,
        Sublist0 = [H|Sublist1],
        sublist(T,Test_LP,Sublist1).
sublist([_|T],Test_LP,Sublist):-
        sublist(T,Test_LP,Sublist).


 % for comments on this code, see compile_version_1.pl

lp_acceptable([],_Subcat).
lp_acceptable(LP,[H|T]):-
        lp_acceptable_0(LP,H,T).

lp_acceptable_0([],_First,_Rest).
lp_acceptable_0([lp(FS1,FS2)|T],First,Rest):-
        lp_aux(Rest,First,FS1,FS2),
        lp_acceptable_0(T,First,Rest).
```

```
lp_aux([],_Last,_,_).
lp_aux([H|T],First,FS1,FS2):-
        (unify_fs(First,FS2,_) ->
            lp_aux_0(T,H,FS1)
        ;
            lp_aux(T,H,FS1,FS2)
        ).

lp_aux_0([],H,FS1):-
        \+ unify_fs(H,FS1,_).
lp_aux_0([Second|T],First,FS1):-
        \+ unify_fs(First,FS1,_),
        lp_aux_0(T,Second,FS1).
```

A.6 compile_version_3.pl

```
:- ensure_loaded(type_union4).
:- ensure_loaded(utilities).


 % next try: interleave  permutation and LP acceptability

 % compile_lp(+SubcatList,+LPrules,-ListOfSubcatLists)
 % finds all permutations of the subcat list that are lp acceptable,
 % does this while constructing the permutations

compile_lp([],_LP,[]).
compile_lp(Subcat,LP,Result):-
        findall(NewSubcat,
                permutation_and_lp(Subcat,NewSubcat,LP),
                Result).


 % permutation_and_lp/4, insert_if_lp/4 are variations of the code found
 % in library(lists)

permutation_and_lp(List, Perm, LP) :-
        permutation_and_lp(List, Perm, Perm, LP).

permutation_and_lp([], [], [], _).
permutation_and_lp([X|Xs], Ys1, [_|Zs], LP) :-
        permutation_and_lp(Xs, Ys, Zs ,LP),
        insert_if_lp(Ys, X, Ys1, LP).

insert_if_lp(L, X, [X|L], LP):-
        lp_acceptable(LP,X,L).
```

```
insert_if_lp([H|T], X, [H|L],LP) :-
        insert_if_lp(T, X, L,LP),
        lp_acceptable(LP,H,L).


 % lp_acceptable(LP_Rules,FirstOfSubcat,RestOfSubcat)
 % First and Rest are separate arguments because insert_if_lp/4 needs
 % them this way and so I keep it that way
 % unpacks the FSs from the lp wrapper

lp_acceptable([],_First,_Rest).
lp_acceptable([lp(FS1,FS2)|T],First,Rest):-
        lp_aux_1(Rest,First,FS1,FS2),
        lp_acceptable(T,First,Rest).


 % lp_aux_1(RestList,First,FS1,FS2)
 % checks whether First unifies with the second FS2 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 unifies with FS1 ->
 % violation of the lp rule.
 % To make this deterministic, I wrote [H|T] in lp_aux_1 and LP_aux
 % though it would have been easier to have just one variable in this
 % place.

lp_aux_1([],_,_,_).
lp_aux_1([H|T],First,FS1,FS2):-
        (unify_fs(First,FS2,_) ->
            lp_aux([H|T],FS1)
        ;
            true
        ).

 % lp_aux(SubcatList,FS)
 % checks whether an element of SubcatList unifies with FS -> fails in
 % this case since a violation of an LP rule has occured.

lp_aux([],_FS1).
lp_aux([Next|T],FS1):-
        \+ unify_fs(Next,FS1,_),
        lp_aux(T,FS1).
```