Mining Syntactically Annotated Corpora with XQuery

Gosse Bouma and **Geert Kloosterman**
Information Science
University of Groningen
The Netherlands
g.bouma|g.j.kloosterman@rug.nl

Abstract

This paper presents a uniform approach to data extraction from syntactically annotated corpora encoded in XML. XQuery, which incorporates XPath, has been designed as a query language for XML. The combination of XPath and XQuery offers flexibility and expressive power, while corpus specific functions can be added to reduce the complexity of individual extraction tasks. We illustrate our approach using examples from dependency treebanks for Dutch.

1 Introduction

Manually annotated treebanks have played an important role in the development of robust and accurate syntactic analysers. Now that such parsers are available for various languages, there is a growing interest in research that uses automatically annotated corpora. While such corpora are not error-free, the fact that they can be constructed relatively easily, and the fact that they can be an order of magnitude larger than manually corrected treebanks, makes them attractive for several types of research. Syntactically annotated corpora have succesfully been used to acquire lexico-semantic information (Lin and Pantel, 2001; Snow et al., 2005), for relation extraction (Bunescu and Mooney, 2005), in IR (Cui et al., 2005), and in QA (Katz and Lin, 2003; Mollá and Gardiner, 2005).

What these tasks have in common is the fact that they all operate on large amounts of data extracted from syntactically annotated text. Tools to perform this task are often developed with only a single application in mind (mostly corpus linguistics) or are developed in an ad-hoc fashion, as part of a specific application.

We propose a more principled approach, based on two observations:

- XML is widely used to encode syntactic annotation. Syntactic annotation is not more complex that some other types of information that is routinely stored in XML. This suggests that XML technology can be used to process syntactically annotated corpora.
- XQuery is a query language for XML data. As such, it is the obvious choice for mining syntactically annotated corpora.

The remainder of this paper is organised as follows. In the next section, we present the Alpino treebank format, which we use for syntactic annotation. The Alpino parser has been used to annotate large corpora, and the results have been used in a number of research projects.[1]

In section 3, we discuss the existing approaches to data extraction from Alpino corpora. We note that all of these have drawbacks, either because they lack expressive power, or because they require a serious amount of programming overhead.

In section 4, we present our approach, starting from a relatively straightforward corpus linguistics task, that requires little more than XPath, and ending with a more advanced relation extraction task,

[1] See `www.let.rug.nl/~vannoord/research.html`

Proceedings of the Linguistic Annotation Workshop, pages 17–24,
Prague, June 2007.

that requires XQuery. We demonstrate that much of the complexity of advanced tasks can be avoided by providing users with a corpus specific module, that makes available common concepts and functions.

2 The Alpino Treebank format

As part of the development of the Alpino parser (Bouma et al., 2001), a number of manually annotated dependency treebanks have been created (van der Beek et al., 2002). Annotation guidelines were adopted from the Corpus of Spoken Dutch (Oostdijk, 2000), a large corpus annotation project for Dutch. In addition, large corpora (e.g. the 80M word Dutch CLEF[2] corpus, the 500M word Twente News corpus[3], and Dutch Wikipedia[4]) have been annotated automatically. Both types of treebanks have been used for corpus linguistics (van der Beek, 2005; Villada Moirón, 2005; Bouma et al., 2007). The automatically annoted treebanks have been used for lexical acquisition (van der Plas and Bouma, 2005), and form the core of a Dutch QA system (Bouma et al., 2005).

The format of Alpino dependency trees is illustrated in figure 1. The (somewhat simplified) XML for this tree is in fig. 2. Nodes in the tree are labeled with a dependency relation and a category or POS-tag. Furthermore, the begin and end position of constituents is represented in attributes,[5] and the root and word form of terminal nodes is encoded. Note that heads do not have their dependents as children, as is the case in most dependency tree formats. Instead, the head is a child of the constituent node if which it is the head, and its dependents are siblings of the head. Finally, trees may contain index nodes (indicated by indices in bold in the graphical representation and by the `index` attribute in the XML) to indicate 'secondary' edges. The subject *Alan Turing* in fig. 2 is a subject of the passive auxiliary *word*, but also a direct object of the verb *aan_tref*. Thus, Alpino dependency trees are actually graphs.

A large syntactically annotated corpus tends to

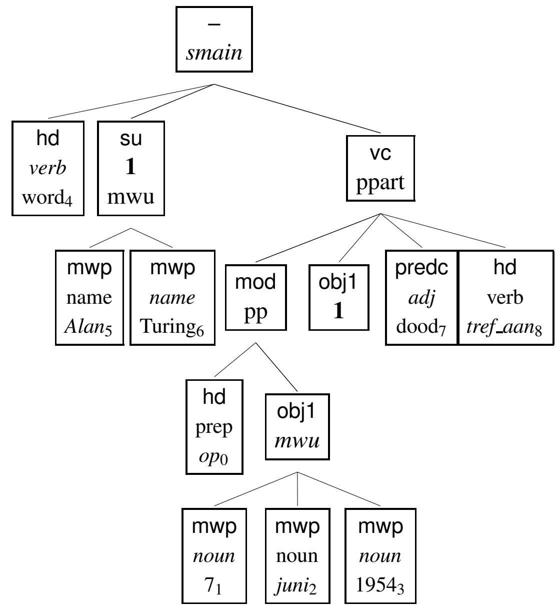

Figure 1: Op 7 juni 1954 werd Alan Turing dood aangetroffen (*On June 7, 1954, Alan Turing was found dead*)

give rise to even larger volumes of XML. To support efficient storage and retrieval of XML data, a set of tools has been developed for compression of XML data (using dictzip[6]) and for efficient visualisation and search of data in compressed XML files. The tools are described in more detail at the Alpino website.[7]

3 Existing approaches to extraction

Users have taken quite different approaches to corpus exploration and data extraction.

- For corpus exploration, Alpino `dtsearch` is the most widely used tool. It allows XPath queries to be matched against trees in a treebank. The result can be a visual display of trees with matching nodes highlighted, but alternative outputs are possible as well. Examples of how XPath can be used for extraction are presented in the next section.

- For relation extraction (i.e. finding symptoms of diseases), the Alpino system itself has been

[2] `www.clef-campaign.org`

[3] `www.vf.utwente.nl/~druid/TwNC/TwNC-main.html`

[4] `nl.wikipedia.org`

[5] Note that constituents may be discontinuous, and thus, the yield of a constituent may not contain every terminal node between `begin` and `end`. See also section 4.2.

[6] `www.dict.org`

[7] `www.let.rug.nl/~vannoord/alp/Alpino/TreebankTools.html`

```
<node begin="0" cat="smain" end="9" rel="--">
  <node begin="4" end="5" pos="verb" rel="hd" root="word" word="werd"/>
  <node begin="5" cat="mwu" end="7" index="1" rel="su">
    <node begin="5" end="6" pos="name" rel="mwp" neclass="PER" root="Alan" word="Alan"/>
    <node begin="6" end="7" pos="name" rel="mwp" neclass="PER" root="Turing" word="Turing"/>
  </node>
  <node begin="0" cat="ppart" end="9" rel="vc">
    <node begin="0" cat="pp" end="4" rel="mod">
      <node begin="0" end="1" pos="prep" rel="hd" root="op" word="Op"/>
      <node begin="1" cat="mwu" end="4" rel="obj1">
        <node begin="1" end="2" pos="noun" rel="mwp" root="7" word="7"/>
        <node begin="2" end="3" pos="noun" rel="mwp" root="juni" word="juni"/>
        <node begin="3" end="4" pos="noun" rel="mwp" root="1954" word="1954"/>
      </node>
    </node>
    <node begin="5" end="7" index="1" rel="obj1"/>
    <node begin="7" end="8" pos="adj" rel="predc" root="dood" word="dood"/>
    <node begin="8" end="9" pos="verb" rel="hd" root="tref_aan" word="aangetroffen"/>
  </node>
</node>
```

Figure 2: XML encoding of the Alpino depedency tree in fig. 1

used. It provides functionality for converting dependency trees in XML into a Prolog list of dependency triples. The full functionality of Prolog can then be used to do the actual extraction.

- Alternatively, one can use XSLT to extract data from the XML directly. As XSLT is primarily intended for transformations, this tends to give rise to complex code.
- Alternatively, a general purpose scripting or programming language such as Perl or Python, with suitable XML support, can be used. As in the Alpino/Prolog case, this has the advantage that one has a full programming language available. A disadvantage is that there is no specific support for working with dependency trees or triples.

None of the approaches listed above is optimal. XPath is suitable only for identifying syntactic patterns, and does not offer the possibility of extraction of elements (i.e. it has no *capturing* mechanism). The other three approaches do allow for both matching and extraction, but they all require skills that go considerably beyond conceptual knowledge of the treebank and some basic knowledge of XML.

Another disadvantage of the current situation is that there is little or no sharing of solutions between users. Yet, different applications tend to encounter the same problems. For instance, multiword expressions (such as *Alan Turing* or *7 juni 1954*) are encoded as trees, dominated by a `cat='mwu'` node. An extraction task that requires names to be extracted must thus take into account the fact that names can be both nodes with a label `pos='name'` as well as `cat='mwu'` nodes (dominating a `pos='name'`). The situation is further complicated by the fact that individual parts of a name, such as *Alan* in *Alan Turing*, should normally not be matched. Similar problems arise if one wants to match e.g. finite verbs (there is no single attribute which expresses tense) or NPs (the `cat='np'` attribute is only present on complex NPs, not on single words). A very frequent issue is the proper handling of index nodes. Searching for the object of the verb *tref_aan* in fig. 2 requires that one finds the node in the tree that is coindexed with the `rel='obj1'` node with index **1**. This is a challenge in all approaches listed above, except for Alpino/Prolog, which solves the problem by converting trees to sets of dependency triples.

Some of the problems mentioned above could be solved by introducing more and more fine-grained attributes (i.e. a separate attribute for tense, assigning both a category and a POS-tag to (non-head) terminal-nodes, etc.) or by introducing unary branching nodes. This has the obvious drawback of introducing redundancy in the encoding, would mean another departure from the usual conception of dependency trees (in the case unary branching is introduced), and may still not cover all distinctions that users need to make. Also, finding the content of an index-node cannot be solved in this way.

One might consider moving to a radically different treebank format, such as Tiger XML[8] for instance, in which trees are basically a listing of nodes, with non-terminal nodes dominating a number of edge elements that take (the index of) other nodes as value. Note, however, that most of the problems mentioned above refer to linguistic concepts, and thus are unlikely to be solved by changing the architecture of the underlying XML representation.

4 XQuery and XPath

Two closely related standards for processing XML documents are XSLT[9] and XQuery[10] . Both make use of XPath[11], the XML language for locating parts of XML documents. While XSLT is primarily intended for transformations of documents, XQuery is primarily intended for extraction of information from XML databases. XQuery is in many respects similar to SQL and is rapidly becoming the standard for XML database systems.[12] A distinctive difference between the XSLT and XQuery is the fact that XSLT documents are themselves XML documents, whereas this is not the case for XQuery. This typically makes XQuery more concise and easier to read than XSLT.[13]

These considerations made us experiment with XQuery as a language for data extraction from syntactically annotated corpora. Similar studies were carried out by Cassidy (2002) (for an early version of XQuery) and Mayo et al. (2006), who compare the NITE Query Language and XQuery. Below, we first illustrate a task that requires use of XPath only, and then move on to tasks that require the additional functionality of XQuery.

[8] www.ims.uni-stuttgart.de/projekte/TIGER/

[9] www.w3.org/TR/xslt20

[10] www.w3.org/TR/xquery

[11] www.w3.org/TR/xpath20

[12] e.g. exist.sourceforge.net, monetdb.cwi.nl, www.oracle.com/database/berkeley-db/xml

[13] See Kay (2005) for a thorough comparison.

4.1 Corpus exploration with XPath

As argued in Bouma and Kloosterman (2002), XPath provides a powerful query language for formulating linguistically relevant queries, provided that the XML encoding of the treebank reflects the syntactic structure of the trees.

Inherent reflexive verbs, for instance, are verbal heads with a `rel='se'` dependent. A verb with an inherently reflexive can therefore be found as follows (remember that in Alpino dependency trees, dependents are actually siblings of the head):

```
//node[@pos="verb"
   and @rel="hd"
   and ../node[@rel="se"]
  ]
```

The double slash ('//') ensures that we search for nodes anywhere within the XML document. The material in brackets (`[ ]`) can be used to specify additional constraints that matching nodes have to meet. The `@`-sign is used to refer to attributes of an element. The double dots ('..') locate the parent element of an XML element. Children of an element are located using the single slash ('/') operator. The two can be combined to locate siblings.

Comparison operators are available to compare e.g. attributes that have a numeric value. The following XPath query identifies cases where the reflexive precedes the subject:

```
//node[@pos="verb"
   and @rel="hd"
   and ../node[@rel="se"]/@begin <
          ../node[@rel="su"]/@begin
  ]
```

Note that we can also use the '/' to locate attributes of an element, and that the `begin` attribute encodes the initial string position of a constituent.

Reflexives preceding the subject are a marked option in Dutch. We may contrast matching verbs with verbs matching the following expression:

```
//node[@pos="verb"
    and @rel="hd"
    and ../node[@rel="se"]/@begin >
            ../node[@rel="su"]/@begin
    and not(../node[@rel="su"]/@begin="0")
  ]
```

Here we have simply reversed the comparison operator. As we want to exclude from consideration cases where the subject precedes the finite verb (e.g. is in sentence-initial position), we have added a negative constraint with this effect.

REFL-SU		SU-REFL		verb (*gloss*)
%	#	%	#	
94.3	33	5.7	2	vorm (*to shape*)
91.7	11	8.3	1	ontvouw (*to unfold*)
74.1	234	25.9	82	doe_voor (*to happen*)
73.5	36	26.5	13	teken_af (*to form*)
58.8	10	41.2	7	wreek (*to take revenge*)
57.1	44	42.9	33	voltrek (*to take place*)
56.0	42	44.0	33	verzamel (*to assemble*)
54.6	309	45.4	257	bevind (*to be located*)
50.0	18	50.0	18	dring_op (*to impose*)
48.3	58	51.7	62	dien_aan (*to announce*)

Table 1: Relative frequency of REFL-SU vs SU-REFL word order

Using the two queries above to search one year of newspaper text, we can collect the outcome and compute, for a given verb, the relative frequency of REFL-SU vs. SU-REFL order for non-subject initial sentences in Dutch. A sample of verbs that have a high percentage of REFL-SU occurrences, is given in table 1. The result confirms an observation in Haesereyn et al. (1997), that REFL-SU word order occurs especially with verbs having a somewhat 'bleeched semantics' and expressing that something exists or comes into existence.

It should be noted that XPath offers considerable more possibilities than what is illustrated here. XPath 2.0 in particular is an important step forward for linguistic search, as it includes far more functionality for string processing (i.e. tokenization and regular expressions) than its predecessors. Bird et al. (2006) propose an extension of XPath 1.0 for linguistic queries. The intuitive notation they introduce might be useful for some users. However, the examples they concentrate on (all having to do with linear order) presuppose trees without 'crossing branches'. The introduction of `begin` and `end` attributes in the Alpino format makes it possible to handle such queries for dependency trees (with crossing branches) as well, and furthermore, does not require an extension of XPath.

4.2 Data Extraction with XQuery

The kind of explorative corpus search for which XPath is ideally suited is supported by most other treebank query languages as well, although not all alternatives offer the same expressive power. There are many applications, however, in which it is necessary to extract more than just (root forms of) matching nodes. XQuery offers the functionality that is required to perform arbitrary extraction.

XQuery programs consist of so-called FLWOR expressions (`for, let, where, order by, return`, not all parts are required). The example below illustrates this. Assume we want to extract from a treebank all occurrences of names, along with their named entity class. The following XQuery script covers the base case.

```
for $name in
   collection('ad1994')//node[@pos="name"]

let $nec := string($node/@neclass)

return
  <term nec="{$nec}">
    {string($name/@word)}
  </term>
```

The `for`-statement locates the nodes to be processed. Nodes are located by XPath expressions. The *collection*-predicate defines the directory to be processed. For every document in the collection, nodes with a POS-attribute *name* are processed. We use a `let`-statement to assign the variable `$nec` is assigned the string value of the `neclass`-attibute (which indicates the named entity class of the name). The `return`-statement returns for each matching node an XML element containing the string value of the word attribute of the name, as well as an attribute indicating the named entity class.

The complexity of XQuery scripts can increase considerably, depending on the complexity of the underlying XML data and the task being performed. One of the most interesting features of XQuery is the possibility to define functions. They can be used to enhance the readibility of code. Furthermore, functions can be collected in modules, and thus can be reused across applications.

For Alpino treebanks, for instance, we have implemented a module that covers concepts and tasks that are needed frequently. As pointed out above, names in the Alpino treebank are not just single nodes, but, in case a name consists of two or more words, can also consist of multiple `node[@pos='name']` elements, with a `node[@cat='mwu']` as parent. This motivates the introduction of a `name` and `neclass` function, as shown in fig. 3. Assuming that the `alpino` module has been imported, we can now write a better name extraction script:

```
for $name in
   collection('ad1994')//node

where alpino:name($name)

return
  <term nec="{alpino:neclass($name)}">
    {alpino:yield{$name)}
  </term>
```

As we are matching with non-terminal nodes as well, we need to take into account that it no longer suffices to return the value of `word` to obtain the yield of a node. As this situation arises frequently as well, we added a `yield` function (see fig. 3). It takes a node as argument, collects all descendant `node/@word` attribute values in the variable `$words`, sorted by the `begin` value of their `node` element. The `yield` function returns the string concatenation of the elements in `$words`, separated by blanks. Note that this solution also gives the correct result for discontinuous constituents.

We used a wrapper around the XQuery processor Saxon[14] to execute XQuery scripts directly on compacted corpora. The result is output such as:

```
<term nec="ORG">PvdA</term>
<term nec="LOC">Atlantische Oceaan</term>
```

A more advanced relation extraction example is given in fig. 4. It is a script for extraction of events involving the death of a person from a syntactically annotated corpus (Dutch wikipedia in this case). It will return the name of the person who died, and, if these can be found in the same sentence, the date, location, and cause of death.[15] The script makes heavy use of functions from the `alpino` module that were added to facilitate relation extraction. The `selector-of` function defines the 'semantic head' of a phrase. This is either the sibling marked `rel='hd'`, or (for nodes that are themselves heads) the head of the mother. For appositions and conjuncts, it it the selector of the head. Note that the last case involves a recursive function call. Similarly, the semantic role is normally identical to the value of the `rel`-attribute, but we go up one additional level for heads, appositions and conjuncts. The value of `$resolved` is given by the `resolve-index` function shown in fig. 3, i.e. if a node is just an index (as is the case for the object of *aan_tref* in fig. 1), the 'antecedent' node is returned. In all other cases, the node itself is returned. Date and place are found using functions for locating the date and place dependents of the verb. Finally, relevant events are found using the `die-verb` and `kill-verb` functions.

Some examples of the output of the extraction script are (i.e. John Lennon was killed on December 8, 1980, and Erasmus died in Basel on July, 12, 1536):

```
<died-how place="nil" file="1687-98"
   person="John Lennon" cause="vermoord"
   date="op 8 december 1980"/>
<died-how place="in Bazel" file="20336-37"
   person="Erasmus"  cause="overlijd"
   date="op 12 juli 1536"/>
```

The functions illustrated in the two examples can be used for a range of similar data extraction tasks, whether these are intended for corpus linguistics research or as part of an information extraction system. The definition of corpus specific functions that cover frequently used syntactic and semantic concepts allows the application specific code to be relatively compact and straightforward. In addition, code which builds upon well tested corpus specific functions tends to give more accurate results than code developed from scratch.

5 Conclusions

In this paper, we have presented an approach to mining syntactically corpora that uses standard XML technology. It can be used both for corpus exploration as well as for information extraction tasks. By providing a corpus specific module, the complexity of such tasks can be reduced. By adopting standard XML languages, we can benefit optimally from the fact that these are far more expressive than what is provided in application specific languages or tools. In addition, there is no shortage of tools or platforms supporting these languages. Thus, development of corpus specific tools can be kept at a minimum.

[14] `www.saxonica.com`

[15] Questions about such facts are relatively frequent in Question Answering evaluation tasks.

```
module namespace alpino="alpino.xq" ;

declare function name($constituent as element(node)) as xs:boolean
  { if   ( $constituent[@pos='name'] or
           $constituent[@cat = 'mwu']/node[@neclass='PER'] )
    then fn:true()
    else fn:false()
  };
declare function neclass($constituent as element(node)) as xs:string
  { if   $constituent[@neclass]
    then fn:string($constituent/@neclass)
    else if   $constituent/node[@neclass]
         then fn:string($constituent/node[1]/@neclass)
  };
declare function alpino:yield($constituent as element(node)) as xs:string
  { let $words :=
          for      $leaf in $constituent/descendant-or-self::node[@word]
          order by number($leaf/@begin)
          return   $leaf/@word
    return string-join($words," ")
  };
declare function alpino:resolve-index($constituent as element(node))
        as element(node)
  { if   ( $constituent[@index and not(@pos or @cat)] )
    then $constituent/ancestor::alpino_ds/
              descendant::node
                 [@index = $constituent/@index and (@pos or @cat)]
    else $constituent
  };
```

Figure 3: XQuery module (fragment) for Alpino treebanks

```
for $node in collection('wikipedia')/alpino_ds//node

let $verb     := alpino:selector-of($node)
let $date     := if   ( exists(alpino:date-dependents($verb)) )
                 then alpino:yield(alpino:date-dependents($verb)[1])
                 else 'nil'
let $place    := if ( exists(alpino:location-dependents($verb)) )
                 then alpino:yield(alpino:location-dependents($verb)[1])
                 else 'nil'
let $cause    := if   ( $verb/../node[@rel="pc"]/node[@root="aan"] )
                 then alpino:yield($verb/../node[@rel="pc"])
                 else [[omitted]]
let $role     := alpino:semantic-role($node)
let $resolved := alpino:resolve-index($node)

where     alpino:person-node($resolved)
      and (    ( $role="su"    and alpino:die-verb($verb)  )
           or (  $role="obj1" and alpino:kill-verb($verb) )
          )

return
<died-how  file="{alpino:file-id($node)}" person="{alpino:root-string($resolved)}"
   cause="{$cause}"  date="{$date}" place = "{$place}" />
```

Figure 4: Extracting circumstances of the death of a person

References

Steven Bird, Yi Chen, Susan B. Davidson, Haejoong Lee, and Yifeng Zheng. Designing and evaluating an XPath dialect for linguistic queries. In *Proceedings of 22nd International Conference on Data Engineering (ICDE)*, 2006.

Gosse Bouma and Geert Kloosterman. Querying dependency treebanks in XML. In *Proceedings of the 3rd conference on Language Resources and Evaluation (LREC)*, Gran Canaria, 2002.

Gosse Bouma, Gertjan van Noord, and Robert Malouf. Alpino: Wide-coverage computational analysis of Dutch. In *Computational Linguistics in The Netherlands 2000*. Rodopi, Amsterdam, 2001.

Gosse Bouma, Ismail Fahmi, Jori Mur, Gertjan van Noord, Lonneke van der Plas, and Jörg Tiedeman. Linguistic knowledge and question answering. *Traitement Automatique des Langues*, 2(46): 15–39, 2005.

Gosse Bouma, Petra Hendriks, and Jack Hoeksema. Focus particles inside prepositional phrases: A comparison of Dutch, English, and German. *Journal of Comparative Germanic Linguistics*, 10(1), 2007.

Razvan Bunescu and Raymond Mooney. A shortest path dependency kernel for relation extraction. In *Proceedings of HLT/EMNLP*, pages 724–731, Vancouver, 2005.

Steve Cassidy. XQuery as an annotation query language: a use case analysis. In *Language Resources and Evaluation Conference (LREC)*, Gran Canaria, 2002.

Hang Cui, Renxu Sun, Keya Li, Min-Yen Kan, and Tat-Seng Chua. Question answering passage retrieval using dependency relations. In *Proceedings of SIGIR 05*, Salvador, Brazil, 2005.

W. Haesereyn, K. Romijn, G. Geerts, J. De Rooy, and M.C. Van den Toorn. *Algemene Nederlandse Spraakkunst*. Martinus Nijhoff Uitgevers Groningen / Wolters Plantyn Deurne, 1997.

Boris Katz and Jimmy Lin. Selectively using relations to improve precision in question answering. In *Proceedings of the workshop on Natural Language Processing for Question Answering (EACL 2003)*, pages 43–50, Budapest, 2003. EACL.

Michael Kay. Comparing XSLT and XQuery. In *Proceedings of XTech 2005*, Amsterdam, 2005. URL `www.idealliance.org/proceedings/xtech05`.

Dekan Lin and Patrick Pantel. Discovery of inference rules for question answering. *Natural Language Engineering*, 7:343–360, 2001.

Neil Mayo, Jonathan Kilgour, and Jean Carletta. Towards an alternative implementation of NXT query language via XQuery. In *Proceedings of the EACL Workshop on Multi-dimensional Markup in NLP*, Trento, 2006.

D. Mollá and M. Gardiner. Answerfinder - question answering by combining lexical, syntactic and semantic information. In *Australasian Language Technology Workshop (ALTW) 2004*, Sydney, 2005.

Nelleke Oostdijk. The Spoken Dutch Corpus: Overview and first evaluation. In *Proceedings of LREC 2000*, pages 887–894, 2000.

Rion Snow, Daniel Jurafsky, and Andrew Y. Ng. Learning syntactic patterns for automatic hypernym discovery. In Lawrence K. Saul, Yair Weiss, and Lon Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 1297–1304. MIT Press, Cambridge, MA, 2005.

L. van der Beek, G. Bouma, R. Malouf, and G. van Noord. The Alpino dependency treebank. In *Computational Linguistics in the Netherlands (CLIN) 2001*, Twente University, 2002.

Leonoor van der Beek. *Topics in Corpus Based Dutch Syntax*. PhD thesis, University of Groningen, Groningen, 2005.

Lonneke van der Plas and Gosse Bouma. Automatic acquisition of lexico-semantic knowledge for question answering. In *Proceedings of Ontolex 2005 – Ontologies and Lexical Resources*, Jeju Island, South Korea, 2005.

Begoña Villada Moirón. Linguistically enriched corpora for establishing variation in support verb constructions. In *Proceedings of the 6th International Workshop on Linguistically Interpreted Corpora (LINC-2005)*, Jeju Island, Republic of Korea, 2005.