A Free-Word-Order Dependency Parser in Prolog

Michael A. Covington
Artificial Intelligence Center
The University of Georgia
www.ai.uga.edu/mc

1 Introduction

This paper presents an implementation of the parsing algorithm of Covington (1990a) in plain Prolog, without using GULP (Covington 1994a,b) or other specialized software. It parses a very small subset of Latin and accepts totally free word order. For ease of understanding by students, the implementation is kept as simple as possible, with no attempt to obtain maximum efficiency. The only "advanced" Prolog technique used here is a manipulation of open lists that is fully described in Section 6.

2 The basic algorithm

A dependency tree is a directed acyclic graph in which all the words in a sentence are connected together by grammatical relations. For example, the subject and object depend on the main verb; adjectives depend on the nouns that they modify; and so on. In each pair of connected words, one is called the *dependent* and the other is called the *head.*

This parsing algorithm accepts words one by one, and maintains two lists: `WordList,` which contains all the words seen so far (in reverse order, most recent first), and `HeadList,` which contains all the words that are not (yet) known to be dependents of other words. Whenever a word W is received from the input list, the parser does the following things:

1. Looks W up in the lexicon and creates a node for it (a process that I will describe in more detail shortly).

2. Looks through `HeadList` for words that can be dependents of W; attaches them as such, removing them from `HeadList.` (There may not be any.)

3. Looks through `WordList` for a word of which W can be a dependent. (There will be at most one.) If such a word is found, attaches W as that word's dependent. Otherwise, adds W to `HeadList`.

4. Adds W to `WordList`.

Note that steps 2 and 3 can both occur. That is, the current word can acquire dependents in step 2, and then also acquire a head in step 3.

When the process is complete, and all the words in the input list have been processed, `HeadList` should have only one element, the main verb.

3 The lexicon

In the sample parser, lexical entries are of the form

```
word( Form, [Category, Gloss, Gender, Number, Case, Person] ).
```

where:

`Form` is the word as it actually occurs in the input.

`Category` is the syntactic category (`n`, `v`, or `adj`).

`Gloss` is a label that will identify the word in the output. In the examples, I use an English translation of the word, with extra quotation marks.

`Gender, Number, Case, Person` are grammatical features.

In a more sophisticated implementation, the second argument of `word` might be a GULP feature structure, and the grammar would have a morphological component rather than listing all the inflected forms separately. Here is the lexicon actually used in the example parser:

```
word(canis,   [n,'''dog''',masc,sg,nom,3]).
word(canem,   [n,'''dog''',masc,sg,acc,3]).
word(canes,   [n,'''dog''',masc,pl,nom,3]).
word(canes,   [n,'''dog''',masc,pl,acc,3]).

word(felis,   [n,'''cat''',masc,sg,nom,3]).
word(felem,   [n,'''cat''',masc,sg,acc,3]).
word(feles,   [n,'''cat''',masc,pl,nom,3]).
word(feles,   [n,'''cat''',masc,pl,acc,3]).

word(video,   [v,'''(I) see''', _,sg,_,1]).
word(videmus, [v,'''(we) see''',_,pl,_,1]).
word(videt,   [v,'''sees''',    _,sg,_,3]).
word(vident,  [v,'''see''',     _,pl,_,3]).
word(parvus,  [adj,'''small''',masc,sg,nom,_]).
word(parvum,  [adj,'''small''',masc,sg,acc,_]).
word(parvi,   [adj,'''small''',masc,pl,nom,_]).
word(parvos,  [adj,'''small''',masc,pl,acc,_]).
```

Note that all categories have the same list of grammatical features, even though gender and case are irrelevant for verbs, and person is trivial for nouns (all nouns are third person) and irrelevant for adjectives.

4 The grammar

The grammar consists of rules that specify what dependents can be connected to what heads. These are of the form

```
dh( Dependent, Head ).
```

where `Dependent` and `Head` are feature lists like those in the lexicon. For example:

- Subject and verb: A noun can depend on a verb provided they agree in number and person, and the noun is nominative case. Other features do not matter.

  ```
  dh([n,_,_,Number,nom,Person], [v,_,_,Number,_,Person]).
  ```

- Object and verb: A noun can depend on a verb provided the noun is accusative case. Other features do not matter.

  ```
  dh([n,_,_,_,acc,_], [v,_,_,_,_,_]).
  ```

- Adjective and noun: An adjective can depend on a noun provided they agree in gender, number, and case.

  ```
  dh([adj,_,Gender,Number,Case,_], [n,_,Gender,Number,Case,_]).
  ```

These are the only grammar rules in the sample parser. Others can be added.

5 Representing nodes in the tree

The job of the parser is to construct a dependency parse tree. This is done by maintaining lists of nodes, where each node is like the feature list used in lexical entries, with two more elements added at the beginning. The structure of a node is:

$[\texttt{Count, Dependents, }\underline{\texttt{Category, Gloss, Gender, Number, Case, Person}}]$

The underlined part is familiar.

`Count` is a unique number that indicates the word's position in the input string. It makes the original word order visible in the tree, and it also keeps successive occurrences of the same word from being equated with each other.

`Dependents` is an open list containing nodes that depend on the current one. At the end of the parsing process, `HeadList` will contain only one node, and its list of dependents will contain, directly or indirectly, everything else.

Thus the parse tree is a recursive data structure in which any node can contain more nodes.

6 Motivation and use of open lists

Recall that we are working with two lists of nodes, `HeadList` and `WordList.` The parser will find nodes by looking in either list, and make changes to them (by adding dependents to their dependent-lists).

Crucially, when a node is changed, we want the change to show up in both `HeadList` and `WordList`. We do not want two separate and potentially different copies of each node. We want each node to exist only once, even though it is accessible through both lists.

The most straightforward way to achieve this is to build the structure by instantiating variables. Recall that all instances of a Prolog variable have the same value, even when the variable occurs in more than one place (such as `HeadList` and `WordList`).

Accordingly, the list of dependents in each node will be an open list. The list will grow by gradually instantiating parts of it that were formerly uninstantiated.

An open list is a list whose tail is uninstantiated. Consider the list:

```
[a,b,c|X]
```

Now instantiate:

```
X = [d|Y]
```

Now the original list has become:

```
[a,b,c,d|Y]
```

and if the variable `X` had been accessible in more than one place in the program (e.g., in both `HeadList` and `WordList`), the change would have happened in both places, because all instances of `X` are actually pointers to the same memory location. This is how we will build dependent lists. In particular, to add an item to the end of an open list, we use the predicate:

```
% add_item(?OpenList,+Item)
%   adds Item just before the tail of OpenList

add_item(X,Item) :- var(X), !, X = [Item|_].

add_item([_|X],Item) :- add_item(X,Item).
```

This predicate has no "output" — it works on `OpenList` by instantiating more of it, not by changing it into something else.

Note that we are using open lists, not difference lists. A difference list is a structure consisting of an open list plus a copy of its uninstantiated tail. If we were using difference lists, `add_item` would need separate "input" and "output" arguments (because the uninstantiated tail becomes a different variable), and that's what we're trying to avoid. The disadvantage of this tactic is that, unlike difference lists, open lists do not give us a way to jump directly to the place where an element will be added; we have to work down the list from the beginning each time, like a conventional list.

Finally, note that the empty open list is `_` (an uninstantiated variable), not `[]`.

7 The parser itself

7.1 parse(+InputList,-Result)

To start parsing a list of words, the parser must initialize the node count as 1 and `HeadList` and `WordList` as `[]`, then pass control to the main loop:

```
parse(InputList,Result) :-
   parse_loop(1,InputList,[],[],Result).
```

7.2 parse_loop(+Count,+InputList,+WordList,+HeadList,-Result)

The predicate `parse_loop` steps through the input list, word by word. For each word, it creates a node, then tries to connect the node to other nodes that already exist. When it runs out of words, it returns `HeadList` as the result:

```
parse_loop(Count,[Word|InputList],WordList,HeadList,Result) :-
   word(Word,WordFeatures),                          % Look up the next word
   Node = [Count, _ | WordFeatures],                 % Build a node for it
   parse_node(Node,WordList,HeadList,NewHeadList),   % Try to attach it
   NewCount is Count + 1,                            % Increment the counter
   NewWordList = [Node|WordList],                    % Add Node to WordList
   parse_loop(NewCount,InputList,NewWordList,NewHeadList,Result).

parse_loop(_,[],_,Result,Result).   % Nothing more to do, so Result = HeadList
```

7.3 parse_node(+Node,+WordList,+HeadList,-NewHeadList)

This is the predicate that takes a newly created node and tries to connect it to other nodes. In so doing, it can alter `HeadList` by adding or removing elements, hence the need for an output argument, `NewHeadList`.

```
parse_node(Node,[],[],[Node]) :- !.
   % If this is the first word, just add it to HeadList.
```

```
parse_node(Node,WordList,HeadList,NewHeadList) :-
   try_inserting_as_head(Node,HeadList,HeadList2),
   try_inserting_as_dependent(Node,WordList,HeadList2,NewHeadList).
```

7.4 try_inserting_as_head(+Node,+HeadList,-HeadList2)

This predicate tries to attach `Node` as a head above one of the words presently in `HeadList`. To do this, it must look at all the elements of `HeadList`, because any and all of them could be dependents of `Node`. For instance, a noun might be preceded by several adjectives that modify it.

There are three clauses. First, and most obviously, when you get to the end of `HeadList`, stop:

```
try_inserting_as_head(_,[],[]).
   % No more elements of HeadList to look at.
```

Second, and more commonly, look at the grammatical features of `Node` and of `Head` (an element of `HeadList`). See if `Head` can be a dependent of `Node`. If so, add `Head` to `Node`'s open list of dependents by instantiating part of the tail (using `add_item`). Then recursively look for more dependents, using a version of `HeadList` from which `Head` has been removed.

```
try_inserting_as_head(Node,[Head|HeadList],NewHeadList) :-
   % Insert Node above Head, and remove Head from HeadList.
   Node = [_,D|NodeFeatures],
   Head = [_,_|HeadFeatures],
   dh(HeadFeatures,NodeFeatures),  % Head is the dependent here
   add_item(D,Head),
   % Now recurse down HeadList and see if it can be done again.
   % (Multiple elements of HeadList may be dependents of Node.)
   try_inserting_as_head(Node,HeadList,NewHeadList).
```

That's not the only option. The alternative is that either the dependency connection couldn't be made, or the parser chose not to do it (because it's a nondeterministic parser). In that case we skip `Head` and recurse, preserving `Head` in the version of `HeadList` that will be passed along to the next step in the parser.

```
try_inserting_as_head(Node,[Head|HeadList],[Head|NewHeadList]) :-
   % Couldn't use Head, so skip it.
   try_inserting_as_head(Node,HeadList,NewHeadList).
```

7.5 try_inserting_as_dependent(+Node,+WordList,+HeadList2,-NewHeadList)

A node can have many dependents but only one head. Its dependents have to be found in `HeadList` (which is what we just did), but its head can be anywhere, so we need to search for it in `WordList`.

Accordingly, this predicate requires `WordList` as well as `HeadList2` as arguments. (`HeadList2` is `HeadList` as output by the previous predicate.)

There are only two clauses. Recall that `?- member(X,[a,b,c])` returns, as backtracking alternatives, all the elements of the list, with `X` instantiated to a different one each time. In the same way, `?- member(Word,WordList)` instantiates `Word` to a member of `WordList` and, upon backtracking, a different member, until it runs out of alternatives. Accordingly, here is how we search for the head of `Node`:

```
try_inserting_as_dependent(Node,WordList,HeadList,HeadList) :-
   member(Word,WordList),
   Word = [_,D|WordFeatures],
   Node = [_,_|NodeFeatures],
   dh(NodeFeatures,WordFeatures),   % Check that dependency is permitted
   add_item(D,Node).                % Place Node into dependent list of Word
```

Here `D` is the open list of dependents of `Word`, and `add_item` instantiates it further to include `Node`.

There is of course an alternative. If the grammar does not permit the link to be established, or if the parser chooses not to establish it, the second clause is used instead:

```
try_inserting_as_dependent(Node,_,HeadList,[Node|HeadList]).
```

This just adds `Node` to `HeadList`.

8 Output utilities

An open list of nodes is hard to print out readably. Accordingly, the program also contains predicates `write_list`, `write_dep` (for lists of dependents), and `try` to actually manage the parsing process, then print out the dependency tree as an indented structure. You can query a sentence and get all its parses like this:

```
try([canis,parvum,videt,felem]).
canis parvum videt felem

3 v 'sees' _ sg _ 3
   1 n 'dog' masc sg nom 3
   4 n 'cat' masc sg acc 3
      2 adj 'small' masc sg acc _

No (more) parses.
```

That is: ‘sees’ is the head word, and it has two dependents, ‘dog’ and ‘cat’. In turn, ‘cat’ has another dependent, ‘small’. If a sentence is ambiguous, multiple parse trees are returned.

9 Limitations

This parser is *far* from adequate for actual parsing of Latin.

- It treats word order as totally free. There are no limits on word order at all (for instance, a preposition could come after its object, or be separated from its object by any distance — things that do not actually happen in Latin).
- It treats all dependents as optional. There is no way to specify that a verb must have an object, or that it must have only one object.

These limitations can be overcome partly by enriching the grammar with feature structures, and partly by adding projectivity constraints to the parsing algorithm (Covington 2001).

References

Covington, Michael A. (1990a) Parsing discontinuous constituents in dependency grammar. *Computational Linguistics* 16:234–236.

Covington, Michael A. (1990b) *A Dependency Parser for Variable-Word-Order Languages.* Research Report AI-1990-01, Artificial Intelligence Center, The University of Georgia. Available online at http://www.ai.uga.edu/ftplib/ai-reports/ai199001.ps.

Covington, Michael A. (1994a) *GULP 3.1: An Extension of Prolog for Unification-Based Grammar.* Research Report AI-1994-06, Artificial Intelligence Center, The University of Georgia. Available online at http://www.ai.uga.edu/ftplib/ai-reports/ai199406.ps.

Covington, Michael A. (1994b) *Natural Language Processing for Prolog Programmers.* Englewood Cliffs, N.J.: Prentice-Hall.

Covington, Michael A. (2001) A fundamental algorithm for dependency parsing. *Proceedings of the 39th Annual ACM Southeast Conference,* ed. John A. Miller and Jeffrey W. Smith, pp. 95–102.