Parsing Corpus-Induced Type-Logical Grammars

Richard Moot

LaBRI
Domaine Universitaire
351, Cours de la Libération
33405 Talence, France
Richard.Moot@labri.fr

Abstract. Type-logical grammars which have been automatically extracted from linguistic corpora provide parsers for these grammars with a considerable challenge. The size of the lexicon and the combinatory possibilities of the lexical entries both call for rethinking of the traditional type-logical parsing strategies. We show how methods from statistical natural language processing can be incorporated into a type-logical parser, give some preliminary data and sketch some new experiments we expect to produce better results.

1 Introduction

Parsers for type-logical grammars, such as the one presented in [1] have been used in teaching and research environments to experiment with carefully designed grammar fragments.

When we look at the algorithm of [2], which produces a type-logical lexicon from the syntactic annotation of the Spoken Dutch Corpus, we see that the size of the final lexicon, even after some reductions we propose, is still prohibitive to parsing even rather small sentences.

We propose to incorporate ideas from the statistical parsing community into a type-logical parser and present some preliminary results.

2 Type-Logical Grammar

Type-logical grammar is an elegant formalism, giving precise logical descriptions of linguistic phenomena [3, 4].

In this section we will first introduce the formalism and then present a basic parsing algorithm for type-logical grammars.

2.1 Formalism

We will present type-logical grammar in the style of [5]. We'll start by presenting it as a simple grammar which simple allows us to combine lexical trees and gradually add what we need to obtain the full type-logical system we're interested in, referring the reader interested in a more detailed account to any of the above-mentioned articles.

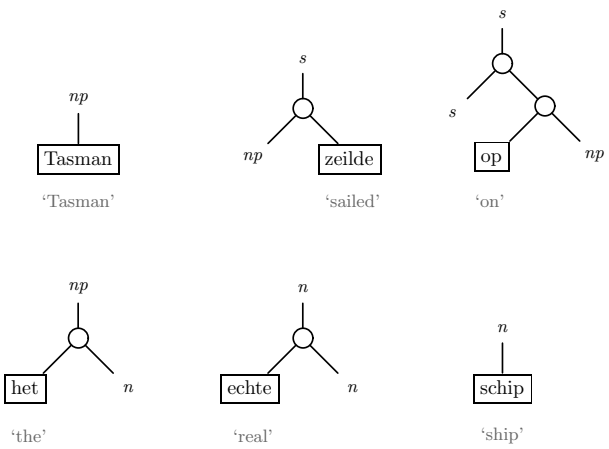

Fig. 1. Sample lexicon

Tree Grammars In it's simplest incarnation we just have a lexicon of trees, where a) every tree is labeled at its root with an atomic formula, b) exactly one leaf of every tree is labeled with an word, and c) every other leaf is labeled with an atomic formula.

An example of such a lexicon for a Dutch grammar is given in Figure 1. For the benefit of the reader, we've given an English translation below every word, though these are not formally a part of the grammar.

A word like 'Tasman' has the most basic lexical entry: it is just an *np* typed tree. The lexical entry for 'sailed' is slightly more complicated it is an *s* typed tree, but it still needs to find an *np* typed tree with 'Tasman' as its lexical leaf. We can combine these two trees to form a tree of 'Tasman zeiled' which is a tree of type *s*, as shown on the left side of Figure 2 on the next page. We can continue combining this tree with the other lexical entries to produce the tree for 'Tasman zeilde op het echte schip' which is shown in the same figure on the right.

Contractions In order to obtain the symmetries required to call our formal system a *logic* we need to be able to do more than construct trees, we also need some inverse operators which destroy or 'cancel out' the normal tree constructors.

Figure 3 on the facing page gives two example lexical entries for the Dutch word 'wie' utilizing these inverse operators, which are drawn with a black circle. Intuitively the left entry says it produces a *wh* question if it combines with a sentence, but in addition it is allowed to fulfill the role of an *np* inside this sentence.

The actual cancellation of the destructors takes place by means of the contractions given in Figure 4 on the next page. In every case a constructor is

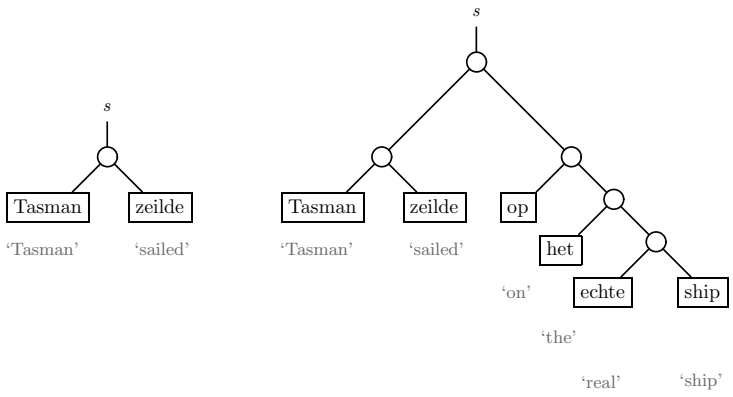

Fig. 2. *s* type tree for 'Tasman zeilde op het echte schip'

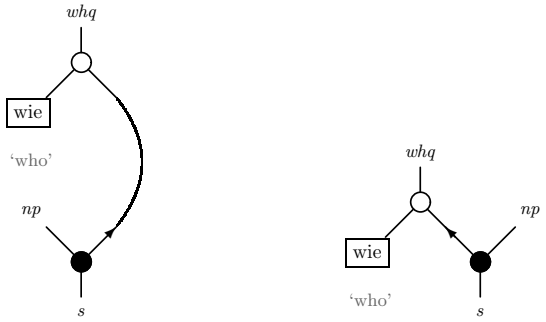

Fig. 3. Two lexical graphs for 'wie'

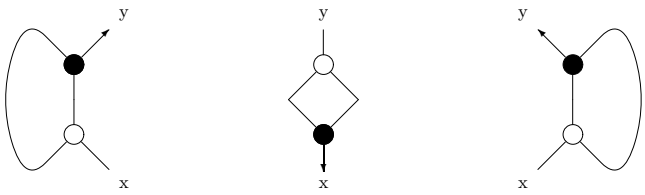

Fig. 4. Contractions

connected to a destructor at the two points not marked by the arrow and the constructor and destructor cancel out each other. In the result of the cancellation the previously distinct vertices x and y will be connected.

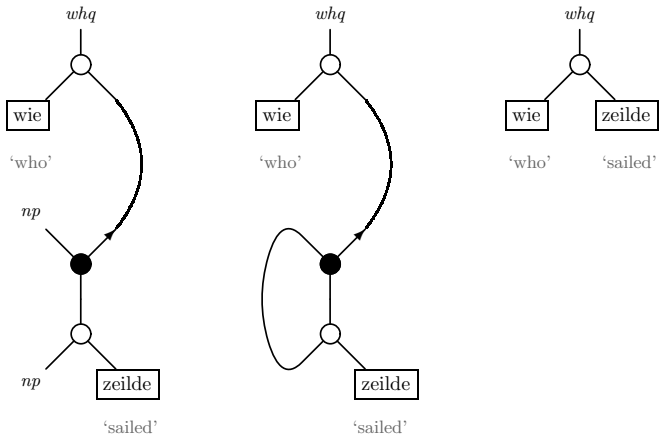

Fig. 5. An example derivation of 'wie zeilde'

As an example, if we combine the lexical entry for 'wie' with the lexical entry for 'zeilde', we can combine them into a valid *wh* question as shown in Figure 5. On the left we see the result after connecting the s formulas, in the middle the np formulas have also been connected, producing the configuration which allows us to contract the linked constructor and destructor. This final tree after the contraction has been performed is shown on the right.

Structural Conversions In addition to the contractions of Figure 4, a grammar my specify any number of structural conversions; essentially tree rewrites like, for example, associativity. We refer the reader to [5] for details.

2.2 Automated Deduction

Automated deduction for type-logical grammar can be divided into three distinct stages:

1. Find a lexical tree for every word in the sentence.
2. Attach positive to negative formulas.
3. Perform all contractions. The result must be a tree with the input sentence as its yield.

In a realistic implementation, it makes sense to interleave these three stages, for example by already performing some of the contractions before all of the formulas have been attached (see [6] for discussion), but for the sake of simplicity we will treat the three stages separately.

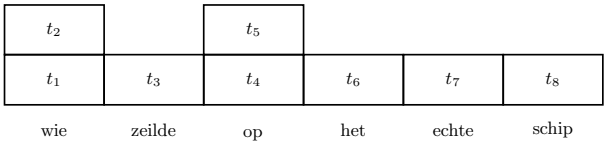

Fig. 6. Lexical lookup

Lexical Lookup In the lexical lookup stage, we select one lexical tree for each word in the input sentence, a situation which is schematically pictured in Figure 6, where 'wie' and 'op' each have two possible lexical trees. For every column of the table in the figure we need to select one tree.

For human-generated lexicons, the lexical ambiguity is typically small; it is considered desirable to reduce cases of lexical ambiguity to derivational ambiguity, ie. instead of having two lexical trees t_1 and t_2 for a single word we use a single lexical tree t_3 such that t_3 subsumes both t_1 and t_2. For example, it might be argued that the two trees in Figure 3 should be replaced by a single one.

When we use a lexicon which is automatically extracted from a corpus, as we will see in Section 3.2, and where, for longer sentences, the mean number of trees assigned to a single word is over 40. Enumerating the possibilities is not a realistic possibility in this case and we need to find a better solution.

Connecting The second stage consists of connecting positive formulas, those formulas which are a root node of the tree, to negative formulas, those formulas which are a leaf of the tree. Figure 7 on the following page pictures the situation schematically.

Aside from the two *whq* formulas, which can only be linked in a single way, we have given every formula occurrence a unique number as its subscript. The objective of the connecting stage is putting exactly one mark in every row and every column of the tables on the right of the figure in such a way that the connections they imply produce a graph which can be contracted. In this case, the gray squares mark the correct solution (at least assuming we have associativity).

In the general case, finding a solution here is known to be NP complete [7] and for parsing sentences from a corpus, which can be quite long, this is a another big problem.

Contraction The contractions can typically be checked efficiently, in the case with only binary constructors we have considered so far we can even apply a greedy contraction strategy, contracting every redex we encounter. When structural conversions are added, however, the situation can become more complex [6].

In the paper, we won't have many things to say about this stage, but we will refer the reader to [2] for discussion about a useful set of structural conversions for our current purposes.

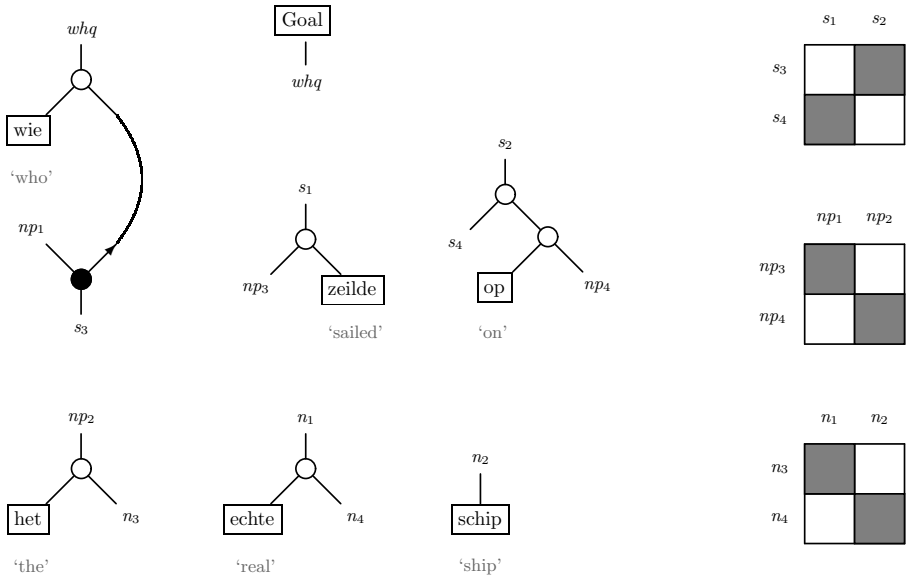

Fig. 7. Selecting a possible connection

3 Generating a Treebank from the CGN Syntactic Annotation

In this section we will talk about generating a type-logical treebank from the syntactic annotation files of the Spoken Dutch Corpus and some methods of reducing the size of the generated lexicon.

3.1 Syntactic Annotation

The Spoken Dutch Corpus ('Corpus Gesproken Nederlands', or CGN) is an ambitious project which — for its final release — will contain 10 million words of contemporary spoken Dutch with various forms of linguistic annotation. We will focus on the 1 million words of the corpus which will receive syntactic annotation, of which the currently available release 6 contains more than half.

For the CGN syntactic annotation, the annotation tools developed for the German NEGRA Corpus [8] have been used to produce syntactic annotation graphs of the form shown in Figure 8 on the next page.

I will briefly note some properties of the annotation. More details on the annotation format and philosophy can be found in [9]. The annotation graphs are directed, acyclic graphs, where every vertex is labeled with a part-of-speech tag (like WW1 for a singular, inflected verb) or a tag for a grammatical constituent (like SV1 for a verb-initial sentence) and every edge is labeled with a dependency relation (like *hd* for head and *obj1* for a direct object).

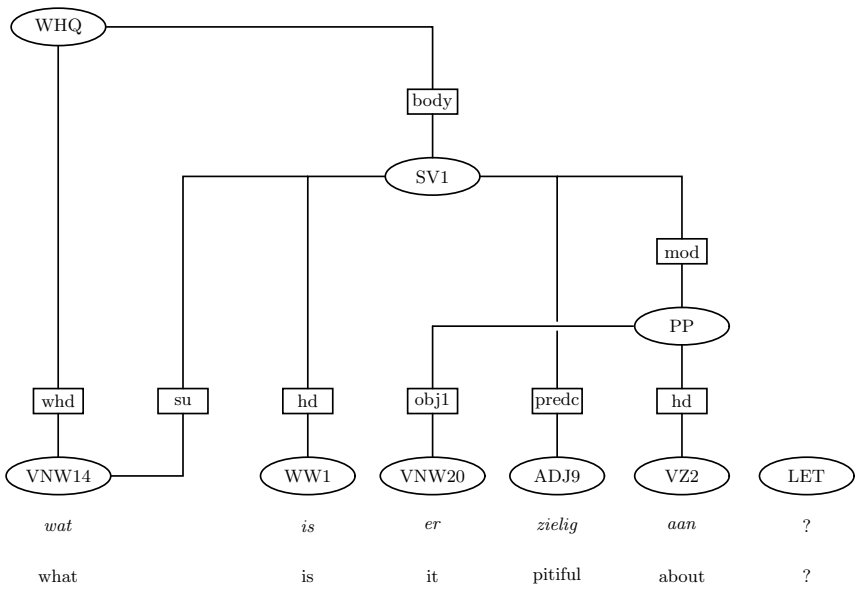

Fig. 8. 'What is pitiful about it?', sentence 99 of CGN section fn000177

Some other properties are:

- we can have multiple dependencies; for example VNW14 (wat) is both the head of the *wh* phrase and the subject of the verb-initial sentence.
- we can have discontinuous constituents; for example 'er aan' is a PP in the example sentence, even though the ADJ9 (zielig) is positioned between these two words.
- the graphs are allowed to be disconnected; for example the LET constituent is an isolated vertex.

3.2 Treebank Generation

In [2] we see how a type-logical treebank can be extracted from the CGN syntactic annotation. A parametric algorithm is given, requiring three functions as its input:

1. a mapping from vertex labels to formulas,
2. a function identifying a head for every grammatical constituent,
3. a function identifying the modifiers of every grammatical constituent.

Since our algorithm requires every domain to have a head and we are dealing with a spoken corpus, with often grammatically incomplete utterances, we assign every domain a head by an order of preference. For example, a verb-initial sentence SV1 has as its head the constituent with edge label *hd*, typically the main verb, but if there is no such constituent then either a verbal complement

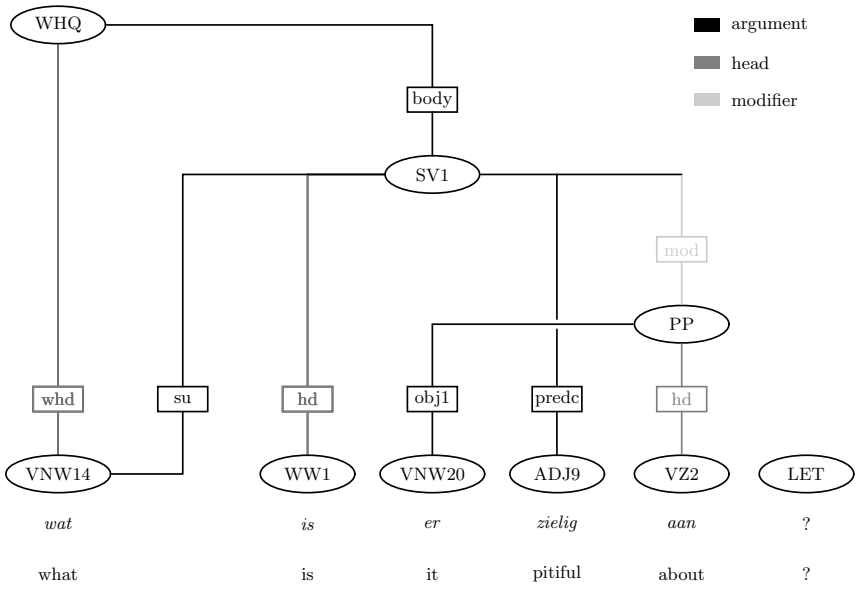

Fig. 9. Heads and modifiers

vc or a predicative complement *predc* or even the subject *su* can function as the head, even though these would normally be considered arguments.

Figure 9 shows the previous annotation graph but this time with the head and modifier information added for every constituent.

The algorithm of [2] splits every vertex which is not a head or a modifier of the current domain. This will produce the formula corresponding to the vertex both as a leaf of the current domain and as the root of the daughter domain.

Modifiers are treated slightly differently, they will be cut in such a way their root is the same as the formula assigned to the root of the current domain and they have an additional leaf which is assigned this same formula. In other words a modifier will modify the result formula.

Figure 10 on the facing page gives the result of cutting the example graph into lexicalized components. Note that (modulo an extra vertex for the modified *sv1*) reconnecting all formulas produces a graph isomorphic to the original one.

Producing a type-logical lexicon from Figure 10 is just a simple matter of replacing the graph connections there with the type-logical connectors of Section 2. The result is shown in Figure 11.

3.3 A More Compact Lexicon

When we attack the 59.910 sentences and the 514.167 words of CGN release 6 with the algorithm of the previous section, the resulting lexicon is rather enormous, containing 4.767 distinct lexical trees.

By inspecting the generated lexicon, however, we will notice some room for improvement. In Figure 11, for example, we assigned the word 'zielig' the atomic

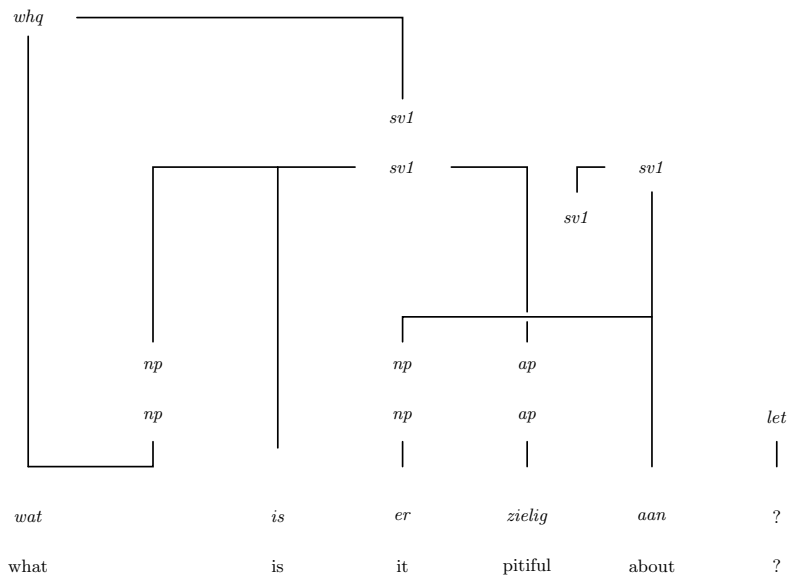

Fig. 10. Sentence 99 of CGN section fn000177 split into lexical components

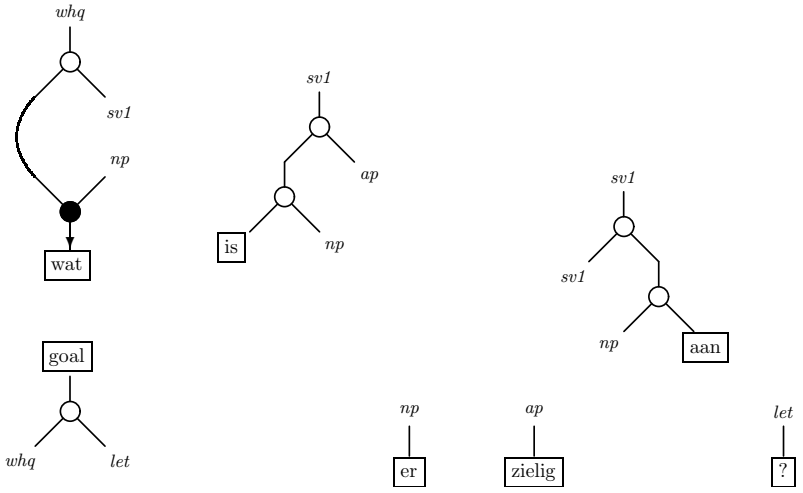

Fig. 11. Lexical entries

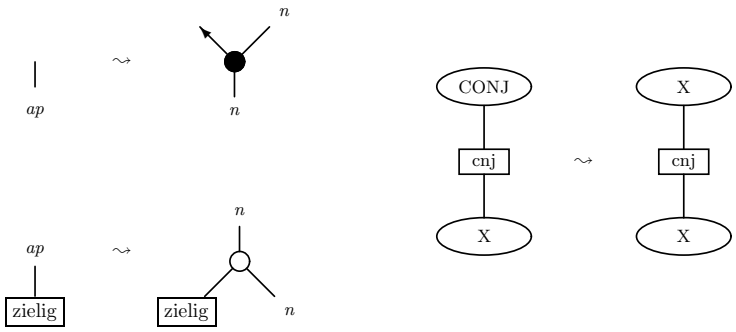

Fig. 12. Reducing lexical ambiguity

category *ap*. A more typical lexical entry for this word is presented in Figure 12, which assigns it the type of an *n* modifier, an entry which was already a possibility for many *ap* words in the corpus. Note that combining the complex entries requires only one extra contraction.

Another CGN grammatical category is CONJ for 'conjunction'. By replacing the CONJ label by the label of one of the conjoints (which can, at least in principle, be any other label in the grammar) we remove the need for any lexical entry to have an explicit entry where it selects for a *conj* type.

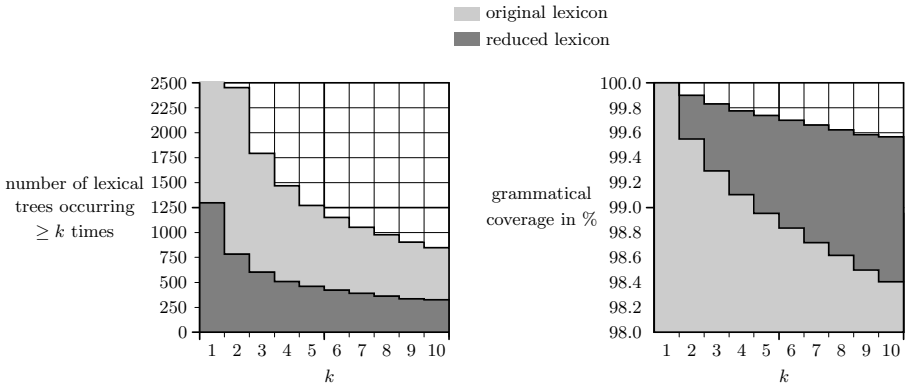

Table 1. Lexicon size for different cutoff values k

After applying these and other simplifications, one at a time, the final lexicon was reduced from 4.767 distinct lexical trees to 1.298 distinct lexical trees. Still a rather large lexicon, but we have to take into account that the corpus contains many sentences a normative account of Dutch would find ungrammatical. Therefore, it is useful to see how large the lexicon is when we remove some of this

noise. A useful measure here is to apply a cutoff, ie. to remove those trees which appear less than k times in the lexicon. The intuition here is that accidents of the data will occur only a few times in total. Table 1 lists the size of the lexicon for different cutoff values together with the percentage of the data the remaining lexical trees account for, both for the original and the reduced lexicon.

To show the potential trouble for the lexical lookup phase, we have also plotted the mean number of lookups per word against different sentence lengths in Table 2. In the sense of Figure 6 this will correspond to the average number of trees in every column. We see that for the unreduced lexicon we will have to consider over 90 trees per word for even medium length sentences. The reduced lexicon fares slightly better, but even here we have to consider over 40 trees per word rather quickly.

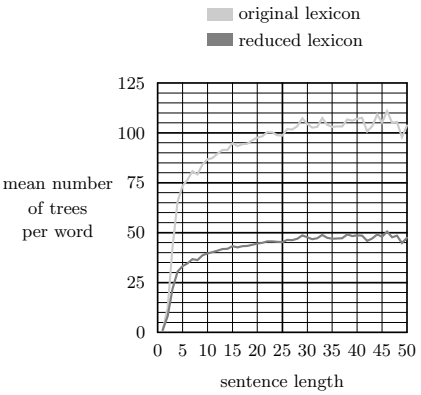

Table 2. Mean number of lookups per word for sentences of different lengths

4 Statistical Parsing

In the previous sections we given an overview the problem: type-logical grammars which have been automatically extracted from linguistic corpora are too large to allow brute-force parsing. Statistical methods, which have been applied successfully in several fields of natural language processing [10], may be able to suggest some solutions, since the problems we encountered in the previous sections have their counterparts in statistical natural language processing.

The price for this, of course, is that we will have a quantifiable amount of error. However, we can raise the bar as high as we like, trading off precision (or, for the logician, completeness) for speed, and in case we want a complete theorem prover or a 100% accurate parser for a given type-logical grammar, the material in the following sections is still useful in the sense that it provides a best-first search algorithm for finding proofs.

4.1 Supertagging

We have seen in that one of the main obstacles for parsing even moderately long CGN sentences using the extracted type-logical lexicon is the number of lexical possibilities.

A similar problem was encountered when a large-scale Tree Adjoining Grammar was developed, in which context the idea of *supertagging* was first introduced [11]. The basic idea is that strategies from part-of-speech tagging, where given a sequence of words we want to assign each of these words a part-of-speech tag, can be used to assign more complex structures as well, ie. given a sequence of word-POS pairs we want to assign a lexical tree to each of these pairs. Modern supertagging implementations get around 92% of lexical assignments correct [12, 13].

To get an indication of the potential usefulness of supertagging for parsing corpus-induced type-logical grammars, we performed two pilot experiments: the first one using unigram models, in a sense the simplest possible way of supertagging, the second a maximum entropy tagger. For our experiments we used every fifth CGN sentence (for a total of 11.981 sentence) as test data and all other sentences (47.928 in total) to train our models. All experiments used the reduced lexicon without cutoffs.

Unigram Models A unigram simply assigns each word in the grammar the tree it is most often associated with in the training data. For words which don't occur enough times in the training to make a reliable estimate (less than 5 times for our experiment) the tree most often associated to its POS tag was used. In the exceedingly rare case that the POS tag occurred less than 5 times in the training data, the most frequent lexical tree was used: a simple *np* lexical tree.

As can be seen from the data in dark gray on Table 3, the total result from this method is 63.44% correct lexical assignments and the results are especially bad for the VZ (preposition) category and the WW (verb) category.

Maximum Entropy Modeling It is clear we need to have at least a bit more contextual information improve the results for verbs and prepositions. A successful method for dealing with many different types of contextual information and deciding on the basis of the training data which are the most relevant to obtaining the correct solution is maximum entropy modeling. This method has also been applied for supertagging in [13].

For our second experiment, we used Adwait Ratnaparki's [14] POS tagger and trained it on producing supertags instead. Training it one the same data as the first experiment produced the results in light gray on Table 3: a total of 77.37% of correctly assigned supertags and a notable improvement in the correct assignments to VZ and WW categories.

Note that these results are still not quite state-of-the-art performance for supertagging but we expect better results when using a dedicated supertagger instead of a POS tagger.

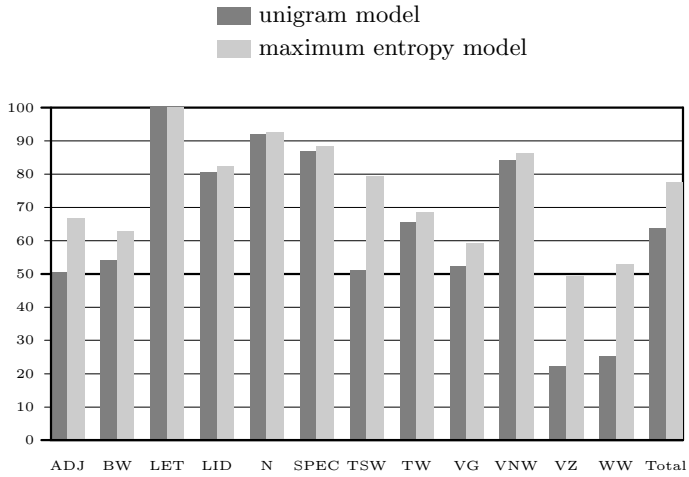

Table 3. Combined Unigram and Maximum Entropy Model Results

Features An important part of the success of a maximum entropy model are the types of features that are available to it. In this context, a feature is just a piece of contextual information, typically something like 'the previous POS tag is DET (a determiner)'.

Useful features in a logical context could take into account the subformulas of surrounding words or the ratio of positive versus negative occurrences of atomic formulas. This will increase the number of features rather dramatically but methods exists of identifying the 'best' features automatically [15].

4.2 Connections

I will briefly touch upon the second complexity problem. As noted before, even after we have found the correct lexical entries, finding the correct connections is still an NP complete problem. But if we look back at Figure 7 on page 6 we see that the problem there was to make exactly one selection for every row and exactly one selection for every column in the tables.

Graph theorists will recognize this problem as the perfect bipartite matching problem. A result from graph theory is that if we have a *weighted* graph, ie. if we put a number in every field of the table, we can generate the k best perfect matchings in $O(kn^3)$ time (where n is the number of vertices in the graph).

This means that if we put some sensible weights in the tables and if we select an appropriate value of k then we have a polynomial approximation of the original problem. The simplest measure to use here is just to note the distance between the words the atomic formulas are part of. This will produce the k best solutions under the processing theory proposed in [16] and [17].

A corpus-based solution is to look, for every atomic positive atomic formula, to which negative atomic formula it is connected in the training data and record the distance (the distance in this case being the number of intervening

negative formulas of the same type) and then using these distances to give all negative formulas up to a certain distance a weight, being the number of times a connection of this distance was encountered in the training data.

More sophisticated training schemes are possible, of course, and experiments need to be done to find out how well this strategy performs.

5 Conclusions

We have seen how the Spoken Dutch Corpus can be used for type-logical grammar extraction. The size of these grammars, however, is rather prohibitive for practical parsing. We have given some preliminary data suggesting that methods from statistical natural language processing can be useful in overcoming these problems and sketched some new experiments from which we expect better results.

References

1. Moot, R.: Grail: an automated proof assistant for categorial grammar logics. In Backhouse, R., ed.: Proceedings of Calculemus/User Interfaces for Theorem Provers. (1998) 120–129
2. Moortgat, M., Moot, R.: Using the Spoken Dutch Corpus for type-logical grammar induction. In: Proceedings of the Third International Language Resources and Evaluation Conference, Las Palmas (2002)
3. Moortgat, M.: Categorial type logics. In van Benthem, J., ter Meulen, A., eds.: Handbook of Logic and Language. Elsevier/MIT Press (1997)
4. Morrill, G.: Type Logical Grammar. Kluwer Academic Publishers, Dordrecht (1994)
5. Moot, R., Puite, Q.: Proof nets for the multimodal Lambek calculus. Studia Logica **71** (2002) 415–442
6. Moot, R.: Proof Nets for Linguistic Analysis. PhD thesis, Utrecht Institute of Linguistics OTS, Utrecht University (2002)
7. Kanovich, M.: The multiplicative fragment of linear logic is NP-complete. Technical report, University of Amsterdam (1991) ITLI Prepublication Series X-91-13.
8. Brants, T.: Tagging and Parsing with Cascaded Markov Models - Automation of Corpus Annotation. PhD thesis, German Research Center for Artificial Intelligence and Saarland University, Saarbrücken, Germany (1999)
9. Hoekstra, H., Moortgat, M., Renmans, B., Schuurman, I., van der Wouden, T.: Syntactic analysis in the Spoken Dutch Corpus (CGN). In: Proceedings of the Third International Language Resources and Evaluation Conference, Las Palmas (2002)
10. Manning, C.D., Schutze, H.: Foundations of Statistical Natural Language Processing. MIT Press (1999)
11. Joshi, A., Srinivas, B.: Disambiguation of super parts of speech (or supertags): Almost parsing. In: Proceedings of the 17th International Conference on Computational Linguistics, Kyoto (1994)
12. Srinivas, B.: Performance evaluation of supertagging for partial parsing. In: Proceedings of Fifth International Workshop on Parsing Technology, Boston (1997)

13. Clark, S.: Supertagging for combinatory categorial grammar. In: Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Formalisms, Venice (2002) 19–24
14. Ratnaparki, A.: A maximum entropy part-of-speech tagger. In: Proceedings of the Empirical Methods in Natural Language Processing Conference, Pennsylvania (1996)
15. Pietra, S.D., Pietra, V.D., Lafferty, J.: Inducing features of random fields. IEEE Transactions on Pattern Analysis and Machine Intelligence **19** (1997) 380–393
16. Morrill, G.: Incremental processing and acceptability. Technical Report LSI–98–46–R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya (1998)
17. Johnson, M.: Proof nets and the complexity of processing center-embedded constructions. Journal of Logic, Language and Information **7** (1998) 443–447