Querying Linguistic Corpora with Prolog

Gerlof Bouma
Department Linguistik
Universität Potsdam
Postsdam, Germany
gerlof.bouma@uni-potsdam.de

Abstract

In this paper we demonstrate how Prolog can be used to query linguistically annotated corpora, combining the ease of dedicated declarative query languages and the flexibility of general-purpose languages. On the basis of a Prolog representation of the German Tüba-D/Z Treebank, we show how one can tally arbitrary features of (groups) of nodes, define queries that combine information from different layers of annotation and cross sentence boundaries, query 'virtual annotation' by transforming annotation on-the-fly, and perform data driven error analysis. Almost all code needed for these case studies is contained in the paper.

1 Introduction

In recent years, there has been a strong increase in the availability of richly annotated corpora and corpora of ever growing size. In tact with this, there is a thriving research into ways of exploiting these corpora, where especially the conflicting constraints posed by the desire for an expressive query formalism and the computational demands of querying a large corpus form a driving tension. In this paper we hope to contribute to this debate by advocating the use of Prolog, a general-purpose language, to query corpora. We will argue by means of a series of concrete examples that Prolog is declarative enough to allow formulation of corpus queries in an intuitive way, that it is flexible and powerful enough to not constrain the computational linguist wishing to get as much as possible out of a corpus, and that on modern Prolog implementations, it is fast enough to intensively use corpora of a million tokens or more.

Before we turn to the examples that make up the body of this paper, we briefly discuss what makes Prolog a good language for corpus querying, but also what its disadvantages are. It should be clear from the outset, however, that we do not propose Prolog per se to be used as a query tool for the (non-programmer) general linguist who wants a fully declarative corpus environment, including tree vizualization, etc. Rather, our target is the advanced corpus user/computational linguist who needs an extendable query language and features beyond what any specific dedicated query tool can offer, that is, the type of user who will end up using a general-purpose language for part of their corpus tasks.

1.1 Why use Prolog?

Semi-declarativeness, non-deterministic search Prolog is well-suited to write database-like queries in (see e.g., Nilsson and Maluszynski (1998) for a description of the relation between relational database algebra and Prolog) – one defines relations between entities in terms of logical combinations of properties of these entities. The Prolog execution model is then responsible for the search for entities that satisfy these properties. This is one of the main advantages over other general-purpose languages: in Prolog, the programmer is relieved of the burden of writing functions to search through the corpus or interface with a database.

Queries as annotation Any query that is more complicated than just requesting an entry from the database is a combination of the user's knowledge of the type of information encoded in the database and its relation to the linguistic phenomenon that the user is interested in. Thus, a query can be understood as adding annotation to a corpus. In Prolog, a query takes the form of a predicate, which can then be used in further predicate definitions. In effect, we can query annotation that we have ourselves added.

The lack of a real distinction between existing annotation (the corpus) and derived annotation (subqueries) is made even more clear if we consider the possibility to record facts into the Prolog database for semi-permanent storage, or to write out facts to files that can then later be loaded as given annotation. This also opens up the possibility of performing corpus transformations by means of querying. Related to the queries as annotation perspective is the fact that by using a general-purpose programming language, we are not bound by the predefined relations of a particular query language. New relations can be defined, for instance, relations that combine two annotation layers (see also Witt (2005)), or cross sentence boundaries.

Constraining vs inspecting TIGERSearch (König et al., 2003) offers facilities to give (statistical) summaries of retrieved corpus data. For instance, one can get a frequency list over the POS tags of retrieved elements. This is a very useful feature, as it is often such summaries that are relevant. The reversibility of (well-written) Prolog predicates facilitates implementing such functionality. It is possible to use the exact same relations that one uses to constrain query matches to request information about a node. If `has_pos/2` holds between a lexical node and its POS-tag, we can use it to require that a lexical node in a query has a certain POS-tag or to ask about a given node what its POS-tag is.

Scope of quantifiers and negation Query languages differ in the amount of control over the scope of quantifiers and negation in a query (Lai and Bird, 2004). For instance, Kepser's (2003) first-order-logic based query language allows full control over scoping by explicit quantification. On the other hand, TIGERSearch's query language (König et al., 2003) is restrictive as it implicitly binds nodes in a query with a wide scope existential quantifier. Queries like *find an NP that does not contain a Det node* are not expressible in this language.

In a general-purpose language we get full control over scope. We can illustrate this with negation, canonically implemented in Prolog by negation as (Prolog) failure (written: `\+`). By doing lookup in the database of subtrees/nodes inside or outside of a negated goal, we vary quantifier scope: `lookup(X), \+ p(X)` succeeds when X does not have property p, and `\+ (lookup(X), p(X))` succeeds when there is no X with p. A discussion of the implementation of queries that rely on this precise control over scope can be found in (Bouma, 2010).

1.2 Why not use Prolog?

Not so declarative Compared to dedicated query languages, Prolog lacks declarativeness. The order in which properties are listed in a query may have consequences for the speed with which answers are returned or even the termination of a query. The use of Prolog negation makes this issue even worse. For many queries, there is a simple pattern that avoids the most common problems, though: 1) supply positive information about nodes, then 2) access the database to find suitable candidates, and 3) check negative information and properties that involve arithmetic operations. Most of the examples that we give in the next section follow this pattern.

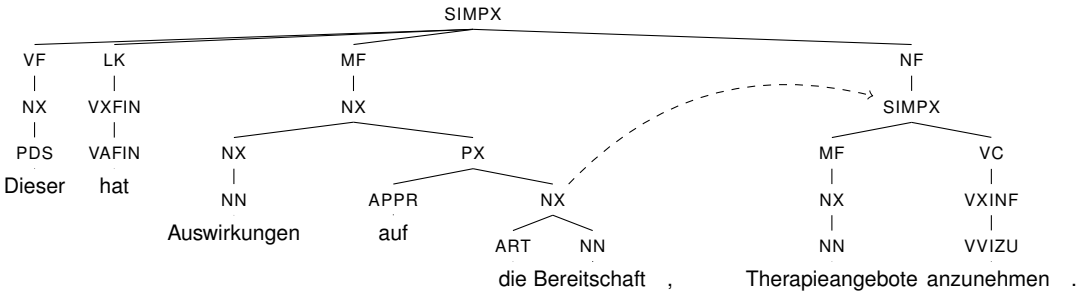
Poor regular expression support Although there are some external resources available, there is no standardized regular expression support in Prolog. This contrasts with both dedicated query languages and with other high-level general-purpose programming languages. However, for some uses of regular expressions, there are good alternatives. For instance, restricting the POS-tag of a node to a known and finite set of POS-tags could also be achieved through a disjunction or by checking whether the POS-tag occurs in a list of allowed POS-tags. These alternatives are typically easy and fast in Prolog.

After these abstract considerations, we shall spend the rest of the paper looking more concretely at Prolog as a query language in a number of cases. After that, in Section 4, we briefly discuss the speed and scalability of a Prolog-based approach.

2 Exploiting the TüBa-D/Z corpus

In this section, we will demonstrate the flexibility of our approach in a series of small case studies on the TüBa-D/Z treebank of German newspaper articles (Telljohann et al., 2006, v5). The treebank has a size of approximately 800k tokens in 45k sentences, and contains annotation for syntax (topological fields, grammatical functions, phrases, secondary relations) and anaphora. We chose this treebank to be able to show a combination of different annotation layers in our queries.

The section is rather code heavy for a conference paper. By including the lion's share of the Prolog code needed to do the tasks in this section, we intend to demonstrate how concise and quick to set up corpus programming in Prolog can be.



"This has effects on the willingness to accept therapy."

```
node(153, 0, 500, 'Dieser', hd, pds, [morph=nsm]).
node(153, 1, 501, hat, hd, vafin, [morph='3sis']).
node(153, 2, 502, 'Auswirkungen', hd, nn, [morph=apf]).
node(153, 3, 508, auf, -, appr, [morph=a]).
node(153, 4, 503, die, -, art, [morph=asf]).
node(153, 5, 503, 'Bereitschaft', hd, nn, [morph=asf]).
node(153, 6, 0, (','), --, '$,', [morph= --]).
node(153, 7, 504, 'Therapieangebote', hd, nn, [morph=apn]).
node(153, 8, 505, anzunehmen, hd, vvizu, [morph= --]).
node(153, 9, 0, '.', --, $., [morph= --]).



secondary(153,503,512,refint).
```

```
node(153, 515, 0, '$phrase', --, simpx, []).
node(153, 506, 515, '$phrase', -, vf, []).
node(153, 500, 506, '$phrase', on, nx, []).
node(153, 507, 515, '$phrase', -, lk, []).
node(153, 501, 507, '$phrase', hd, vxfin, []).
node(153, 513, 515, '$phrase', -, mf, []).
node(153, 511, 513, '$phrase', oa, nx, []).
node(153, 502, 511, '$phrase', hd, nx, []).
node(153, 508, 511, '$phrase', -, px, []).
node(153, 503, 508, '$phrase', hd, nx, []).
node(153, 514, 515, '$phrase', -, nf, []).
node(153, 512, 514, '$phrase', mod, simpx, []).
node(153, 509, 512, '$phrase', -, mf, []).
node(153, 504, 509, '$phrase', oa, nx, []).
node(153, 510, 512, '$phrase', -, vc, []).
node(153, 505, 510, '$phrase', hd, vxinf, []).
```

Figure 1: A tree from Tüba-D/Z and its Prolog representation.

2.1 Corpus representation

We take the primary syntactic trees as the basis of the annotation. Following Brants (1997), we store the corpus as collection of directed acyclic graphs, with edges directed towards the roots of the syntactic trees. A tree from the TüBa-D/Z corpus is represented as a collection of facts `node/7`, which contain for each node: an identifier (a sentence id and a node id), the id of the mother node, the edge label, its surface form, its category or POS-tag and a list with possible other information. Below we see two facts for illustration – a lexical (terminal) node and a phrasal node that dominates it.[1] Phrases carry a dummy surface form `'$phrase'`. Secondary edges are represented as `secondary/4` facts, and use the sentence and node ids of the primary trees.

```
% node/7 SentId NodeId MotherId
%                          Form Edge Cat Other
node(153, 4, 503, die, -, art, [morph=asf]).
node(153, 503, 508, '$phrase', hd, nx, []).
```

```
% secondary/4 SentId NodeId MotherId Edge
secondary(153,503,512,refint).
```

By using the sentence number as the first argument of `node/7` facts, we leverage first argument indexing to gain fast access to any node in the treebank. If we know a node's sentence number, we never need to search longer than the largest tree in the corpus. Since syntactic relations hold within a sentence, querying syntactic structure is generally fast (Section 4). A tree and its full representation is given in Figure 1. We will not use the secondary edges in this paper: their use does not differ much from querying primary trees and the anaphora annotation (Section 2.4). We can define interface relations on these facts that restrict variables without looking up any nodes in the database by partially instantiating them.

```
has_sentid(node(A_s,_,_,_,_,_,_),A_s).
has_nodeid(node(_,A_n,_,_,_,_,_),A_n).
has_mother(node(_,_,A_m,_,_,_,_),A_m).
  has_form(node(_,_,_,A_f,_,_,_),A_f).
  has_edge(node(_,_,_,_,_,A_e,_),A_e).
has_poscat(node(_,_,_,_,_,A_p,_),A_p).

is_under(A,B):-
    has_mother(A,A_m,A_s),
    is_phrasal(B),
    has_nodeid(B,A_m,A_s).
```

[1] Coding conventions: Most predicate names are VS(O) sentences: `has_edge(A,A_e)` reads *node A has edge label* A_e. Predicates defined for their side-effects get imperative verbs forms. Variables `A`,`B`,`C` refer to nodes, and subscripts are used for properties of these nodes. Variables that represent updates are numbered `A`, `A1`, `A2`. Lists receive a plural-*s*. For readability, we use the if-then-else notation instead of cuts as much as possible.

```
is_under_as(A,B,A_e):-
    is_under(A,B),
    has_edge(A,A_e).

are_sentmates(A,B):-
    has_sentid(A,A_s),
    has_sentid(B,A_s).

is_phrasal(A):-
    has_form(A,'$phrase').
```

Actually looking up a node in the corpus involves calling a term `node/7`, for instance by defining a property of a node-representing variable and then calling the variable: `is_phrasal(A), A` will succeed once for each phrasal node in the corpus.

Transitive closures over the simple relations above define familiar predicates such as dominance (closure of `is_above/2`). In contrast with the simple relations, these closures do look up (instantiate) their arguments. In addition, `has_ancestor/3` also returns a list of intermediate nodes.

```
has_ancestor(A,B):-
    has_ancestor(A,B,_).

has_ancestor(A,B,AB_path):-
    are_sentmates(A,B),
    A,  is_under(A,A1),  A1,
    has_ancestor_rfl(A1,B,AB_path).
    has_ancestor_rfl(A,A,[]).

 has_ancestor_rfl(A,B,[A|AB_path]):-
     is_under(A,A1),  A1,
     has_ancestor_rfl(A1,B,AB_path).
```

With these basic relations in place, let us look at determining linear order of phrases.

2.2 Linear order

As yet, linear order is a property of words in the string (lexical nodes), that we may determine by looking at their node ids (cf. Figure 1). Linear order of phrases is not defined. We can define the position of *any* node as its span over the string, which is determined by the outermost members in the node's yield, that is, the members of the yield with the minimum and maximum node id.[2]

[2] `fold/3` (left fold, aka *reduce*) and `map/N` are higher order predicates that generalize predicates to apply to list(s) of arguments, familiar from functional programming (see e.g., Naish (1996)). Given that `min/3` relates two numbers and their minimum, the goal `fold(min,Ns,M)` succeeds if `M` is the lowest number in of the list `Ns`. Given that `has_nodeid/2` relates one node and its id, the goal `map(has_nodeid,As,A_ns)` succeeds on a list of nodes `As` and a list of their corresponding ids `A_ns`.

```
spans(A,A_beg,A_end):-
    yields_dl(A,Bs\[]),
    map(has_nodeid,Bs,B_ns),
    fold(min,B_ns,A_beg),
    fold(max,B_ns,B_n_mx),
    A_end is B_n_mx+1
```

The yield of a phrase is the combined yields of its daughters. A lexical node is its own yield.

```
yields_dl(A,Bs):-
    is_phrasal(A)
      -> ( is_above(A,A1),
           findall(A1, A1, A1s),
           map(yields_dl,A1s,Bss),
           fold(append_dl,Bss,Bs)
         )
    ; % is_lexical(A)
      Bs = [A|Cs]\Cs.
```

According to this definition, the span of the word Auswirkungen in the tree in Figure 1 is 2–3, and the span of the MF-phrase is 2–6.

It makes sense to store the results from `spans/2` instead of recalculating them each time, especially if we intend to use this information often. Using a node's span, we can define other relations, such as precedence between phrasal and lexical nodes, and edge alignment.

```
precedes(A,B):-
    are_sentmates(A,B),
    spans(A,_,A_end),
    spans(B,B_beg,_),
    A_end =< B_beg.

are_right_aligned(A,B):-
    are_sentmates(A,B),
    spans(A,_,A_end),
    spans(B,_,A_end).
```

Although there are no discontinuous primary phrases in Tüba-D/Z, the definition of precedence above would be appropriate for such phrases, too. Note, however, that in this case two nodes may be unordered even when one is not a descendant of the other. In TIGERSearch, precedence between phrases is defined on both left corners (König et al., 2003). It would be trivial to implement this alternative.

2.3 Phrase restricted bigrammes

In the first task, we see a combination of negation scoping over an existential quantifier in a query and the use of a predicate to ask for a property (surface form) rather than to constrain it. The task is to retrieve token bigrammes contained in non-recursive NPs, which are NPs that do not contain other NPs.

This requires a negation scoping over the selection of a descendent NP node. Once we have a non-recursive NP, we select adjacent pairs of nodes from its ordered yield and return the surface forms:

```
bigr_in_nonrec_NP(A_f,B_f):-
    has_form(A,A_f),
    has_form(B,B_f),
    has_cat(C,nx),
    has_cat(D,nx),
    C,  \+ has_ancestor(D,C),
    yields_ord(C,Es),
    nextto(A,B,Es).
```

`yields_ord/2` holds between a node and its ordered yield. Ordering is done by decorate-sort-undecorate.

```
yields_ord(A,Bs):-
    yields_dl(A,Cs\[]),
    keys_values(CnsCs,C_ns,Cs),  % decorate
    map(has_nodeid,Cs,C_ns),
    keysort(CnsCs,BnsBs),        % sort
    values(BnsBs,Bs).            % undecorate
```

The query succeeds 160968 times, that is, there are 161k bigramme tokens in non-recursive NPs in the corpus. There are 105685 bigramme types, of which the top 10 types and frequencies are:

(1)	1	der Stadt	141	6	der Welt	103
	2	der Nato	138	7	den letzten	103
	3	mehr als	136	8	die Polizei	98
	4	die Nato	125	9	ein paar	97
	5	die beiden	107	10	den USA	90

2.4 Combining annotation layers

As mentioned, the Tüba-D/Z corpus additionally contains annotation of anaphora and coreference. This annotation layer can be considered as a graph, too, and may be stored in a fashion similar to the secondary edges. The sentence and node ids in the `anaphor/5` facts are again based on the primary trees.

```
% anaphor/4 SentId NodeId Rel SentIdM NodeIdM
anaphor(4, 527, coreferential, 1, 504).
anaphor(4, 6, anaphoric, 4, 527).
anaphor(6, 522, coreferential, 4, 512).
```

In addition, we have a convenience predicate that links `node/7` terms to the anaphora facts.

```
is_linked_with(A,Rel,B):-
    has_nodeid(A,A_n,A_s),
    has_nodeid(B,B_n,B_s),
    anaphor(A_s,A_n,Rel,B_s,B_n).
```

A very basic combination of the two annotation layers allows us to formulate the the classic i-within-i constraint (Hoeksema and Napoli, 1990).

	Antecedent GF				
Pron. GF	on	oa	od	rest	Total
on	**2411** **.03**	236 -.09	162 -.06	589	3398
oa	168 -.18	**46** **.73**	16 .08	62	292
od	173 -.03	20 .02	**19** **.38**	45	257
rest	142	17	15	63	237
Total	2894	319	212	759	4184

Table 1: Cross tabulation of grammatical function of pronoun-antecedent pairs from adjacent sentences, counts and association scores (PMI).

```
% [ ... X_i ... ]_i
i_within_i(A,B):-
    is_linked(A,_Rel,B),
    has_ancestor(A,B).
```

The query returns 19 hits, amongst which (2):

(2) [die kleine Stadt mit ihren$_i$ 7.000 Einwohnern]$_i$
the small town with its inhabitants

2.5 Grammatical function parallelism

In anaphora annotation, links can be made between nodes that are not contained within one sentence – a coreference link could span the entire corpus. In this task, we follow intra-sentential links. We will try to find corpus support for the (not uncontested) claim that people prefer to interpret pronouns such that the antecedent and pronoun have the same grammatical function (Smyth, 1994, a.o.). As a reflection of this preference, we might expect that there is a trend for a pronoun and its antecedent in the directly preceding sentence to have the same grammatical function. The predicate `ana_ant_gf/2`, defined below, returns the grammatical functions of an anaphoric NP headed by a personal pronoun and its NP antecedent if it occurs in the immediately preceding sentence.

```
ana_ant_gf(A_e,B_e):-
    has_edge(A,A_e,A_s),
    is_under_as(A1,A,hd),
    has_pos(A1,pper),
    has_edge(B,B_e,B_s),
    has_cat(B,nx),
    is_linked_with(A,anaphoric,B),
    B_s is A_s-1,    % B in sentence before A
    A,  A1,  B.
```

The query succeeds just over 4k times. Table 1 summarizes the results. For the top-left cells, we've calculated pointwise association (PMI) between the two

variables. The rows on the diagonal have positive values, which means the combination occurs more often than expected by chance, as expected by the parallelism hypothesis.[3] In Section 2.7, we will revisit this task.

2.6 Coreference chains

Until now, we have used the anaphoric annotation as-is. However, we can also consider it in terms of coreference chains, rather than single links between nodes. That is, we can construct equivalence classes of nodes that are (transitively) linked to each other: they share one discourse referent. Naïve construction of such classes is hampered by the occurrence of cycles in the anaphora graph. Therefore, we need to check for each anaphoric node whether its containing graph contains a cycle. If it does, we pick any node in the cycle as the root of the graph. Non-cyclic graphs have the (unique) node with out-degree zero as their root. We use the Prolog database to record the roots of cyclic graphs, so that we can pick them as the root next time we come to this graph.

```
has_root(A,B):-
    is_linked_with(A,A1),
    ( leads_to_cycle_at(A1,C)
      -> ( B = C,
           record_root(B)
         )
    ; has_root_rfl_nocycles(A,B)
    ).

has_root_rfl_nocycles(A,B):-
    is_recorded_root(A)
      -> B = A
    ; is_linked_with(A,A1)
      -> has_root_rfl_nocycles(A1,B)
    ; B = A.
```

The cycle check itself is based on Floyd's tortoise and hare algorithm, whose principle is that if a slow tortoise and a fast hare traversing a graph land on the same node at the same time, there has to be a cycle in the graph. In our version, the tortoise does not traverse already recorded root nodes, to prevent the same cycle from being detected twice.

```
leads_to_cycle_at(A,B):-
    is_linked_with(A,A1),
    is_linked_with(A1,A2),
    tortoise_hare(A1,A2,B).
```

[3]This should not be taken as serious support for the hypothesis, though. The association values are small and there are other positive associations in the table. Also, we have not put a lot of thought into how the annotation relates to the linguistic phenomenon that we are trying to investigate.

	Antecedent GF				
Pron. GF	on	oa	od	rest	Total
on	**4563** **.02**	500 -.07	353 -.05	1312	6728
oa	367 -.13	**85** **.53**	43 .21	134	629
od	392 -.02	48 .00	**47** **.34**	115	602
rest	309	39	26	136	510
Total	5631	672	469	1697	8469

Table 2: Cross tabulation of grammatical function of pronoun-antecedent pairs from adjacent sentences, counts and association scores (PMI). Revisited.

```
tortoise_hare(A,B,C):-
    \+ is_recorded_root(A),
    ( A = B  % evidence of cycle
      -> C = A
    ; ( is_linked_with(A,A1),  % tort. to A1
        is_linked_with(B,B1),  % hare to
        is_linked_with(B1,B2), %         B2
        tortoise_hare(A1,B2,C)
      )
    ).
```

Collecting all solutions for `has_root/2`, we find 20516 root and 50820 non-root referential expressions. The average coreference chain length is 3.48, the longest chain has 176 mentions in 164 sentences.

2.7 Revisiting parallelism

Let us go back to pronoun-antecedent parallelism with this alternative view of the anaphora annotation. In our first attempt, we missed cases where a pronoun's referent is mentioned in the previous sentence, just not in a node directly anaphorically linked to the pronoun. The coreference chain view gives us a chance to get at these cases. Note that now a pronoun may have more than one antecedent in the preceding sentence. The predicate `ana_ant_gf_rev/2` succeeds once for each of the possible pairs:

```
ana_ant_gf_rev(A_e,B_e):-
    has_edge(A,A_e,A_s),
    is_under_as(A1,A,hd),
    has_pos(A1,pper),
    has_edge(B,B_e,B_s),
    has_cat(B,nx),
    A1,  A,
    B_s is A_s-1,
    corefer(A,B).
```

Coreference between two given and distinct nodes can be defined on `has_root/2`. That is, the definition

of coreference relies on a transformation of the original annotation, that we are producing on-the-fly.

```
corefer(A,B):-
    has_root(A,B).
corefer(A,B):-
    has_root(B,A).
corefer(A,B):-
    has_root(A,C),
    has_root(B,C).
```

Just like in our first attempt, we collect the results in a table and calculate observed vs expected counts. As could be expected, we get many more datapoints (∼8.5k). Table 2 shows a similar picture as before: small, positive associations in the diagonal cells.

3 Corpus inspection

With the techniques introduced thus far, we can perform corpus inspection by formulating and calling queries that violate corpus well-formedness constraints. A data-driven, large scale approach to error mining is proposed by Dickinson and Meurers (2003). Errors are located by comparing the analyses assigned to multiple occurrences of the same string. A version of this idea can be implemented in the space of this paper. Naïve comparison of all pairs of nodes would take time quadratic in the size of the corpus. Instead, we record the string yield and category of each interesting phrasal node in the database, and then retrieve conflicts by looking for strings that have more than one analysis. First, an interesting phrase is one that contains two or more words, unary branching supertrees.

```
interesting_node(A,Bs):-
    phrasal(A), A,
    \+ is_alone_under(_,A),
    Bs = [_,_|_],   % >= two nodes in yield
    yields_ord(A,Bs).

is_alone_under(A,B):-
    is_under(A,B), A, B,
    \+ ( is_above(B,A1),  A1,  A1\=A ).
```

Then, recording a string involves checking whether we have already seen it before and, if so, whether we have a new analysis or an existing one.

```
record_string_analysis(A,Bs):-
    map(has_form,Bs,B_fs),
    fold(spaced_atom_concat,B_fs,String),
    ( retract(str_analyses(String,Analyses))
      -> insert_analysis(A,Analyses,Analyses1)
    ; insert_analysis(A,[],Analyses1)
    ),
    assert(str_analyses(String,Analyses1)).

insert_analysis(A,Analyses,Analyses1):-
    has_cat(A,A_c),
    ( select(A_c-As,Analyses,Analyses2)
      -> Analyses1 = [A_c-[A|As]|Analyses2]
    ; Analyses1 = [A_c-[A]|Analyses]
    ).
```

Exhaustively running the two main queries asserts 364785 strings into the database, with averages of 1.0005 different categories per string and 1.1869 occurrences per string.

The query `str_analyses(Str,[_,_|_])` succeeds 178 times, once for each string with more than one analysis in the corpus. Far from all of these are true positives. Common false positives are forms that can be AdvPs (ADVX), NPs (NX) or DPs, such as *immer weniger* 'less and less':

(3) das Flügelspiel fand [$_{\text{ADVX}}$ immer weniger]
the piano playing found less and less
statt
place
'The piano was played less and less often.'

(4) Japan importiert [$_{\text{NX}}$ immer weniger]
Japan imports less and less
'Japan imports fewer and fewer goods.'

(5) Die braven BürgerInnen produzieren [$_{\text{DP}}$
Those good citizens produce
immer weniger] Müll
less and less waste
'The good citizens produce less and less waste'

We also see borderline cases of particles that might or might not be attached to their neighbours:

(6) Für Huhn ungewöhnlich saftig [$_{\text{MF}}$ auch sie]
for chicken remarkably juicy also it
'It, too, was remarkably juicy, for being chicken.'

(7) Wahrscheinlich streift [$_{\text{NX}}$ auch sie] in diesem
Probably roams also she at this
Moment durch ihr Nachkriegsberlin.
moment through her post-war Berlin
'Probably, she, too, roams through her post-war Berlin at this moment.'

In (6), the node of interest is labelled MF for the topological Mittelfeld. This is not a traditional constituent, but since Tüba-D/Z annotates topological fields we also capture some cases where a string is a constituent in one place and a non-constituent in another. Dickinson and Meurers (2003) introduce dummy constituents to systematically detect a much wider range of those cases.

Finally, real errors include the following example of an NP that should have been an AdjP:

(8) [$_{NX}$ Drei Tage lang] versuchte Joergensen ...
three days long tried Joergensen
'Joergensen tried for three days to ...'

(9) [$_{ADJX}$ Drei Tage lang] versuchten hier
three days long tried here
Museumsarchitekten ...
museum architects
'Museum architects tried for three days to ...'

The proposal in Dickinson and Meurers (2003) is more elaborate than our implementation here, but it is certainly possible to extend our setup further. We have shown that a basic but flexible query environment is quick to set up in Prolog. Prolog makes a suitable tool for corpus investigation and manipulation because it is a general-purpose programming language that by its very nature excels at programming in terms of relations and non-deterministic search.

4 Performance

With ever growing corpora, speed of query evaluation becomes a relevant issue. To give an idea of the performance of our straightforward use of Prolog, Table 3 shows wall-clock times of selected tasks.[4]

The uncompiled corpus of 45k sentences (∼1.8M Prolog facts) loads in about half a minute, but using precompiled prolog code – an option many implementations offer – reduces this to 3 seconds. The bottom of the table gives the time it takes to calculate the number of solutions for queries described in the previous section, plus `lookup/1` which returns once for each node in the corpus. As can be seen, queries are generally fast, except for those that involve calculating the yield. The use of memoization or even pre-computation would speed these queries up. Memory consumption is also moderate: even with `record_str_analyses/0`, the system runs in around 0.5Gbytes of RAM.

As an indication of the scalability of our approach, we note that we (Bouma et al., 2010) have run queries on dependency parsed corpora of around 40M words (thus 40M facts). Loading such a corpus takes about 10 minutes (or under 1 minute when precompiled) and uses 13GByte on a 64bit machine. Because of first-argument indexing on sentence ids, time per answer does not increase noticeably. We conclude that the approach in this paper scales to at least medium-large corpora. Scaling to even larger corpora remains a topic for future investigation. Possible solutions involve connecting Prolog to an external database, or (as a low-tech alternative) sequential loading of parts of the corpus.

[4]Test machine specifications: 1.6Ghz Intel Core 2 Duo, 2GBytes RAM, SWI-prolog v5.6 on 32-bit Ubuntu 8.04

Task	# Solutions	Time
Loading & indexing corpus		31s
Loading & indexing compiled corpus		3s
`lookup/1`	1741889	2s
`yields_ord/2`	1741889	81s
`spans/3`	1741889	87s
`bigr_in_nonrec_NP/2`	160968	80s
`i_within_i/2`	19	1s
`has_root/2`	50820	5s
`ana_ant_gf/2`	4184	1s
`ana_ant_gf_rev/2`	8471	9s
`record_str analyses/0`	1	101s
`inconsistency/2`	178	1s

Table 3: Wall-clock times of selected tasks.

5 Conclusions

In this paper, we hope to have shown the merits of Prolog as a language for corpus exploitation with the help of a range of corpus tasks. It is a flexible and effective language for corpus programming. The fact that most Prolog code needed for our demonstrations is in this paper makes this point well. Having said that, it is clear that the approach demonstrated in this paper is not a complete replacement of dedicated query environments that target non-programmers. In depth comparison with alternatives – corpus query environments, general-purpose language libraries, etc. – is beyond the scope of this paper, but see Bouma (2010) for a comparison of Prolog's performance and expressiveness with TIGERSearch on number of canonical queries.

Future work will include the investigation of techniques from constraint-based programming to make formulating queries less dependent on the procedural semantics of Prolog and the exploitation of corpora that cannot be fitted into working memory.

Our studies thus far have resulted not only in queries and primary code, but also in conversion scripts, auxiliary code for pretty printing, etc. We intend to collect all these and make these available on-line, so as to help interested other researchers to use Prolog in corpus investigations and to facilitate reproducibility of studies relying on this code.

References

Gerlof Bouma, Lilja Øvrelid, and Jonas Kuhn. 2010. Towards a large parallel corpus of cleft constructions. In *Proceedings of LREC 2010*, pages 3585–3592, Malta.

Gerlof Bouma. 2010. Syntactic tree queries in Prolog. In *Proceedings of the Fourth Linguistic Annotation Workshop*, pages 212–217, Uppsala.

Thorsten Brants. 1997. The NEGRA export format. Technical report, Saarland University, SFB378.

Markus Dickinson and Detmar Meurers. 2003. Detecting inconsistencies in treebanks. In *Proceedings of the Second Workshop on Treebanks and Linguistic Theories (TLT 2003)*, Växjö.

Jack Hoeksema and Donna Jo Napoli. 1990. A condition on circular chains: A restatement of i-within-i. *Journal of Linguistics*, 26(2):403–424.

Stephan Kepser. 2003. Finite structure query - a tool for querying syntactically annotated corpora. In *Proceedings of EACL 2003*, pages 179–186.

Esther König, Wolfgang Lezius, and Holger Voormann. 2003. Tigersearch 2.1 user's manual. Technical report, IMS Stuttgart.

Catherine Lai and Steven Bird. 2004. Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasion Language Technology Workshop*, Sydney.

Lee Naish. 1996. Higher-order logic programming in Prolog. Technical Report 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, February.

Ulf Nilsson and Jan Maluszynski. 1998. *Logic, programming and Prolog*. John Wiley & Sons, 2nd edition.

Ron Smyth. 1994. Grammatical determinants of ambiguous pronoun resolution. *Journal of Psycholinguistic Research*, 23:197–229.

Heike Telljohann, Erhard Hinrichs, Sandra Kübler, and Heike Zinsmeister. 2006. Stylebook for the Tübingen treebank of written German (TüBa-D/Z). revised version. Technical report, Seminar für Sprachwissenschaft, Universität Tübingen.

Andreas Witt. 2005. Multiple hierarchies: New aspects of an old solution. In Stefanie Dipper, Michael Götze, and Manfred Stede, editors, *Heterogeneity in Focus: Creating and Using Linguistic Databases*, Interdisciplinary Studies on Information Structure (ISIS) 2, pages 55–86. Universitätsverlag Potsdam, Potsdam.