

LEXICAL AMBIGUITY IN TREE ADJOINING GRAMMARS

Pradip DEY and Barrett R. BRYANT

Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294, USA

Tadao TAKAOKA

Department of Information Science, Ibaraki University, Hitachi, Ibaraki 316, Japan

Communicated by W.M. Turski

Received 16 June 1989

Revised 21 November 1989

There are at least two ways of handling lexical ambiguity in a tree adjoining grammar. One of them seems to be computationally intractable. The other is computationally efficient. This paper describes these two methods with algorithms and their analyses.

Keywords: Linguistics, syntactic analysis, parsing, interpreters, language processors

1. Introduction

Lexical ambiguity is a major problem in natural language parsing [2,11]. Parsing is always based on a finite description of the language called grammar. Tree adjoining grammars (TAG's) developed at the University of Pennsylvania seem to have the essential features necessary for natural language parsing [6,9]. TAG's define a finite set of elementary trees and an adjunction operation that produces composite trees through the combination of simple (elementary) ones. The trees are taken to be structural descriptions of sentences in the language. The central module of the parser is a pattern matcher that finds every tree that matches the input string. If there is only one tree that matches the input, then the input is structurally unambiguous. If the input matches more than one tree, it is structurally ambiguous. Structural ambiguity often results from lexical ambiguity, that is, when a word belongs to more than one categories (e.g. the work *flies* is a noun and a verb) [2]. Two algorithms for processing lexical ambiguity in TAG's are described with their analyses.

2. The TAG parser

The major components of the TAG parser are shown in Fig. 1.

(1) Fig. 1.

The tree bank has two types of trees: initial trees and auxiliary trees. Auxiliary trees are used in a composition operation called adjoining to account for recursion; they do not occur independently in the language. We describe auxiliary trees with adjoining operation at the end of this section. Before that we present a simplified version of TAG's for expository purposes. You may assume

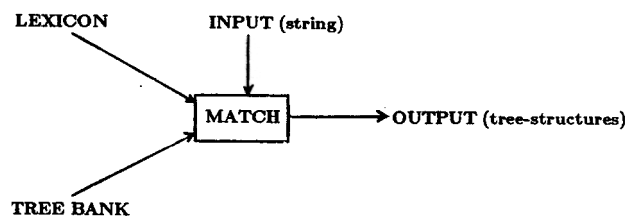


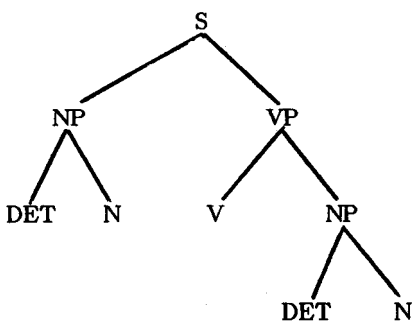
Fig. 1.

for the time being that the tree bank has only initial trees such as those in (3) and (4) whose tip-nodes (frontier nodes) are preterminal symbols. The categories of the lexicon are also pre-terminal symbols. The lexicon looks like (2):

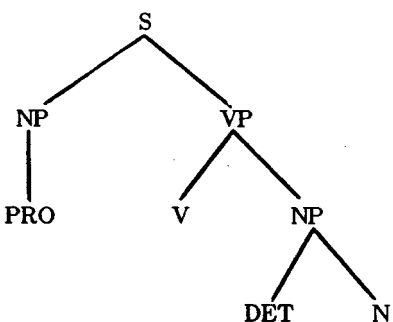
- (2) LEXICON
- (a DET)
- (an DET)
- (arrow N)
- (flies (V N))
- (like (P V A))
- (machine N)
- (the DET)
- (time (N A V))

The tree bank for English includes the two initial trees given in (3) and (4). (N = Noun, V = Verb, DET = Determiner, A = Adjective, P = Preposition, PRO = Pronoun, NP = Noun phrase, VP = Verb phrase, S = Sentence.)

(3) Tree1 =

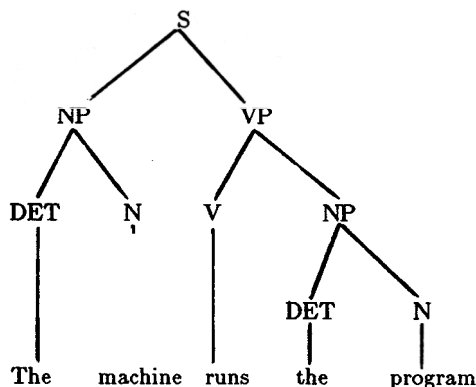


(4) Tree2 =



These two trees match a large number of sentences in English. For example, Tree1 matches sentences like (5) as shown below.

(5) The machine runs the program.
Tree1 =



In the match phase the tip-nodes of the trees are matched against the categories of words. Suppose that the input is the sentence given in (5). There is a procedure in MATCH called "CATEGORIES" that returns the lexical categories of the input string as shown in the list form below.

(CATEGORIES

'(The machine runs the program))
= (DET N V DET N)

When a lexical item belongs to two or more lexical categories it gives rise to lexical ambiguity which is a source of inefficiency in traditional parsers. We address the problem of lexical ambiguity in considerable detail in Section 3. There is a procedure in MATCH called "TIPNODES" that returns the tip-nodes (frontier nodes) of a tree. Thus,

(TIPNODES Tree1) = (DET N V DET N)

MATCH simply asks if the categories of the input string are equal to the tip-nodes of a tree. This type of parsing is different from traditional parsing with grammar rules [1,5,11] and it has at least two advantages: (i) the trees are not built at run time, so it is fast, (ii) the parser can be developed incrementally by adding new trees to the tree-bank.

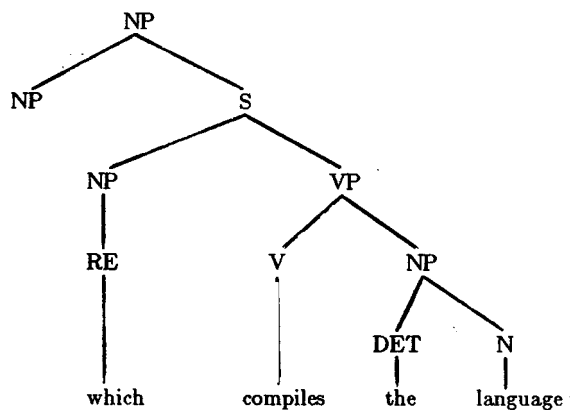
2.1. Factoring recursion by adjoining

The parser outlined above works well if an additional problem is solved: that of embedded clauses. An unbounded number of embeddings can occur in English. An example of embedding is given in (6) where "which compiles the language" is an embedded clause.

(6) The machine which compiles the language runs the program.

This phenomenon is handled in TAG's by a composition operation called adjoining. Adjoining is comparable to transformations of early transformational grammars [3]. We mentioned earlier that the tree bank has two kinds of trees: initial trees and auxiliary trees. Examples of initial trees are given in (3) and (4) above. An example of auxiliary tree is given in (7). Lexical items are inserted into this tree for expository purposes; usually trees are stored without lexical items. usually trees are stored without lexical items.

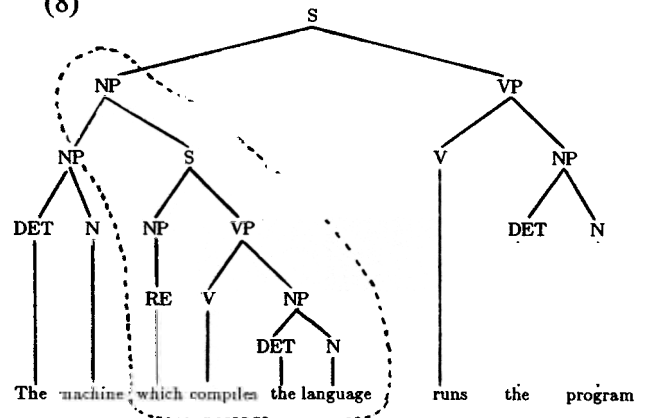
(7) Tree3 =



Unlike initial trees, auxiliary trees have a tip-node which is identical to its root. This tip-node is called a *hook* and plays an important role in adjoining. An auxiliary tree can be adjoined at any node of a nonauxiliary tree (initial or composite) if that node is labeled by the same symbol as the root of the auxiliary. An example of adjoining is shown in (8) in which Tree3 is adjoined at an NP node in Tree1. The auxiliary tree is marked by a dotted line in the resulting composite tree

and lexical items are inserted for the purpose of demonstration.

(8)



If Tree3 is adjoined at the NP dominated by VP of Tree1 then we get sentences like (9). If Tree3 is adjoined at both the NP's of Tree1 then we get sentences like (10).

(9) The machine runs the program which compiles the language.

(10) The machine which compiles the language runs the program which computes an answer.

It is to be noted that the adjoining can be applied before run-time (e.g. during compile-time) to generate a large number of trees. Since people ordinarily use short sentences, all the trees with up to a fixed number (say 25) of tip-nodes can be pre-generated. This reduces run-time computation of parsing to pattern match. Only on rare occasions will the purser use run-time adjoining.

3. Lexical ambiguity and match

Lexical ambiguity is often the source of inefficiency in a parser. In this section we describe two methods of processing lexical ambiguity in TAG's: (1) Distribution Method, and (2) Membership Method. We present two algorithms and analyze their time complexity. We show that the latter is more efficient than the former. The parameters used in the expressions of the time complexity are:

- *n*: the length of the input string,
- *l*: the number of entries in the lexicon,

- x : the number of nonauxiliary trees in the tree-bank,
- k : the number of categories in the lexicon.

3.1. Distribution Method

We explain the Distribution Method with an example from [11,92] which is reproduced in (11).

(11) Time flies like an arrow.

If a word belongs to k lexical categories, then we say that the word is k -way ambiguous. According to the lexicon given in (2) the word "Time" is three-way ambiguous. The word "flies" has two-way ambiguity and the word "like" has three-way ambiguity. For the sentence in (11) the procedure CATEGORIES returns the following sequence of categories:

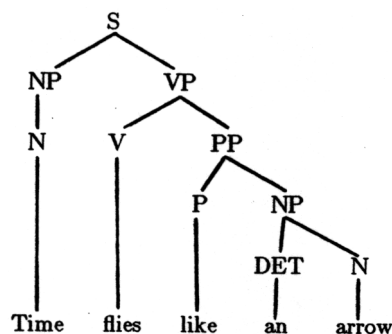
(12) ((N A V) (V N) (P V A) DET N)

In order to match against the tip-nodes of the trees the categories of ambiguous words are distributed creating 18 sequences:

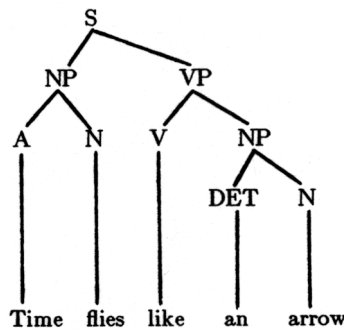
((N V P DET N) (A N V DET N) (V N P DET N) (N V V DET N) (N N P DET N) (N N V DET N) (A V P DET N) (A V V DET N) (A N P DET N) (V V V DET N) (V V P DET N) (V N V DET N) (N V A DET N) (N N A DET N) (A V A DET N) (A N A DET N) (V V A DET N) (V N A DET N))

Two trees that match the first two sequences of categories are given in (13) and (14).

(13)



(14)



Processing lexical ambiguity after distribution of categories is extremely costly. If each word is k -way lexically ambiguous, then for each tree the algorithm has to do at most nk^n comparisons. The worst-case complexity of such a matching for a k -way ambiguous language can be expressed as: $O(xnk^n)$ where x is the number of trees considered.

3.2. Membership Method

We develop a MATCH process that is quadratic. It does not distribute the categories. It takes a sequence like (12) and compares it with the tip-nodes of a tree by membership. For example, the first tip-node of the candidate tree must be a member of the first subsequence of (12). The procedure is given below:

Procedure MATCH(TREES, W)

```
// Input: TREES = {TREE1, TREE2, ..., TREEx},
//          a finite sequence of trees //
//          W = {W1, W2, ..., Wn},
//          (input string (a sequence of
//          words) //
// Output: MTREES = the set of all trees that
//          match W //
1 CAT ← CATEGORIES(W)
2 MTREES ← ( )
3 for i = 1 to x do:
4   TIPS ← TIPNODES(TREEi)
5   if |TIPS| = |CAT| then do:
6     if, for j = 1 to |TIPS|,
```

```

MEMBER(TIPSj, CATj),
then M TREES ← M TREES ∪ TREEi
end if
end for
7 return M TREES

```

This MATCH procedure is efficient. It needs at most $O(xkn)$ comparisons. Certain types of lexical ambiguity can be resolved by contexts as suggested in [8,10,11] which will further improve the efficiency of the system. This latter improvement will not, however, be reflected in the worst-case analysis. Considerable speed-up can be achieved by doing parallel matching [4]. We did not include adjoining in the above algorithms because we assumed that run-time adjoining can be avoided.

A comparison of the membership method with some popular methods of dealing with lexical ambiguity follows. Earley's algorithm can parse any context-free language with lexical ambiguity in n^3 time [5]. Earley's method is not related to the methods described in this paper. Earley's method uses a set of states obtained from the production rules of a context-free grammar. Shieber presents an algorithm for parsing ID/LP (immediate dominance and linear precedence) grammars and claims that its time complexity is bounded by n^3 [7]. ID/LP grammars are context-free and Shieber's algorithm is based on Earley's [7]. However, Shieber did not consider lexical ambiguity in his algorithm. Parsing ID/LP grammars is shown to be NP-complete when lexical ambiguity is present [2]. It is shown that Shieber's algorithm suffers from combinatorial explosion in the state set due to lexical ambiguity [2]. The net effect is similar to that of the distribute method described above. The time complexity of the ATN (augmented transition network) based parsers is exponential [12]. The standard ATN deals with lexical ambiguity by chronological backtracking which is computationally inefficient. However, the ATN formalism can parse unrestricted languages because it has the power of a Turing machine. TAG's are mildly context-sensitive because they generate all of the context-free languages and a small subset of the context-sensitive languages [6,9]. Although lexical ambiguity has not been explicitly discussed in any earlier paper on TAG, it is provable that a parsing

method described by Joshi and Vijay-Shankar [9] is capable of parsing lexical ambiguity in TAG's in n^6 time. The method uses a dynamic programming technique based on the Cocke-Younger-Kasami algorithm [1]. The membership method presented in this paper is a viable alternative to the method of [9].

Acknowledgment

This research was supported in part by grant number 62045007 under the Monbuscho International Scientific Research Program from the Ministry of Education of Japan. The authors are grateful to some IPL referees whose comments were useful in the process of revision of this paper.

References

- [1] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and Compiling: Volume 1* (Prentice-Hall, Englewood Cliffs, NJ, 1972).
- [2] G.E. Barton Jr, On the complexity of ID/LP parsing, *Comput. Linguist.* **11** (1985) 205-218.
- [3] N. Chomsky, *Syntactic Structures* (Mouton, The Hague, 1957).
- [4] P. Dey, S.S. Iyengar and J.S. Byoun, Parallel processing of tree adjoining grammars, Tech. Rept., Dept. of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham (1988).
- [5] J. Earley, An efficient context-free parsing algorithm, *Comm. ACM* **13** (1970) 94-102.
- [6] A.K. Joshi, Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?, in: J.H. Dowty et al., eds., *Natural Language Parsing* (Cambridge University Press, Cambridge, MA, 1985) 206-250.
- [7] S.M. Shieber, Direct parsing of ID/LP grammars, *Linguist. and Philos.* **7** (1984) 135-154.
- [8] S.L. Small, G.W. Cottrell and M.K. Tanenhaus, eds., *Lexical Ambiguity Resolution* (Morgan Kaufman, San Mateo, CA, 1988).
- [9] K. Vijay-Shankar and A.K. Joshi, Some computational properties of tree adjoining grammars, in: *Proc. 23rd Annual Meeting of The Association for Computational Linguistics* (1985) 82-93.
- [10] T. Winograd, *Understanding Natural Language* (Academic Press, New York, 1972).
- [11] T. Winograd, *Language as a Cognitive Process: Syntax* (Addison-Wesley, Reading, MA, 1983).
- [12] W. Woods, Transition network grammars for natural language analysis, *Comm. ACM* **13** (1970) 591-606.