# Algorithms for LINGUISTIC PROCESSING

**NWO PIONIER**

**Progress Report**

**August 2002**

# Algorithms for Linguistic Processing
# NWO PIONIER
# Progress Report

Leonoor van der Beek
Gosse Bouma
Jan Daciuk
Tanja Gaustad
Robert Malouf
Gertjan van Noord
Robbert Prins
Begoña Villada

Graduate School for Behavioral and Cognitive Neurosciences
Alfa-informatica
P.O. Box 716
NL 9700 AS Groningen

August, 2002

# Chapter 5

# The Alpino Dependency Treebank

## 5.1 Introduction

In this section we present the Alpino Dependency Treebank and the tools that we have developed to facilitate the annotation process. Annotation typically starts with parsing a sentence with the Alpino parser. The number of parses that is generated is reduced through interactive lexical analysis and constituent marking. A tool for on line addition of lexical information facilitates the parsing of sentences with unknown words. The selection of the best parse is done efficiently with the parse selection tool. At this moment, the Alpino Dependency Treebank consists of about 6,000 sentences of newspaper text that are annotated with dependency trees. The corpus can be used for linguistic exploration as well as for training and evaluation purposes.

A syntactically annotated corpus is needed to train disambiguation models for computational grammars, as well as to evaluate the performance of such models, and the coverage of computational grammars. For this purpose we have started to develop the Alpino Dependency Treebank.

The treebank consists of sentences from the newspaper (`cdbl`) part of the Eindhoven corpus (Uit den Boogaard 1975). The sentences are each assigned a dependency structure, which is a relatively theory independent annotation format. The format is taken from the corpus of spoken Dutch (CGN)[1] (Oostdijk 2000), which in turn based its format on the Tiger Treebank (Skut 1997). In section 5.2 we go into the characteristics of dependency structures and motivate our choice for this annotation format.

Section 5.3 is the central part of this paper. Here we explain the annotation method as we use it, the tools that we have developed, the advantages and the shortcomings of the system. It starts with a description of the parsing process that is at the beginning of the annotation process. Although it is a good idea to start annotation with parsing (building dependency trees manually is very time consuming and error prone), it has one main disadvantage: ambiguity. For a sentence of average length typically a set of hundreds or even thousands of parses is generated. Selection of the best parse from this large set of possible parses is time intensive.

The tools that we present in this paper aim at facilitating the annotation process

[1] `http://lands.let.kun.nl/cgn`

and making it less time consuming. We present two tools that reduce the number of parses generated by the parser and a third tool that facilitates the addition of lexical information during the annotation process. Finally a parse selection tool is developed to facilitate the selection of the best parse from the reduced set of parses.

The Alpino Dependency Treebank is a searchable treebank in an XML format. In section 5.4 we present examples illustrating how the standard XML query language XPath can be used to search the treebank for linguistically relevant information. In section 5.5 we explain how the corpus can be used to evaluate the Alpino parser and to train the probabilistic disambiguation component of the grammar. We end with conclusions and some pointers to future work in 5.7.

## 5.2 Dependency Trees

The meaning of a word or a sentence is represented in standard HPSG by semantic representations that are added to lexical entries and phrases. Semantic principles define the construction of a semantic structure from these representations. In Alpino we have added the DT features with which we build a dependency tree instead.

Dependency structures represent the grammatical relations that hold in and between constituents. On the one hand they are more abstract than syntactic trees (word order for example is not expressed) and on the other hand they are more explicit about the dependency relations. Indices denote that constituents may have multiple (possibly different) dependency relations with different words. Fig. 5.1 shows the dependency tree for the sentence *Kim wil weten of Anne komt*. The dependency relations are the top labels in the boxes. In addition, the syntactic category, lexical entry and string position are added to each leaf. The index **1** indicates that *Kim* is the subject of both *wil* (wants) and *weten* (to know).

The main advantage of this format is that it is relatively theory independent, which is important in a grammar engineering context. A second advantage is that the format is similar to the format CGN uses (and that they in turn based on the Tiger Treebank), which allowed us to base our annotation guidelines on theirs (Moortgat, Schuurman and van der Wouden 2001). The third and last argument for using dependency structures is that it is relatively straightforward to perform evaluation of the parser on dependency structures: one can compare the automatically generated dependency structure with the one in the treebank and calculate statistical measures such as F-score based on the number of dependency relations that are identical in both trees (Carroll, Briscoe, and Sanfilippo, 1998).

## 5.3 The annotation process

The annotation process is roughly divided into two parts: we first parse a sentence with the Alpino parser and then select the best parse from the set of generated parses. Several tools that we have developed and implemented in Hdrug, a graphical environment for natural language processing (van Noord and Bouma 1997), facilitate the two parts of the annotation process. In section 5.3.1 we present an

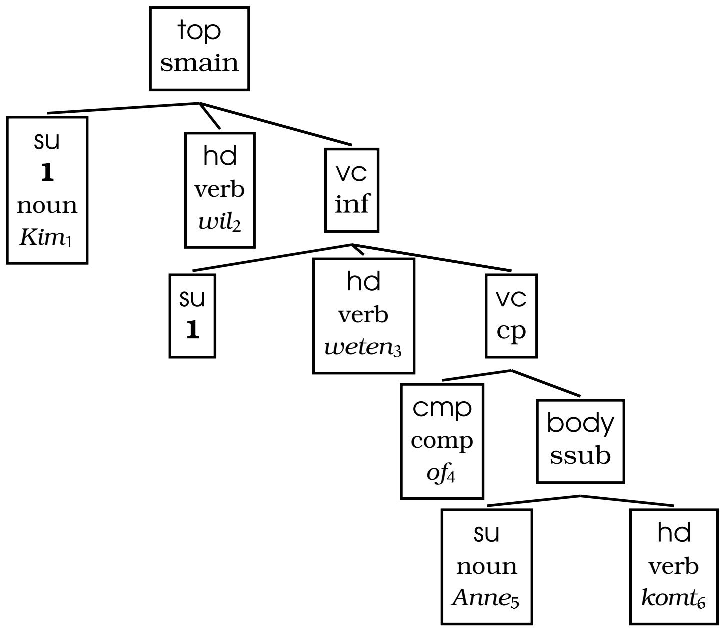

Figure 5.1: Dependency tree voor de zin *Kim wil weten of Anne komt*

interactive lexical analyzer, a constituent marker and a tool for temporary addition of lexical information. The parse selection tool is described in in section 5.3.5.

### 5.3.1 Parsing

The annotation process typically starts with parsing a sentence from the corpus with the Alpino parser. This is a good method, since building up dependency trees manually is extremely time consuming and error prone. Usually the parser produces a correct or almost correct parse. If the parser cannot build a structure for a complete sentence, it tries to generate as large a structure as possible (e.g. a noun phrase or a complementizer phrase). The main disadvantage of parsing is that the parser produces a large set of possible parses (see fig.5.2). This is a well known problem in grammar development: the more linguistic phenomena a grammar covers, the greater the ambiguity per sentence. Because selection of the best parse from such a large set of possible parses is time consuming, we have tried to reduce the set of generated parses. The interactive lexical analyzer and the constituent marker restrict the parsing process which results in reduced sets of parses. A tool for on line addition of lexical information makes parsing of sentences with unknown words more accurate and efficient.

### 5.3.2 Interactive lexical analysis

The interactive lexical analyzer is a tool that facilitates the selection of lexical entries for the words in a sentence. It presents all possible lexical entries for all words in the sentence to the annotator. He or she may mark them as correct, good or bad.

- Correct Parse must include it

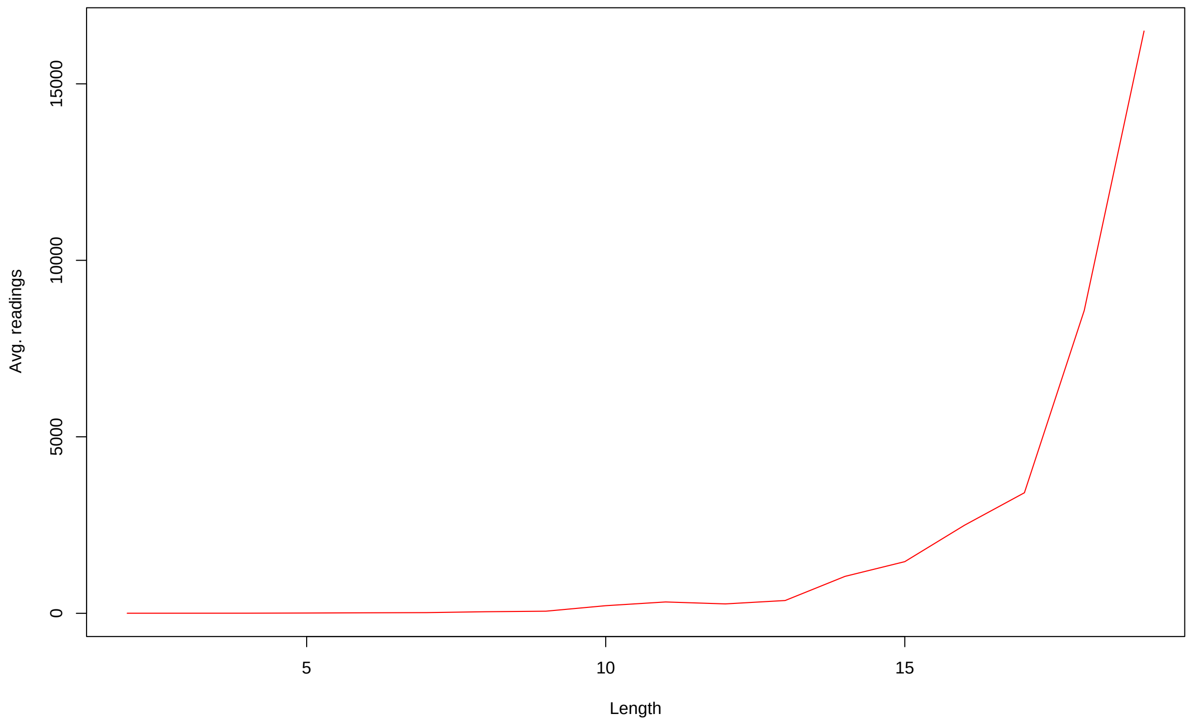

Figure 5.2: Number of parses generated per sentence by the Alpino parser

- Good  Parse may include it
- Bad  Parse may not include it

One *correct* mark for a particular lexical entry automatically produces *bad* marks for all other entries for the same word. The parser uses the reduced set of entries to generate a significantly smaller set of parses in less processing time.

### 5.3.3 Constituent Marking

The annotator can mark a piece of the input string as a constituent by putting square brackets around the words. The type of constituent can be specified after the opening bracket. The parser will only produce parses that have a constituent of the specified type at the string position defined in the input string. Even if the parse cannot generate the correct parse, it will produce parses that are likely to be close to the best possible parse, because they do oblige to the restrictions posed on the parses by the constituent marker.

Constituent marking has some limitations. First, the specified constituent borders are defined on the syntactic tree, not the dependency tree (dependency structures are an extra layer of annotation that is added to the syntactic structure). Using the tool therefore requires knowledge of the Alpino grammar and the syntactic trees that it generates.

Second, specification of the constituent type is necessary in most cases, especially for disambiguating prepositional phrase attachments. As shown in fig. 5.3, a noun phrase and a prepositional phrase can form a constituent on different levels.

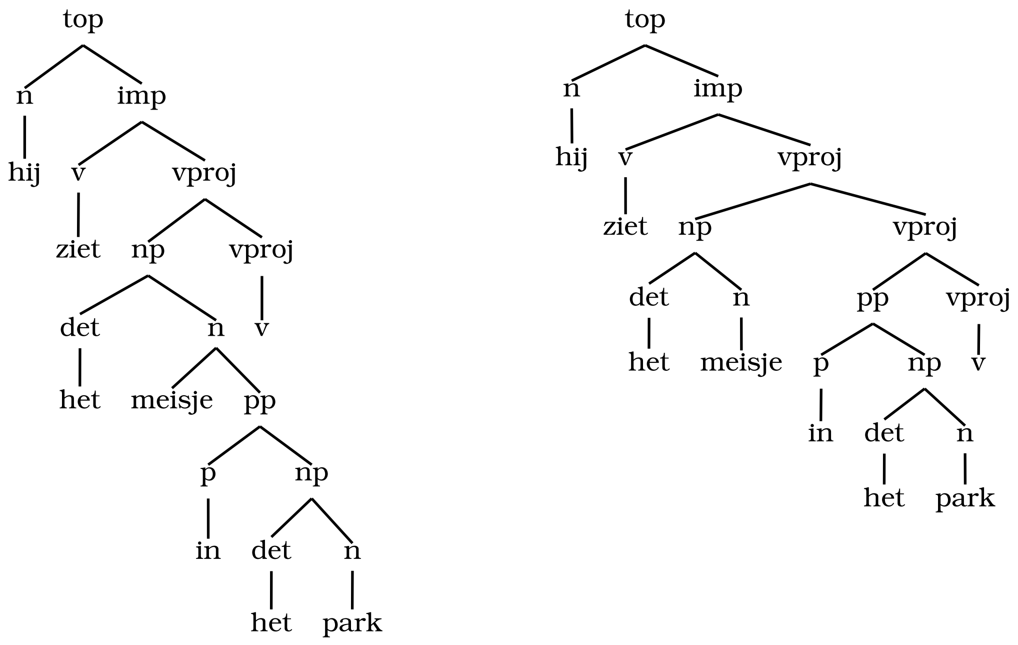

Figure 5.3: PP attachment ambiguity in Alpino

The two phrases can form either a noun phrase or a verbal projection with an empty verb (which is used in the grammar to account for verb second). The first structure corresponds to a dependency structure with a noun phrase internal prepositional modifier, the second corresponds to a dependency tree in which the prepositional phrase is a modifier on the sentence level. Marking the string `het meisje in het park` as a constituent without further specification does not disambiguate between the two readings: in both readings the string is a constituent. One has to specify that the string should be a noun phrase, not a verbal projection. This specification of the constituent type requires even more knowledge of the grammar. If one specifies a constituent type that cannot be formed at the denoted string position, the parser treats the specification as an illegal character, skips it and generates partial parses only.

## 5.3.4 Addition of lexical information

Alpino is set up as a broad coverage parser. The goal is to build an analyzer of unrestricted text. Therefore a large lexicon has been created and extensive unknown word heuristics have been added to the grammar. Still, it is inevitable that the parser will come across unknown words that it cannot handle yet. Verbs are used with extra or missing arguments, Dutch sentences are mingled with foreign words, spelling mistakes make common words unrecognizable. In most cases, the parser will either skip such a word or assign an inappropriate category to it. The only way to make the system correctly use the word, is to add a lexical entry for it in the lexicon.

Adding new words to the lexicon costs time: one has to write the entry, save the new lexicon and reload it. It would be far more efficient to add all new words one

| | | |
|---|---|---|
| top:hd | = | v *wil* |
| top:su | = | n *Kim* |
| top:vc:hd | = | v *weet* |
| top:vc:su | = | n *Kim* |
| top:vc:vc:cmp | = | comp *of* |
| top:vc:vc:body:hd | = | v *kom* |
| top:vc:vc:body:su | = | n *Anne* |

Figure 5.4: Set of dependency paths for the sentence *Kim wil weten of Anne komt*

comes across during an annotation session at once, avoiding spurious reloadings. Furthermore, not all unknown words the parser finds should be added to the lexicon. One would want to use misspelled words and verbs with an incorrect number of arguments only once to build a parse with.

Alpino has temporary, on line addition of lexical information built in for this purpose. Unknown words can temporarily be added to the lexicon with the command `add_tag` or `add_lex`. Like the words in the lexicon, this new entry should be assigned a feature structure. `add_tag` allows the user to specify the lexical type as the second argument. However types may change and especially for verbs it is sometimes hard to decide which of the subcategorization frames should be used. For that reason the command `add_lex` allows us to assign to unknown words the feature structure of a similar word, that could have been used on that position. The command `add_lex stoel tafel` for instance assigns all feature structures associated with *tafel* to the word *stoel*. The command `add_lex zoen slaap` assigns *zoen* all feature structures of *slaap*, including imperative and 1st person singular present for all sub-categorization frames of *slapen*. The lexical information is automatically deleted when the annotation session is finished.

### 5.3.5 Selection

Although the number of parses that is generated is strongly reduced through the use of different tools, the parser usually still produces a set of parses. Selection of the best parse (i.e. the parse that needs the least editing) from this set of parses is facilitated by the parse selection tool. This design of this tool is based on the SRI Treebanker (Carter 1997).

The parse selection takes as input a set of dependency paths for each parse. A dependency path specifies the grammatical relation of a word in a constituent (e.g. head (hd) or determiner (det)) and the way the constituent is embedded in the sentence. The representation of a parse as a set of dependency paths is a notational variant of the dependency tree. The set of dependency triples that corresponds to the dependency tree in fig. 5.1 is in fig. 5.4.

From these sets of dependency paths the selection tool computes a (usually much smaller) set of maximal discriminants. This set of maximal discriminants consists of the triples with the shortest dependency paths that encode a certain difference between parses. In example 5.5 the triples *s:su:det = det het* and *s:su = np het meisje* always co-occur, but the latter has a shorter dependency path and

| | | | | | |
|---|---|---|---|---|---|
| s:hd | = | v *zag* | s:hd | = | v *zag* |
| *s:su | = | np *jan* | *s:su | = | np *het meisje* |
| *s:obj1 | = | np *het meisje* | s:su:det | = | det *het* |
| s:obj1:det | = | det *het* | s:su:hd | = | n *meisje* |
| s:obj1:hd | = | n *meisje* | *s:obj1 | = | np *jan* |

Figure 5.5: Two readings of the sentence *Jan zag het meisje* represented as sets of dependency paths. An '*' indicates a maximal discriminant

is therefore a maximal discriminant. Other types of discriminants are lexical and constituent discriminants. Lexical discriminants represent ambiguities that result form lexical analysis, e.g. a word with an uppercase first letter can be interpreted as either a proper name or the same word without the upper case first letter. Constituent discriminants define groups of words as constituents without specifying the type of the constituent.

The maximal discriminants are presented to the annotator, who can mark them as either good (parse must include it) or bad (parse may not include it). The parse selection tool then automatically further narrows down the possibilities using four simple rules of inference. This allows users to focus on discriminants about which they have clear intuitions. Their decisions about these discriminants combined with the rules of inference can then be used to make decisions about the less obvious discriminants.

1. If a discriminant is bad, any parse which includes it is bad
2. If a discriminant is good, any parse which doesn't include it is bad
3. If a discriminant is only included in bad parses, it must be bad
4. If a discriminant is included in all the undecided parses, it must be good

The discriminants are presented to the annotator in a specific order to make the selection process more efficient. The highest ranked discriminants are always the lexical discriminants. Decisions on lexical discriminants are very easy to make and greatly reduce the set of possibilities.

After this the discriminants are ranked according to their power: the sum of the number of parses that will be excluded after the discriminant has been marked *bad* and the number of parses that will be excluded after it has been marked *good*. This way the ambiguities with the greatest impact on the number of parses are resolved first.

The parse that is selected is stored in the treebank. If the best parse is not fully correct yet, it can be edited in the Thistle (Calder 2000) tree editor and then stored again. A second annotator checks the structure, edits it again if necessary and stores it afterwards.

```
<node rel="top" cat="smain" start="0" end="6" hd="2">
  <node rel="su" pos="noun" cat="np" index="1"
        start="0" end="1" hd="1" root="Kim" word="Kim"/>
  <node rel="hd" pos="verb"
        start="1" end="2" hd="2" root="wil" word="wil"/>
  <node rel="vc" cat="inf" start="2" end="6" hd="3">
     ....
  </node>
</node>
```

Figure 5.6: XML encoding of dependency trees.

## 5.4 Querying the treebank

The results of the annotation process are stored in XML. XML is widely in use for storing and distributing language resources, and a range of standards and software tools are available which support creation, modification, and search of XML documents. Both the Alpino parser and the Thistle editor output dependency trees encoded in XML.

As the treebank grows in size, it becomes increasingly interesting to explore it interactively. Queries to the treebank may be motivated by linguistic interest (i.e. which verbs take inherently reflexive objects?) but can also be a tool for quality control (i.e. find all PPs where the head is not a preposition).

The XPath standard[2] implements a powerful query language for XML documents, which can be used to formulate queries over the treebank. XPath supports conjunction, disjunction, negation, and comparison of numeric values, and seems to have sufficient expressive power to support a range of linguistically relevant queries. Various tools support XPath and can be used to implement a query-tool. Currently, we are using a C-based tool implemented on top of the LibXML library.[3]

The XML encoding of dependency trees used by Thistle (and, for compatibility, also by the parser) is not very compact, and contains various layers of structure that are not linguistically relevant. Searching such documents for linguistically interesting patterns is difficult, as queries tend to get verbose and require intimate knowledge of the XML structure, which is mostly linguistically irrelevant. We therefore transform the original XML documents into a different XML format, which is much more compact (the average file size reduces with 90%) and which provides maximal support for linguistic queries.

As XML documents are basically trees, consisting of elements which contain other elements, dependency trees can simply be represented as XML documents, where every node in the tree is represented by an element `node`. Properties are represented by attributes. Terminal nodes (leaves) are nodes which contain no daughter elements. The XML representation of (the top of) the dependency tree given in figure 5.1 is given in figure 5.6.

The transformation of dependency trees into the format given in figure 5.6 is not only used to eliminate linguistically irrelevant structure, but also to make explicit

[2]www.w3.org/TR/xpath
[3]www.xmlsoft.org/

information which was only implicitly stored in the original XML encoding. The indices on root forms that were used to indicate their string position are removed and the corresponding information is added in the attributes `start` and `end`. Apart from the root form, the inflected form of the word as it appears in the annotated sentence is also added. Words are annotated with part of speech (`pos`) information, whereas phrases are annotated with category (`cat`) information. A drawback of this distinction is that it becomes impossible to find all NPs with a single (non-disjunctive) query, as phrasal NPs are `cat="np"` and lexical NPs are `pos="noun"`. To overcome this problem, category information is added to non-projecting (i.e. non-head) leaves in the tree as well. Finally, the attribute `hd` encodes the string position of the lexical head of every phrase. The latter information is useful for queries involving discontinuous constituents. In those cases, the start and end positions may not be very informative, and it can be more interesting to be able to locate the position of the lexical head.

We now present a number of examples which illustrate how XPath can be used to formulate various types of linguistic queries. Examples involving the use of the `hd` attribute can be found in Bouma and Kloosterman (2002).

Objects of prepositions are usually of category NP. However, other categories are not completely excluded. The query in (1) finds the objects within PPs.

(1) `//node[@cat="pp"]/node[@rel="obj1"]`

The double slash means we are looking for a matching element anywhere in the document (i.e. it is an ancestor of the top element of the document), whereas the single slash means that the element following it must be an immediate daughter of the element preceding it. The @-sign selects attributes. Thus, we are looking for nodes with dependency relation `obj1`, immediately dominated by a node with category `pp`. In the current state of the dependency treebank, 98% (5,892 of 6,062) of the matching nodes are regular NPs. The remainder is formed by relative clauses (*voor wie het werk goed kende, for who knew the work well*), PPs (*tot aan de waterkant, till on the waterfront*), adverbial pronouns (see below), and phrasal complements (*zonder dat het een cent kost, without that it a penny costs*).

The CGN annotation guidelines distinguish between three possible dependency relations for PPs: complement, modifier, or 'locative or directional complement' (a more or less obligatory dependent containing a semantically meaningful preposition which is not fixed). Assigning the correct dependency relation is difficult, both for the computational parser and for human annotators. The following query finds the head of PPs introducing locative dependents:

(2) `//node[@rel="hd" and ../@cat="pp" and ../@rel="ld"]`

Here, the double dots allow us to refer to attributes of the dominating XML element. Thus, we are looking for a node with dependency relation *hd*, which is *dominated* by a PP with a *ld* dependency relation. Here, we exploit the fact that the mother node in the dependency tree corresponds with the immediately dominating element in the XML encoding as well.

Comparing the list of matching prepositions with a general frequency list reveals that about 6% of the PPs are locative dependents. The preposition *naar* (*to, towards*) typically introduces locative dependents (50% (74 out of 151) of its usage), whereas the most frequent preposition (i.e. *van, of*) does introduce a locative in only 1% (15 out of 1496) of the cases.

In PPs containing an impersonal pronoun like *er* (*there*), the pronoun always precedes the preposition. The two are usually written as a single word (*eraan, there-on*). A further peculiarity is that pronoun and preposition need not be adjacent (*In Delft wordt* **er** *nog* **over** *vergaderd* (*In Delft, one still talks about it*)). The following query finds such discontinuous phrases:

```
(3) //node[@cat="pp" and
    ./node[@rel="obj1"]/@end < ./node[@rel="hd"]/@start ]
```

Here, the '<'-operator compares the value of the end position of the object of the PP with the start position of the head of the PP. If the first is strictly smaller than the second, the PP is discontinuous. The corpus contains 133 discontinuous PPs containing an impersonal pronoun vs. almost 322 continuous pronoun-preposition combinations, realized as a single word, and 17 cases where these are realized as two words. This shows that in almost 25% of the cases, the preposition + impersonal pronoun construction is discontinuous.

## 5.5 Evaluation Metrics

One of the applications of the Alpino treebank is the evaluation of the performance of the parser. Evaluation is done by comparing the dependency structure that the parser generates for a given corpus sentence, to the dependency structure that is stored in the treebank. For the purpose of comparison, we do not use the representation of dependency structures as trees, but the alternative notation as sets of dependency paths that we already saw in the previous section. Comparing these sets, we can count the number of relations that are identical in the parse that the system generated and the stored structure. From these counts precision, recall and F-score can be calculated. In our experience, the following metric, *concept accuracy* (CA), is a somewhat more reliable indicator of the quality of the system.[4] This metric is defined for a given sentence $s$:

$$CA(s) = 1 - \frac{D_f(s)}{max(D_g(s), D_p(s))}$$

Here, $D_p(s)$ is the set of dependency relations of the parse for sentence $s$ (generated by the parser). $D_g(s)$ is the set of dependency relations of the gold parse that is stored in the treebank for $s$. $D_f(s)$ is the number of incorrect or missing relations in $D_p(s)$.

For a corpus of sentences $S$, we often report the *per sentence* mean CA score. In addition, it is useful to consider the overall CA score:

$$CA(S) = 1 - \frac{\Sigma_{s \in S} D_f(s)}{max(\Sigma_{s \in S} D_g(s), \Sigma_{s \in S} D_p(s))}$$

[4]This metric is a variant of the metric introduced in (Boros et al., 1996); we adapted the metric to ensure that the score is between 0 and 100.

In chapter 11, a number of disambiguation models is described. Such disambiguation models aim to pick out the best parse from a set of possible parses. It is therefore natural to think of the upper and lower bounds of parse selection, on the basis of the potential scores of each of the parses in the set from which a choice has to be made. The lower bound *baseline* CA is defined as the CA of a random parse from the set of parses. The upper bound *best* CA is defined as the maximum CA of any of the parses. *best* CA is 100 just in case the correct parse is among the set of parses generated by the grammar. The *best* CA score reflects the accuracy of the grammar. Based on these bounds, we define an adjusted concept accuracy:

$$\mathrm{CA}_\kappa = 100 \times \frac{\mathrm{CA} - \textit{baseline}\ \mathrm{CA}}{\textit{best}\ \mathrm{CA} - \textit{baseline}\ \mathrm{CA}}$$

The adjusted concept accuracy $\mathrm{CA}_\kappa$ allows the models to be more broadly compared to others by incorporating not only the concept accuracy of the model but also the lower and upper bound accuracy.

## 5.6 Annotator Agreement

This section describes a small experiment that we performed in order to investigate the agreement among annotators that can be expected for treebanks of this sort. The setup of the experiment was as follows. Two annotators were asked to annotate the same set of sentences, independently from each other. These two annotators were trained to annotate according to the CGN guidelines. The set consisted of the last one hundred sentences of nineteen words of the `cdbl` corpus. Perhaps it would have been better to perform the experiment on sentences of varying length, but all shorter sentences of this corpus were already annotated. Note that the length of nineteen words is a rather typical sentence length for this corpus (mean sentence length is about twenty words).

For 42 sentences, the two annotators produced the same set of dependency relations. In table 5.1 we list the annotator agreement, expressed in terms of the *concept accuracy* metric defined in the previous section. In a few cases, the differences in annotation were due to simple mistakes. In most cases, the differences were due to a true difference in linguistic analysis. In the table we quantify what types of disagreement occurred often. The most frequent cause of disagreement were differences in attachment of modifiers. Another frequent cause of disagreement is related to the choice of the dependency relation: typical disagreements involve the labels MOD, LD, PC for modifiers, directional complements, and prepositional complements, respectively. In some cases, the disagreement is about whether or not a fixed phrase should be treated as a multi-word unit, or not. Disagreements of this sort are counted rather heavily in the concept accuracy measure, because each relation that involve the multi-word unit will be treated as incorrect. The final somewhat larger class of mistakes were related to the identification of discourse units (for instance for the analysis of certain spoken language constructs that sometimes occur in written texts too).

| | |
|---|---|
| agreement | 93.1 % |
| mistakes | 1.5 % |
| agreement after correction | 94.6 % |
| attachment of modifiers | 1.5 % |
| different dependency labels | 1.3 % |
| multi word units | 1.1 % |
| discourse units | 0.7 % |
| misc | 0.7 % |

Table 5.1: Results of Annotator Agreement Experiment

## 5.7 Conclusions

A treebank is very important for both evaluation and training of a grammar. For the Alpino parser, no suitable treebank existed. For that reason we have started to develop the Alpino Dependency Treebank by annotating a part of the Eindhoven corpus with dependency structures. As the treebank grows in size, it becomes more and more attractive to use it for linguistic exploration as well, and we have developed an XML format which supports a range of linguistic queries.

To facilitate the time consuming annotation process, we have developed several tools: interactive lexical analysis and constituent marking reduce the set of parses that is generated by the Alpino parser, the tool for addition of lexical information makes parsing of unknown words more efficient and the parse selection tool facilitates the selection of the best parse from a set of parses. In the future, constituent marking could be made more user friendly. We could also look into ways of further reducing the set of maximal discriminants that is generated by the parse selection tool.

The treebank currently contains over 6,000 sentences, and is available on `http://www.let.rug.nl/~vannoord/trees`. Much effort will be put in extending the treebank to at least the complete cdbl newspaper part of the Eindhoven corpus, which contains more than 7,100 sentences.