Research Report AI-1993-03

The Enhancement of a Dependency Parser for Latin

Ulrich Koch

Artificial Intelligence Programs
The University of Georgia
Athens, Georgia 30602–7415 U.S.A.

The enhancement of a dependency parser for Latin

Ulrich Koch

Winter 1993

Artificial Intelligence Programs
University of Georgia
Athens, GA 30602

Reprinted with corrections, February 1994

Introduction

This paper describes a dependency parser for a tiny subset of Latin. It is the enhancement of a parser that is described in [Covington 1990a]. The parsing algorithm can be used for parsing variable-word-order languages other than Latin. Information about variable-word-order languages can be found in [Covington 1990b].

The parser is written in GULP [Covington 1987, Covington 1989], which extends Prolog by providing feature structure unification.

The features that I have added to the parser include verb subcategorization and checks for a minimal and/or maximal number of dependencies of certain types.

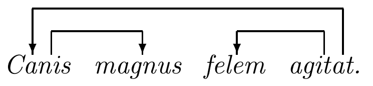
1 Dependency grammar

The traditional approach to parsing is phrase-structure grammar, which analyzes a sentence by dividing it into constituents. The constituents are divided in turn, until we arrive at lexical constituents, i.e. single words.

Dependency grammar [Tesnière 1959, Schubert 1987, Mel'čuk 1988], by contrast, does not rely on the notion of constituents.[1] Instead, the main notion is that of a dependency relation between two words. This relation is not symmetric; one word (called the "dependent") is said to depend on the other one (the "governor" or "head").

Because only word-to-word links are established, dependency grammar can easily handle variable word order and discontinuous constituents. Phrase-structure grammar is not well suited to handling these phenomena.

Dependency relations can be represented graphically by drawing an arrow from the governor to the dependent, like in the following example.



Dog big cat chases.

'The big dog chases the cat.'

If we put each governor above its dependent(s), we get a tree-like representation:

[1] However, constituents can easily be defined in terms of dependency.

$$\begin{tikzcd}
 & \text{agitat} \arrow[dl] \arrow[dr] & \\
\text{canis} \arrow[d] & & \text{felem} \\
\text{magnus} & &
\end{tikzcd}$$

Such a tree can be printed out on the computer in indented form:

```
agitat
  canis
    magnus
  felem
```

In the above examples, *agitat* is the main verb of the sentence, and *canis* is the subject. Accordingly, the dependency relation between these two words is said to be of type SUBJECT. Another example of a dependency type is MODIFIER, which is used for the relation between a noun and an adjective that modifies it. In graphical representations of dependency analysis, dependency types can be attached to the links:

$$\begin{tikzcd}
\textit{Canis} & \textit{dormit.} \arrow[l, bend right, "\text{SUBJ}"']
\end{tikzcd}$$

Dog sleeps.

'The dog sleeps.'

Not every dependency type is possible for each syntactic category[2]. For instance, only a verb can have a subject.

2 Feature structure unification

The variety of dependency grammar that is used by the parser is unification-based [Shieber 1986]. That is, categories are represented as feature structures, and features are assigned values by unification.

2.1 Feature structures

A feature structure is a set of feature-value pairs. A feature is just a name, where the names of all the features in a feature structure must be distinct. A value can be anything from another name or a number to another feature structure. Here are some examples of feature structures:

$$\begin{bmatrix} \textit{a:x} \\ \textit{b:y} \\ \textit{c:z} \end{bmatrix} \quad \begin{bmatrix} \textit{height:5}'\,\textit{11}'' \\ \textit{weight:123 lbs.} \\ \textit{age:24} \end{bmatrix} \quad \begin{bmatrix} \textit{cat:verb} & \\ \textit{object:} & \begin{bmatrix} \textit{cat:noun} \\ \textit{case:acc} \end{bmatrix} \end{bmatrix}$$

The order of feature-value pairs within a feature structure is not important. For instance, the following two feature structures are considered the same:

$$\begin{bmatrix} \textit{a:x} \\ \textit{b:y} \\ \textit{c:z} \end{bmatrix} \quad \begin{bmatrix} \textit{c:z} \\ \textit{a:x} \\ \textit{b:y} \end{bmatrix}$$

In GULP, the value of a feature can be any Prolog term, or another feature structure, or a Prolog term containing feature structures. GULP uses the functor '`::`' to represent feature structures. For example, the feature structure

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \\ \textit{num:pl} \end{bmatrix}$$

[2]Syntactic categories include verb, noun, adjective, verb phrase, and noun phrase.

looks like this in GULP:

```
cat:verb :: pers:2 :: num:pl
```

2.2 Unification without variables

Unification is an operation that, when performed on two feature structures, yields a third feature structure. It can also be viewed as a way of testing if two feature structures match in a certain way. This test can succeed or fail. The unification operator is written as '$\sqcup$'.

In the simplest case of unification, all the features in the feature structures are distinct. Unification is then reduced to set union, as in the following example.

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \end{bmatrix} \sqcup \begin{bmatrix} \textit{num:pl} \end{bmatrix} = \begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \\ \textit{num:pl} \end{bmatrix}$$

It works the same way if feature-value pairs are doubled in the second feature structure:

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \end{bmatrix} \sqcup \begin{bmatrix} \textit{cat:verb} \\ \textit{num:pl} \end{bmatrix} = \begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \\ \textit{num:pl} \end{bmatrix}$$

Unification fails if the two feature structures contain the same feature with different values, as in this example.

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \end{bmatrix} \sqcup \begin{bmatrix} \textit{cat:noun} \\ \textit{num:pl} \end{bmatrix} \text{fails!}$$

2.3 Unification with variables

So far, the values of the features have been constants. However, variables[3] can take their place. If a feature f occurs with value a in one feature structure and with value X in the other feature structure, then unification assigns a to X.

In the following example, the variable C is set to *verb:*

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \end{bmatrix} \sqcup \begin{bmatrix} \textit{cat:C} \\ \textit{num:pl} \end{bmatrix} = \begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \\ \textit{num:pl} \end{bmatrix}$$

It is also possible to have multiple variables.

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \\ \textit{num:N} \end{bmatrix} \sqcup \begin{bmatrix} \textit{cat:C} \\ \textit{num:pl} \end{bmatrix} = \begin{bmatrix} \textit{cat:verb} \\ \textit{pers:2} \\ \textit{num:pl} \end{bmatrix}$$

Here *verb* is assigned to C, and *pl* is assigned to N.

The following unification fails, because the variable X cannot take on two different values at the same time:

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{num:X} \end{bmatrix} \sqcup \begin{bmatrix} \textit{cat:X} \\ \textit{num:pl} \end{bmatrix} \text{fails!}$$

If the value of a feature is given as a variable in both feature structures, the variable names simply become equivalent. In this example, C becomes equivalent to D.

$$\begin{bmatrix} \textit{cat:C} \\ \textit{pers:2} \end{bmatrix} \sqcup \begin{bmatrix} \textit{cat:D} \\ \textit{num:pl} \end{bmatrix} = \begin{bmatrix} \textit{cat:C} \\ \textit{pers:2} \\ \textit{num:pl} \end{bmatrix} = \begin{bmatrix} \textit{cat:D} \\ \textit{pers:2} \\ \textit{num:pl} \end{bmatrix}$$

[3]By convention, variables begin with an uppercase letter, whereas constants begin with a lowercase letter.

It is also possible to have variables inside feature structures that are themselves values of features. Here is an example where C is set to *noun:*

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{object:} \begin{bmatrix} \textit{cat:noun} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \textit{cat:verb} \\ \textit{object:} \begin{bmatrix} \textit{cat:}C \\ \textit{case:acc} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \textit{cat:verb} \\ \textit{object:} \begin{bmatrix} \textit{cat:noun} \\ \textit{case:acc} \end{bmatrix} \end{bmatrix}$$

Of course, a variable can also stand in for a whole feature structure:

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{object:}O \end{bmatrix} \sqcup \begin{bmatrix} \textit{cat:verb} \\ \textit{object:} \begin{bmatrix} \textit{cat:noun} \\ \textit{case:acc} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \textit{cat:verb} \\ \textit{object:} \begin{bmatrix} \textit{cat:noun} \\ \textit{case:acc} \end{bmatrix} \end{bmatrix}$$

3 The problems

3.1 Constraints on the number of dependencies

There are restrictions on the number of dependencies of certain types that can be established for any one word. One of these restrictions says that a preposition must have an object. For instance, the following sentence is ungrammatical because *per* has no object.

**Canis*	*agitat*	*felem*	*per.*
*Dog	chases	cat	through.

‘*The dog chases the cat through.’

Furthermore, a preposition can have at most one object. Here is a sentence that violates this constraint.

**Canis*	*agitat*	*felem*	*per*	*domum*	*silvam.*
*Dog	chases	cat	through	house	forest.

‘*The dog chases the cat through the house the forest.’

Each of these constraints applies to dependencies of a certain type, and it specifies either a minimal or a maximal number of dependencies of this type that can be established.

The minimal number of dependents can be 0 or 1, and the maximal number of dependents can be 1 or infinity.

In the above examples, the dependency type is OBJECT, the minimal number of dependents is 1, and the maximal number of dependents is also 1.

3.2 Verb subcategorization

Restrictions on the type of dependencies for each category are not sufficient. For example, the following sentence is ungrammatical.

**Canis*	*felem*	*dormit.*
	acc	
*Dog	cat	sleeps.

‘*The dog sleeps the cat.’

The reason is that some verbs can have an object in the accusative case while others cannot. This is expressed by subdividing the verb category into different subcategories. Verbs like *dormire* that cannot have an object might belong to subcategory 1, for example, and verbs that can have an accusative object might belong to subcategory 2. The subcategory of a verb can then be added to its features:

$$\begin{bmatrix} \textit{phon:dormit} \\ \textit{cat:verb} \\ \textit{subcat:1} \\ \textit{num:sing} \\ \textit{pers:3} \end{bmatrix}$$

4 The algorithm

4.1 The algorithm of the original parser

The original parser accepts words from an input list one by one. For each word W, it first tries to insert it as the governor of words read earlier. Then it tries to make W depend on a previous word.

To accomplish this, the parser maintains three lists:

1. The input list,
2. the list of previous words, and
3. the head list.

Words are read from the input list one by one and added, in reverse order, to the list of previous words. The head list contains all the words, together with their direct and indirect dependents, that do not (yet) depend on other words. In the end, the head list becomes the result of the parsing process. It should then contain only the main verb, with all the other words directly or indirectly depending on it.

4.2 Checking the number of dependencies

The check for the maximal number of dependencies is done while parsing. Every time the parser tries to establish a dependency of type T between a word G as the governor and a word D as the dependent, it checks whether the maximal number of dependencies of type T has already been established for G. If so, the parse fails. Otherwise, D is inserted into the list of dependents of type T for G.

The minimal number of dependencies is checked after the parsing proper. For every word W in the input string, the parser checks if for each possible dependency type T, the minimal number of dependents of type T is present for W. If so, this test is done recursively for each dependent. Otherwise, the parse fails.

4.3 Checking the verb subcategory

The subcategory information is included in the lexical entry of each verb. It is used by D-rules like the following one.

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{subcat:2} \end{bmatrix} \longleftarrow \begin{bmatrix} \textit{cat:noun} \\ \textit{case:acc} \end{bmatrix}$$

When applying such a rule, the parser unifies the feature structure on the left hand side with a feature structure that is derived from a lexical entry. This ensures that the subcategories match.

5 The implementation

5.1 The implementation of the original parser

5.1.1 The features

Words are represented as feature structures. The features serve the following purposes:

phon: This feature contains the written form of the word.

cat, subcat: The category and subcategory of the word.

case, num, gen, pers: These are agreement features. For example, a noun and an adjective that modifies it must agree in case, number, and gender.

dep: The value of the *dep* feature of each word is a list of the dependents of that word. After a whole sentence has been parsed successfully, every word hangs directly or indirectly from the main verb (see Figure 1).

$$\begin{bmatrix} \textit{phon:agitat} \\ \textit{cat:verb} \\ \textit{subcat:2} \\ \vdots \\ \textit{dep:}[(\textit{accobject}, [\begin{bmatrix} \textit{phon:felem} \\ \textit{cat:noun} \\ \textit{case:acc} \\ \vdots \end{bmatrix} |\_], 1), (\textit{subject}, [\begin{bmatrix} \textit{phon:canis} \\ \textit{cat:noun} \\ \textit{case:nom} \\ \vdots \end{bmatrix} |\_], 0)|\_] \end{bmatrix}$$

Figure 1: Feature structure for the sentence *Canis felem agitat.*

5.1.2 D-rules

D-rules are implemented as clauses with the principal functor '`<-`'. For example, the D-rule shown above as

$$\begin{bmatrix} \textit{cat:verb} \\ \textit{subcat:2} \end{bmatrix} \longleftarrow \begin{bmatrix} \textit{cat:noun} \\ \textit{case:acc} \end{bmatrix}$$

would look like this in the original parser:

```
cat:verb :: subcat:2 <-    /* governor */
  cat:noun :: case:acc.    /* dependent */
```

My enhancement of the parser uses a different representation (see section 5.2).

5.2 The implementation of my enhancements

The features that are crucial for my enhancement of the parser are *subcat* and *dep.* The *subcat* feature indicates the subcategory of a verb. The value of *dep* is an open list (i.e., one with an uninstantiated tail) of ordered triples. The components of each of these triples are:

1. a dependency type,
2. an open list of dependents (feature structures), and
3. the minimal number of dependencies (of the type mentioned in 1.) that must be established.

That is, the dependents of a word are sorted by dependency type. A verb with a subject and an accusative object might look like in Figure 1.

In a lexical entry, the *dep* feature is only partly instantiated for words with required dependents. For instance, this is a lexical entry for the verb *agitare.* It contains e.g. no triple for the subject, because there need not be a subject in a Latin sentence.[4]

```
word(phon:agitat :: cat:verb :: subcat:2 :: num:sing
  :: pers:3 :: dep:[(accobject,_,1)|_]).
```

For words that do not require dependents, the *dep* feature is not instantiated at all in the lexical entry:

```
word(phon:dormit :: cat:verb :: subcat:1 :: num:sing
  :: pers:3).
```

[4]The number and person of the verb indicate the personal pronoun that serves as logical subject, if no other subject is present.

Further instantiation takes place during the parsing process, when dependents are added.

While the lexicon contains the minimal number of dependencies, the maximal number of dependencies is stored in the D-rules. This makes the parser more efficient, because the maximal number is checked during parsing, i.e. when the D-rules are being accessed anyway. The minimal number, by contrast, cannot be checked until after the parsing proper, when it would be inefficient to go through all D-rules again.

A D-rule also contains the dependency type that it establishes, as can be seen in the following example.

```
drule(cat:verb :: subcat:2,  /* governor */
  cat:noun :: case:acc,      /* dependent */
  accobject,                 /* dependency type */
  1).                        /* max. number of dependencies */
```

When displaying the parsed structure of a sentence, the parser prints out the dependency type for each dependent of a word. The output for the sentence *Canis felem agitat* looks like this:

```
agitat
  felem accobject
  canis subject
```

A The code

The code was written in the Quintus Prolog version of GULP. A complete listing follows.

```
/* File LATIN.GLP */
/* This is the dependency parser.
   There is also a Russian version, which contains mistakes
   (this Latin one is believed to be mistake-free).
   This is not Prolog, this is GULP. */

/*
 * Unification-based dependency grammar
 *
 * This sample parser parses Latin sentences.
 * It considers word order to be totally free.
 */

/*
 * Vocabulary
 */

g_features([phon,cat,subcat,case,num,gen,pers,dep]).

/* The value of dep (if any) is an open list of
   ordered triples.
   The components of each triple are: the dependency type,
   an open list of words (FS's), and the minimal number of
   dependencies (of this type) that must be established.
   The minimal number of dependencies can be 0 or 1. */

word(phon:felis :: cat:noun :: case:nom :: num:sing
 :: gen:masc :: pers:3).
word(phon:felem :: cat:noun :: case:acc :: num:sing
 :: gen:masc :: pers:3).
word(phon:feles :: cat:noun :: case:nom :: num:pl
 :: gen:masc :: pers:3).
word(phon:feles :: cat:noun :: case:acc :: num:pl
 :: gen:masc :: pers:3).
word(phon:canis :: cat:noun :: case:nom :: num:sing
 :: gen:masc :: pers:3).
word(phon:canem :: cat:noun :: case:acc :: num:sing
 :: gen:masc :: pers:3).
word(phon:canes :: cat:noun :: case:nom :: num:pl
 :: gen:masc :: pers:3).
```

```
word(phon:canes :: cat:noun :: case:acc :: num:pl
 :: gen:masc :: pers:3).

word(phon:silva  :: cat:noun :: case:nom :: num:sing
 :: gen:fem :: pers:3).
word(phon:silvam :: cat:noun :: case:acc :: num:sing
 :: gen:fem :: pers:3).
word(phon:silvae :: cat:noun :: case:nom :: num:pl
 :: gen:fem :: pers:3).
word(phon:silvas :: cat:noun :: case:acc :: num:pl
 :: gen:fem :: pers:3).

word(phon:agitat  :: cat:verb :: subcat:2 :: num:sing
 :: pers:3 :: dep:[(accobject,_,1)|_]).
word(phon:agitant :: cat:verb :: subcat:2 :: num:pl
 :: pers:3 :: dep:[(accobject,_,1)|_]).
word(phon:dormit   :: cat:verb :: subcat:1 :: num:sing
 :: pers:3).
word(phon:dormiunt :: cat:verb :: subcat:1 :: num:pl
 :: pers:3).
word(phon:albus :: cat:adj :: case:nom :: num:sing
 :: gen:masc).
word(phon:album :: cat:adj :: case:acc :: num:sing
 :: gen:masc).
word(phon:albi  :: cat:adj :: case:nom :: num:pl
 :: gen:masc).
word(phon:albos :: cat:adj :: case:acc :: num:pl
 :: gen:masc).

word(phon:ater  :: cat:adj :: case:nom :: num:sing
 :: gen:masc).
word(phon:atrum :: cat:adj :: case:acc :: num:sing
 :: gen:masc).
word(phon:atri  :: cat:adj :: case:nom :: num:pl
 :: gen:masc).
word(phon:atros :: cat:adj :: case:acc :: num:pl
 :: gen:masc).
word(phon:atra  :: cat:adj :: case:nom :: num:sing
 :: gen:fem).
word(phon:atram :: cat:adj :: case:acc :: num:sing
 :: gen:fem).
word(phon:atrae :: cat:adj :: case:nom :: num:pl
 :: gen:fem).
word(phon:atras :: cat:adj :: case:acc :: num:pl
 :: gen:fem).
word(phon:parvus :: cat:adj :: case:nom :: num:sing
 :: gen:masc).
word(phon:parvum :: cat:adj :: case:acc :: num:sing
 :: gen:masc).
word(phon:parvi  :: cat:adj :: case:nom :: num:pl
 :: gen:masc).
word(phon:parvos :: cat:adj :: case:acc :: num:pl
 :: gen:masc).

word(phon:magnus :: cat:adj :: case:nom :: num:sing
 :: gen:masc).
word(phon:magnum :: cat:adj :: case:acc :: num:sing
 :: gen:masc).
word(phon:magni  :: cat:adj :: case:nom :: num:pl
 :: gen:masc).
word(phon:magnos :: cat:adj :: case:acc :: num:pl
 :: gen:masc).

word(phon:per :: cat:prep :: case:acc :: dep:[(object,_,1)|_]).
word(phon:in  :: cat:prep :: case:acc :: dep:[(object,_,1)|_]).
/*
 * D-rules (initial set, ignoring word order)
 */

/*
 * drule(Head,Dependent,DepType,Max)
 *   Head is also known as 'governor'.
 *   Max is the maximal number of dependencies (of type
 *   DepType) that can be established.
 *   Max can be 1 or infinity.
 */

drule(cat:verb :: pers:P :: num:N,
  cat:noun :: case:nom :: pers:P :: num:N,
  subject, 1).
drule(cat:verb :: subcat:2,
  cat:noun :: case:acc,
  accobject, 1).
drule(cat:verb, /* no constraint on subcat of verb:
                   prepobject is an adjunct */
  cat:prep,
  prepobject, infinity).
drule(cat:noun :: case:C :: num:N :: gen:G,
  cat:adj :: case:C :: num:N :: gen:G,
  modifier, infinity).
drule(cat:prep :: case:C,
  cat:noun :: case:C,
  object, 1).
/*
 * Dependency parsing
 */

parse(InputList,Result) :-
  parse_aux(InputList,[],[],Result),
  check_req_deps_of_list(Result).

/*
 * parse_aux(InputList,PrevWords,HeadList,Result)
 *
 *   where InputList is the list of words (or rather FS)
 *   yet to be examined, PrevWords contains (in reverse order)
 *   the words already examined, HeadList contains one or more
 *   structures built so far, and Result will be the output of
 *   the parsing process.
 */

parse_aux([],_,Result,Result).

parse_aux([Head|Tail],PrevWords,HeadList,Result) :-
     parse_item(Head,PrevWords,HeadList,NewHeadList),
     parse_aux(Tail,[Head|PrevWords],NewHeadList,Result).

     /* Work along InputList calling parse_item for
        each element. */
/*
 * check_req_deps(Word)
 *   checks if all required dependents of Word are present
 */

check_req_deps(Word) :-
  Word = dep:Deps,
  check_num_deps(Deps).

/*
 * check_req_deps_of_list(Words)
 *   for each element of Words, which is a (possibly open)
 *   list, checks if all its required dependents are present
 */
```

```
check_req_deps_of_list(Words) :-
  nonvar(Words),
  Words = [Word|Tail],
  check_req_deps(Word),
  check_req_deps_of_list(Tail).

check_req_deps_of_list(Words) :-
  nonvar(Words),
  Words = [].  /* at end of non-open list */

check_req_deps_of_list(Words) :-
  var(Words).  /* at end of open list */
/*
 * check_num_deps(DepList)
 *   for each element of DepList, which is an open list of
 *   triples, checks if the minimal number (0 or 1) of
 *   dependents of the type in question is present
 */

check_num_deps(DepList) :-
  nonvar(DepList),
  DepList = [(DepType,Words,1)|Tail],
  nonvar(Words),    /* there is at least one dependent of
                       this type */
  check_req_deps_of_list(Words),
  check_num_deps(Tail).

check_num_deps(DepList) :-
  nonvar(DepList),
  DepList = [(DepType,Words,0)|Tail],
  check_req_deps_of_list(Words),
  check_num_deps(Tail).

check_num_deps(DepList) :- var(DepList).  /* at end of list */
/*
 * parse_item(Item,PrevWords,HeadList,NewHeadList)
 *
 *   inserts Item into HeadList in the proper place,
 *   giving NewHeadList.
 */

parse_item(Item,[],[],[Item]) :- !.
     /* If this is the first word, simply add it to
        HeadList. */

parse_item(Item,PrevWords,HeadList,NewHeadList) :-
     /* First insert Item above as many items as it
        can dominate...*/
     try_inserting_as_head(Item,PrevWords,HeadList,HeadList2),
     /* Then add Item to HeadList...*/
     HeadList3 = [Item|HeadList2],
     /* Then insert Item as dependent of another word
        if possible. */
     try_inserting_as_dependent(Item,PrevWords,HeadList3,
       NewHeadList).
/*
 * try_inserting_as_head(Item,PrevWords,HeadList,NewHeadList)
 *
 *   Search through Headlist and insert Item above one or more
 *   pre-existing items, if possible.
 */

try_inserting_as_head(Item,_,[],[]).
   /* We have recursed all the way down Headlist;
      we may or may not have attached Item above things
      encountered along the way. */
```

```
try_inserting_as_head(Item,_,[Head|HeadList],NewHeadList) :-
   /* Insert Item above Head, and remove Head from HeadList. */
      drule(Item,Head,DepType,Max),
      Item = dep:Deps,
      insert_into_deps(Head,Deps,DepType,Max),
      try_inserting_as_head(Item,_,HeadList,NewHeadList).
          /* Recurse down the rest of HeadList. */

try_inserting_as_head(Item,_,[Head|HeadList],
  [Head|NewHeadList]) :-
   /* Can't do it or choose not to do it, so continue without
      doing it. */
     try_inserting_as_head(Item,_,HeadList,NewHeadList).
/*
 * try_inserting_as_dependent(Item,PrevWords,HeadList,
 *                            NewHeadList)
 *
 *   inserts Item as a dependent of an earlier word
 *   if possible.
 */

try_inserting_as_dependent(Item,PrevWords,HeadList,
                           NewHeadList) :-
     member(PrevWord,PrevWords),
     drule(PrevWord,Item,DepType,Max),
     PrevWord = dep:PrevDeps,
     insert_into_deps(Item,PrevDeps,DepType,Max),
     delete_element(Item,HeadList,NewHeadList).

try_inserting_as_dependent(Item,_,HeadList,HeadList).
/*
 * insert_into_deps(Item,Deps,DepType,Max)
 *   Inserts Item into Deps (in the proper place) as
 *   dependent of type DepType iff less than Max
 *   dependencies of this type have already been established.
 *   (Deps is an open list of triples, Max is either 1 or
 *   infinity.)
 */

insert_into_deps(Item,Deps,DepType,_) :-
  var(Deps),   /* open list is empty */
  Deps = [(DepType,[Item|_],0)|_].
  /* insert with 0 required dependents */

insert_into_deps(Item,Deps,DepType,1) :-
  nonvar(Deps),
  Deps = [(DepType,Words,_)|_],
  var(Words),   /* there are no dependents of this type yet */
  Words = [Item|_].

insert_into_deps(Item,Deps,DepType,infinity) :-
  nonvar(Deps),
  Deps = [(DepType,Words,_)|_],
  insert(Item,Words).

insert_into_deps(Item,Deps,DepType,Max) :-
  nonvar(Deps),
  Deps = [(OtherDepType,_,_)|Tail],
  \+ (OtherDepType = DepType),
  insert_into_deps(Item,Tail,DepType,Max).
/*
 * lex_scan(InputString,ScannedList)
 *   Performs lexical scan, replacing each word with its
 *   lexical entry.
 */

lex_scan(InputString,ScannedList) :-
```

```
     lex_scan_aux(InputString,_,ScannedList).

lex_scan_aux([],_,[]).

lex_scan_aux([Word|Tail],PrevWord,[ScannedWord|ScannedTail]) :-
     ScannedWord = phon:Word,
     word(ScannedWord),
     lex_scan_aux(Tail,ScannedWord,ScannedTail).


/*
 * Manipulation of open lists (with uninstantiated tails)
 */

/*
 * new_tail(List,Tail)
 *  where List has an uninstantiated tail,
 *  instantiates the tail of List to Tail.
 *  NOTE: The empty open list is _, not [_].
 */

new_tail(Y,X) :- nonvar(Y), Y=[_|Tail], new_tail(Tail,X).
new_tail(X,X).

/*
 * insert(Item,List)
 *  adds Item to the end of List, which is (and
 *  remains) an open list.
 */

insert(Item,List) :- new_tail(List,[Item|_]).
/*
 * Utilities
 */


/*
 * once(Goal)
 *   used instead of Arity/Prolog snips
 */

once(X) :- X, !.


/*
 * delete_element(Element,OldList,NewList)
 *   deletes Element from OldList giving NewList.
 */

delete_element(_,[],[]).

delete_element(Element,[Element|Tail],Tail) :- !.

delete_element(X,[Y|Z],[Y|NewZ]) :-
  \+ (Y = X), delete_element(X,Z,NewZ).
/*
 * display_dependency_tree(HeadList)
 *   Displays a dependency tree in indented form,
 *   including dependency type information.
 *   (HeadList may be open.)
 */

display_dependency_tree(HeadList) :-
     display_dependency_indented(HeadList,none,0).

/*
 * display_dependency_indented(HeadList,DepType,Indent)
 *   Auxiliary predicate for display_dependency_tree/1.
 *   DepType is the dependency type of HeadList's elements.
 *   (A dependency type of 'none' is not printed out.)
 *   Indent is the indentation.
 */

display_dependency_indented(HeadList,_,_) :-
     var(HeadList).  /* at end of open list */

display_dependency_indented(HeadList,_,_) :-
     nonvar(HeadList),
     HeadList = [].  /* at end of non-open list */

display_dependency_indented(HeadList,DepType,N) :-
     nonvar(HeadList),
     HeadList = [Head|Tail],
     nonvar(Head),
     tab(N),
     Head = phon:Hphon,
     write(Hphon),
     display_dep_type(DepType),
     nl,
     Head = dep:Hdeps,
     NewN is N+2,
     display_triple_list_indented(Hdeps,NewN),
     display_dependency_indented(Tail,DepType,N).
/*
 * display_triple_list_indented(Deps,N)
 *   Displays the elements of Deps, which is an open list
 *   of triples.
 */

display_triple_list_indented(Deps,_) :-
     var(Deps).

display_triple_list_indented(Deps,N) :-
     nonvar(Deps),
     Deps = [(DepType,Words,_)|Tail],
     display_dependency_indented(Words,DepType,N),
     display_triple_list_indented(Tail,N).

/*
 * display_dep_type(DepType)
 */

display_dep_type(none) :- !.

display_dep_type(DepType) :- write(' '), write(DepType).


/*
 * exhibit(HeadList)
 *   Normal way of displaying a list of feature structures.
 */

exhibit(X) :- var(X), !, write(X), nl.

exhibit([]) :- !,nl.

exhibit([Head|Tail]) :- !,nl,exhibit(Head),exhibit(Tail).

exhibit(X) :- g_display(X).
/* Test */

pause :-  nl,
          write('Press Ctrl-Break to quit, any other key'),
          write(' to look for alternatives...'),
          nl,
          get0(_),
          nl.

try(String) :-      /* String is a list of words as atoms */
        lex_scan(String,X),
        write(String),nl,
        once((write('Scanned string: '),nl,exhibit(X),nl,nl)),
        parse(X,Parsed),
        Parsed = [_],   /* we only want unitary parses */
        once((write('Parsed structure: '),nl,
              display_dependency_tree(Parsed))),
        pause,
        fail.

test1 :- try([agitat,canis,per,silvam,felem]).
test2 :- try([canes,agitant,feles]).
test3 :- try([agitat,agitat]).  /* not a sentence */
test4 :- try([agitat,canis,felem]).
test5 :- try([agitat,felem,canis]).
test6 :- try([canis,agitat,felem]).
test7 :- try([canis,felem,agitat]).
test8 :- try([felem,canis,agitat]).
test9 :- try([felem,agitat,canis]).
test10 :- try([agitat,canis,per,felem]).
  /* is rejected */
test11 :- try([agitat,canis,parvus,ater,per,atram,silvam,
               felem,album]).
test12 :- try([dormit,canis]).
test13 :- try([dormit,canis,per,silvam]).
  /* is accepted: the anomaly is semantic rather than
     syntactic */
test14 :- try([dormit,canis,felem]).  /* is rejected */
```

References

[Covington 1987] Covington, Michael A. (1987) *GULP 1.1: an extension of Prolog for unification-based grammar.* Research report 00-0021, Advanced Computational Methods Center, University of Georgia.

[Covington 1989] Covington, Michael A. (1989) *GULP 2.0: an extension of Prolog for unification-based grammar.* Research report AI-1989-01, Artificial Intelligence Programs, University of Georgia.

[Covington 1990a] Covington, Michael A. (1990) *A dependency parser for variable-word-order languages.* Research report AI-1990-01, Artificial Intelligence Programs, University of Georgia.

[Covington 1990b] Covington, Michael A. (1990) Parsing discontinuous constituents in dependency grammar. *Computational Linguistics* 16.4:234–236.

[Mel'čuk 1988] Mel'čuk, Igor A. (1988) *Dependency syntax: theory and practice.* Albany: State University Press of New York.

[Schubert 1987] Schubert, Klaus (1987) *Metataxis: contrastive dependency syntax for machine translation.* Dordrecht, Holland: Foris.

[Shieber 1986] Shieber, Stuart M. (1986) *An introduction to unification-based approaches to grammar.* (CSLI Lecture Notes, 4.) Stanford: CSLI.

[Tesnière 1959] Tesnière, Lucien (1959) *Éléments de syntaxe structurale.* Paris: Klincksieck.