Incrementality in Deterministic Dependency Parsing

Joakim Nivre
School of Mathematics and Systems Engineering
Växjö University
SE-35195 Växjö
Sweden
joakim.nivre@msi.vxu.se

Abstract

Deterministic dependency parsing is a robust and efficient approach to syntactic parsing of unrestricted natural language text. In this paper, we analyze its potential for incremental processing and conclude that strict incrementality is not achievable within this framework. However, we also show that it is possible to minimize the number of structures that require non-incremental processing by choosing an optimal parsing algorithm. This claim is substantiated with experimental evidence showing that the algorithm achieves incremental parsing for 68.9% of the input when tested on a random sample of Swedish text. When restricted to sentences that are accepted by the parser, the degree of incrementality increases to 87.9%.

1 Introduction

Incrementality in parsing has been advocated for at least two different reasons. The first is mainly practical and has to do with real-time applications such as speech recognition, which require a continually updated analysis of the input received so far. The second reason is more theoretical in that it connects parsing to cognitive modeling, where there is psycholinguistic evidence suggesting that human parsing is largely incremental (Marslen-Wilson, 1973; Frazier, 1987).

However, most state-of-the-art parsing methods today do not adhere to the principle of incrementality, for different reasons. Parsers that attempt to disambiguate the input completely — full parsing — typically first employ some kind of dynamic programming algorithm to derive a packed parse forest and then applies a probabilistic top-down model in order to select the most probable analysis (Collins, 1997; Charniak, 2000). Since the first step is essentially nondeterministic, this seems to rule out incrementality at least in a strict sense. By contrast, parsers that only partially disambiguate the input — partial parsing — are usually deterministic and construct the final analysis in one pass over the input (Abney, 1991; Daelemans et al., 1999). But since they normally output a sequence of unconnected phrases or chunks, they fail to satisfy the constraint of incrementality for a different reason.

Deterministic dependency parsing has recently been proposed as a robust and efficient method for syntactic parsing of unrestricted natural language text (Yamada and Matsumoto, 2003; Nivre, 2003). In some ways, this approach can be seen as a compromise between traditional full and partial parsing. Essentially, it is a kind of full parsing in that the goal is to build a complete syntactic analysis for the input string, not just identify major constituents. But it resembles partial parsing in being robust, efficient and deterministic. Taken together, these properties seem to make dependency parsing suitable for incremental processing, although existing implementations normally do not satisfy this constraint. For example, Yamada and Matsumoto (2003) use a multi-pass bottom-up algorithm, combined with support vector machines, in a way that does not result in incremental processing.

In this paper, we analyze the constraints on incrementality in deterministic dependency parsing and argue that strict incrementality is not achievable. We then analyze the algorithm proposed in Nivre (2003) and show that, given the previous result, this algorithm is optimal from the point of view of incrementality. Finally, we evaluate experimentally the degree of incrementality achieved with the algorithm in practical parsing.

2 Dependency Parsing

In a dependency structure, every word token is dependent on at most one other word token, usually called its *head* or *regent*, which

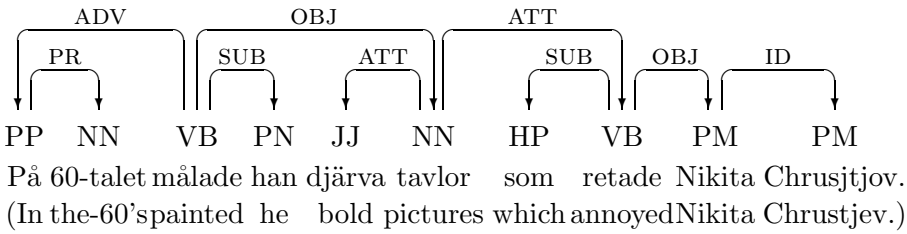

Figure 1: Dependency graph for Swedish sentence

means that the structure can be represented as a directed graph, with nodes representing word tokens and arcs representing dependency relations. In addition, arcs may be labeled with specific dependency types. Figure 1 shows a labeled dependency graph for a simple Swedish sentence, where each word of the sentence is labeled with its part of speech and each arc labeled with a grammatical function.

In the following, we will restrict our attention to *unlabeled* dependency graphs, i.e. graphs without labeled arcs, but the results will apply to labeled dependency graphs as well. We will also restrict ourselves to *projective* dependency graphs (Mel'cuk, 1988). Formally, we define these structures in the following way:

1. A dependency graph for a string of words $W = w_1 \cdots w_n$ is a labeled directed graph $D = (W, A)$, where
   (a) W is the set of nodes, i.e. word tokens in the input string,
   (b) A is a set of arcs (w_i, w_j) $(w_i, w_j \in W)$.

   We write $w_i < w_j$ to express that w_i precedes w_j in the string W (i.e., $i < j$); we write $w_i \rightarrow w_j$ to say that there is an arc from w_i to w_j; we use $\rightarrow^*$ to denote the reflexive and transitive closure of the arc relation; and we use $\leftrightarrow$ and $\leftrightarrow^*$ for the corresponding undirected relations, i.e. $w_i \leftrightarrow w_j$ iff $w_i \rightarrow w_j$ or $w_j \rightarrow w_i$.
2. A dependency graph $D = (W, A)$ is well-formed iff the five conditions given in Figure 2 are satisfied.

The task of mapping a string $W = w_1 \cdots w_n$ to a dependency graph satisfying these conditions is what we call *dependency parsing*. For a more detailed discussion of dependency graphs and well-formedness conditions, the reader is referred to Nivre (2003).

3 Incrementality in Dependency Parsing

Having defined dependency graphs, we may now consider to what extent it is possible to construct these graphs incrementally. In the strictest sense, we take incrementality to mean that, at any point during the parsing process, there is a single connected structure representing the analysis of the input consumed so far. In terms of our dependency graphs, this would mean that the graph being built during parsing is connected at all times. We will try to make this more precise in a minute, but first we want to discuss the relation between incrementality and determinism.

It seems that incrementality does not by itself imply determinism, at least not in the sense of never undoing previously made decisions. Thus, a parsing method that involves backtracking *can* be incremental, provided that the backtracking is implemented in such a way that we can always maintain a single structure representing the input processed up to the point of backtracking. In the context of dependency parsing, a case in point is the parsing method proposed by Kromann (Kromann, 2002), which combines heuristic search with different repair mechanisms.

In this paper, we will nevertheless restrict our attention to deterministic methods for dependency parsing, because we think it is easier to pinpoint the essential constraints within a more restrictive framework. We will formalize deterministic dependency parsing in a way which is inspired by traditional shift-reduce parsing for context-free grammars, using a buffer of input tokens and a stack for storing previously processed input. However, since there are no nonterminal symbols involved in dependency parsing, we also need to maintain a representation of the dependency graph being constructed during processing.

We will represent parser configurations by

Unique label	$(w_i \xrightarrow{r} w_j \wedge w_i \xrightarrow{r'} w_j) \Rightarrow r = r'$
Single head	$(w_i \rightarrow w_j \wedge w_k \rightarrow w_j) \Rightarrow w_i = w_k$
Acyclic	$\neg(w_i \rightarrow w_j \wedge w_j \rightarrow^* w_i)$
Connected	$w_i \leftrightarrow^* w_j$
Projective	$(w_i \leftrightarrow w_k \wedge w_i < w_j < w_k) \Rightarrow (w_i \rightarrow^* w_j \vee w_k \rightarrow^* w_j)$

Figure 2: Well-formedness conditions on dependency graphs

triples $\langle S, I, A\rangle$, where S is the stack (represented as a list), I is the list of (remaining) input tokens, and A is the (current) arc relation for the dependency graph. (Since the nodes of the dependency graph are given by the input string, only the arc relation needs to be represented explicitly.) Given an input string W, the parser is initialized to $\langle \mathbf{nil}, W, \emptyset\rangle$ and terminates when it reaches a configuration $\langle S, \mathbf{nil}, A\rangle$ (for any list S and set of arcs A). The input string W is *accepted* if the dependency graph $D = (W, A)$ given at termination is well-formed; otherwise W is *rejected*.

In order to understand the constraints on incrementality in dependency parsing, we will begin by considering the most straightforward parsing strategy, i.e. left-to-right bottom-up parsing, which in this case is essentially equivalent to shift-reduce parsing with a context-free grammar in Chomsky normal form. The parser is defined in the form of a transition system, represented in Figure 3 (where w_i and w_j are arbitrary word tokens):

1. The transition **Left-Reduce** combines the two topmost tokens on the stack, w_i and w_j, by a left-directed arc $w_j \rightarrow w_i$ and reduces them to the head w_j.
2. The transition **Right-Reduce** combines the two topmost tokens on the stack, w_i and w_j, by a right-directed arc $w_i \rightarrow w_j$ and reduces them to the head w_i.
3. The transition **Shift** pushes the next input token w_i onto the stack.

The transitions **Left-Reduce** and **Right-Reduce** are subject to conditions that ensure that the **Single head** condition is satisfied. For **Shift**, the only condition is that the input list is non-empty.

As it stands, this transition system is nondeterministic, since several transitions can often be applied to the same configuration. Thus, in order to get a deterministic parser, we need to introduce a mechanism for resolving transition conflicts. Regardless of which mechanism is used, the parser is guaranteed to terminate after at most $2n$ transitions, given an input string of length n. Moreover, the parser is guaranteed to produce a dependency graph that is acyclic and projective (and satisfies the single-head constraint). This means that the dependency graph given at termination is well-formed if and only if it is connected.

We can now define what it means for the parsing to be incremental in this framework. Ideally, we would like to require that the graph $(W - I, A)$ is connected at all times. However, given the definition of **Left-Reduce** and **Right-Reduce**, it is impossible to connect a new word without shifting it to the stack first, so it seems that a more reasonable condition is that the size of the stack should never exceed 2. In this way, we require every word to be attached somewhere in the dependency graph as soon as it has been shifted onto the stack.

We may now ask whether it is possible to achieve incrementality with a left-to-right bottom-up dependency parser, and the answer turns out to be no in the general case. This can be demonstrated by considering all the possible projective dependency graphs containing only three nodes and checking which of these can be parsed incrementally. Figure 4 shows the relevant structures, of which there are seven altogether.

We begin by noting that trees (2–5) can all be constructed incrementally by shifting the first two tokens onto the stack, then reducing – with **Right-Reduce** in (2–3) and **Left-Reduce** in (4–5) – and then shifting and reducing again – with **Right-Reduce** in (2) and (4) and **Left-Reduce** in (3) and (5). By contrast, the three remaining trees all require that three tokens are

Initialization	$\langle \mathbf{nil}, W, \emptyset \rangle$	
Termination	$\langle S, \mathbf{nil}, A \rangle$	
Left-Reduce	$\langle w_j w_i \vert S, I, A \rangle \rightarrow \langle w_j \vert S, I, A \cup \{(w_j, w_i)\} \rangle$	$\neg \exists w_k (w_k, w_i) \in A$
Right-Reduce	$\langle w_j w_i \vert S, I, A \rangle \rightarrow \langle w_i \vert S, I, A \cup \{(w_i, w_j)\} \rangle$	$\neg \exists w_k (w_k, w_j) \in A$
Shift	$\langle S, w_i \vert I, A \rangle \rightarrow \langle w_i \vert S, I, A \rangle$	

Figure 3: Left-to-right bottom-up dependency parsing

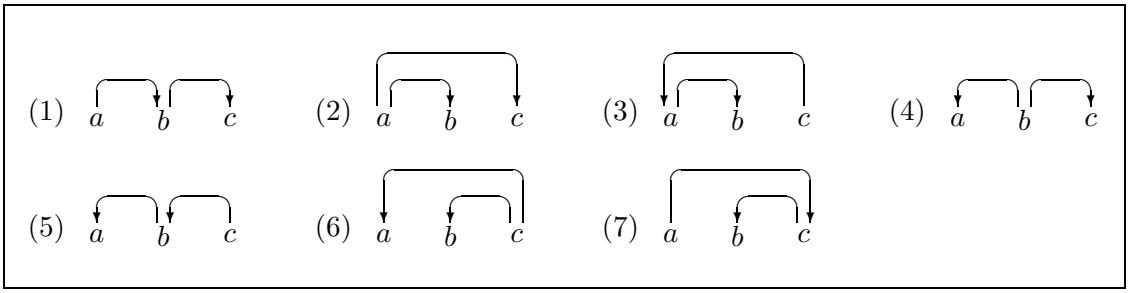

Figure 4: Projective three-node dependency structures

shifted onto the stack before the first reduction. However, the reason why we cannot parse the structure incrementally is different in (1) compared to (6–7).

In (6–7) the problem is that the first two tokens are not connected by a single arc in the final dependency graph. In (6) they are sisters, both being dependents on the third token; in (7) the first is the grandparent of the second. And in pure dependency parsing without nonterminal symbols, every reduction requires that one of the tokens reduced is the head of the other(s). This holds necessarily, regardless of the algorithm used, and is the reason why it is impossible to achieve strict incrementality in dependency parsing as defined here. However, it is worth noting that (2–3), which are the mirror images of (6–7) *can* be parsed incrementally, even though they contain adjacent tokens that are not linked by a single arc. The reason is that in (2–3) the reduction of the first two tokens makes the third token adjacent to the first. Thus, the defining characteristic of the problematic structures is that precisely the leftmost tokens are not linked directly.

The case of (1) is different in that here the problem is caused by the strict bottom-up strategy, which requires each token to have found all its dependents before it is combined with its head. For left-dependents this is not a problem, as can be seen in (5), which can be processed by alternating **Shift** and **Left-Reduce**. But in (1) the sequence of reductions has to be performed from right to left as it were, which rules out strict incrementality. However, whereas the structures exemplified in (6–7) can never be processed incrementally within the present framework, the structure in (1) can be handled by modifying the parsing strategy, as we shall see in the next section.

It is instructive at this point to make a comparison with incremental parsing based on extended categorial grammar, where the structures in (6–7) would normally be handled by some kind of concatenation (or product), which does not correspond to any real semantic combination of the constituents (Steedman, 2000; Morrill, 2000). By contrast, the structure in (1) would typically be handled by function composition, which corresponds to a well-defined compositional semantic operation. Hence, it might be argued that the treatment of (6–7) is only pseudo-incremental even in other frameworks.

Before we leave the strict bottom-up ap-

proach, it can be noted that the algorithm described in this section is essentially the algorithm used by Yamada and Matsumoto (2003) in combination with support vector machines, except that they allow parsing to be performed in multiple passes, where the graph produced in one pass is given as input to the next pass.[1] The main motivation they give for parsing in multiple passes is precisely the fact that the bottom-up strategy requires each token to have found all its dependents before it is combined with its head, which is also what prevents the incremental parsing of structures like (1).

4 Arc-Eager Dependency Parsing

In order to increase the incrementality of deterministic dependency parsing, we need to combine bottom-up and top-down processing. More precisely, we need to process left-dependents bottom-up and right-dependents top-down. In this way, arcs will be added to the dependency graph as soon as the respective head and dependent are available, even if the dependent is not complete with respect to its own dependents. Following Abney and Johnson (1991), we will call this *arc-eager parsing*, to distinguish it from the standard bottom-up strategy discussed in the previous section.

Using the same representation of parser configurations as before, the arc-eager algorithm can be defined by the transitions given in Figure 5, where w_i and w_j are arbitrary word tokens (Nivre, 2003):

1. The transition **Left-Arc** adds an arc $w_j \overset{r}{\rightarrow} w_i$ from the next input token w_j to the token w_i on top of the stack and pops the stack.
2. The transition **Right-Arc** adds an arc $w_i \overset{r}{\rightarrow} w_j$ from the token w_i on top of the stack to the next input token w_j, and pushes w_j onto the stack.
3. The transition **Reduce** pops the stack.
4. The transition **Shift** (**SH**) pushes the next input token w_i onto the stack.

The transitions **Left-Arc** and **Right-Arc**, like their counterparts **Left-Reduce** and **Right-Reduce**, are subject to conditions that ensure that the **Single head** constraint is satisfied, while the **Reduce** transition can only be applied if the token on top of the stack already has a head. The **Shift** transition is the same as before and can be applied as long as the input list is non-empty.

Comparing the two algorithms, we see that the **Left-Arc** transition of the arc-eager algorithm corresponds directly to the **Left-Reduce** transition of the standard bottom-up algorithm. The only difference is that, for reasons of symmetry, the former applies to the token on top of the stack and the next input token instead of the two topmost tokens on the stack. If we compare **Right-Arc** to **Right-Reduce**, however, we see that the former performs no reduction but simply shifts the newly attached right-dependent onto the stack, thus making it possible for this dependent to have right-dependents of its own. But in order to allow multiple right-dependents, we must also have a mechanism for popping right-dependents off the stack, and this is the function of the **Reduce** transition. Thus, we can say that the action performed by the **Right-Reduce** transition in the standard bottom-up algorithm is performed by a **Right-Arc** transition in combination with a subsequent **Reduce** transition in the arc-eager algorithm. And since the **Right-Arc** and the **Reduce** can be separated by an arbitrary number of transitions, this permits the incremental parsing of arbitrary long right-dependent chains.

Defining incrementality is less straightforward for the arc-eager algorithm than for the standard bottom-up algorithm. Simply considering the size of the stack will not do anymore, since the stack may now contain sequences of tokens that form connected components of the dependency graph. On the other hand, since it is no longer necessary to shift both tokens to be combined onto the stack, and since any tokens that are popped off the stack are connected to some token on the stack, we can require that the graph (S, A_S) should be connected at all times, where A_S is the restriction of A to S, i.e. $A_S = \{(w_i, w_j) \in A | w_i, w_j \in S\}$.

Given this definition of incrementality, it is easy to show that structures (2–5) in Figure 4 can be parsed incrementally with the arc-eager algorithm as well as with the standard bottom-up algorithm. However, with the new algorithm we can also parse structure (1) incrementally, as

[1] A purely terminological, but potentially confusing, difference is that Yamada and Matsumoto (2003) use the term **Right** for what we call **Left-Reduce** and the term **Left** for **Right-Reduce** (thus focusing on the position of the head instead of the position of the dependent).

Initialization	$\langle \mathbf{nil}, W, \emptyset \rangle$	
Termination	$\langle S, \mathbf{nil}, A \rangle$	
Left-Arc	$\langle w_i \vert S, w_j \vert I, A \rangle \rightarrow \langle S, w_j \vert I, A \cup \{(w_j, w_i)\} \rangle$	$\neg \exists w_k (w_k, w_i) \in A$
Right-Arc	$\langle w_i \vert S, w_j \vert I, A \rangle \rightarrow \langle w_j \vert w_i \vert S, I, A \cup \{(w_i, w_j)\} \rangle$	$\neg \exists w_k (w_k, w_j) \in A$
Reduce	$\langle w_i \vert S, I, A \rangle \rightarrow \langle S, I, A \rangle$	$\exists w_j (w_j, w_i) \in A$
Shift	$\langle S, w_i \vert I, A \rangle \rightarrow \langle w_i \vert S, I, A \rangle$	

Figure 5: Left-to-right arc-eager dependency parsing

is shown by the following transition sequence:

$$\begin{array}{rl} \langle \mathbf{nil}, abc, \emptyset \rangle & \\ \downarrow & (\mathbf{Shift}) \\ \langle a, bc, \emptyset \rangle & \\ \downarrow & (\mathbf{Right\text{-}Arc}) \\ \langle ba, c, \{(a,b)\} \rangle & \\ \downarrow & (\mathbf{Right\text{-}Arc}) \\ \langle cba, \mathbf{nil}, \{(a,b),(b,c)\} \rangle & \end{array}$$

We conclude that the arc-eager algorithm is optimal with respect to incrementality in dependency parsing, even though it still holds true that the structures (6–7) in Figure 4 cannot be parsed incrementally. This raises the question how frequently these structures are found in practical parsing, which is equivalent to asking how often the arc-eager algorithm deviates from strictly incremental processing. Although the answer obviously depends on which language and which theoretical framework we consider, we will attempt to give at least a partial answer to this question in the next section. Before that, however, we want to relate our results to some previous work on context-free parsing.

First of all, it should be observed that the terms *top-down* and *bottom-up* take on a slightly different meaning in the context of dependency parsing, as compared to their standard use in context-free parsing. Since there are no nonterminal nodes in a dependency graph, top-down construction means that a head is attached to a dependent before the dependent is attached to (some of) its dependents, whereas bottom-up construction means that a dependent is attached to its head before the head is attached to its head. However, top-down construction of dependency graphs does not involve the *prediction* of lower nodes from higher nodes, since all nodes are given by the input string. Hence, in terms of what drives the parsing process, all algorithms discussed here correspond to bottom-up algorithms in context-free parsing. It is interesting to note that if we recast the problem of dependency parsing as context-free parsing with a CNF grammar, then the problematic structures (1), (6–7) in Figure 4 all correspond to right-branching structures, and it is well-known that bottom-up parsers may require an unbounded amount of memory in order to process right-branching structure (Miller and Chomsky, 1963; Abney and Johnson, 1991).

Moreover, if we analyze the two algorithms discussed here in the framework of Abney and Johnson (1991), they do not differ at all as to the order in which *nodes* are enumerated, but only with respect to the order in which *arcs* are enumerated; the first algorithm is *arc-standard* while the second is *arc-eager*. One of the observations made by Abney and Johnson (1991), is that arc-eager strategies for context-free parsing may sometimes require less space than arc-standard strategies, although they may lead to an increase in local ambiguities. It seems that the advantage of the arc-eager strategy for dependency parsing with respect to structure (1) in Figure 4 can be explained along the same lines, although the lack of nonterminal nodes in dependency graphs means that there is no corresponding increase in local ambiguities. Although a detailed discussion of the relation between context-free parsing and dependency parsing is beyond the scope of this paper, we conjecture that this may be a genuine advantage of dependency representations in parsing.

Connected components	Parser configurations Number	Percent
0	1251	7.6
1	10148	61.3
2	2739	16.6
3	1471	8.9
4	587	3.5
5	222	1.3
6	98	0.6
7	26	0.2
8	3	0.0
≤ 1	11399	68.9
≤ 3	15609	94.3
≤ 8	16545	100.0

Table 1: Number of connected components in (S, A_S) during parsing

5 Experimental Evaluation

In order to measure the degree of incrementality achieved in practical parsing, we have evaluated a parser that uses the arc-eager parsing algorithm in combination with a memory-based classifier for predicting the next transition. In experiments reported in Nivre et al. (2004), a parsing accuracy of 85.7% (unlabeled attachment score) was achieved, using data from a small treebank of Swedish (Einarsson, 1976), divided into a training set of 5054 sentences and a test set of 631 sentences. However, in the present context, we are primarily interested in the incrementality of the parser, which we measure by considering the number of connected components in (S, A_S) at different stages during the parsing of the test data.

The results can be found in Table 1, where we see that out of 16545 configurations used in parsing 613 sentences (with a mean length of 14.0 words), 68.9% have zero or one connected component on the stack, which is what we require of a strictly incremental parser. We also see that most violations of incrementality are fairly mild, since more than 90% of all configurations have no more than three connected components on the stack.

Many violations of incrementality are caused by sentences that cannot be parsed into a well-formed dependency graph, i.e. a single projective dependency tree, but where the output of the parser is a set of internally connected components. In order to test the influence of incomplete parses on the statistics of incrementality, we have performed a second experiment, where we restrict the test data to those 444 sentences (out of 613), for which the parser produces a well-formed dependency graph. The results can be seen in Table 2. In this case, 87.1% of all configurations in fact satisfy the constraints of incrementality, and the proportion of configurations that have no more than three connected components on the stack is as high as 99.5%.

It seems fair to conclude that, although strict word-by-word incrementality is not possible in deterministic dependency parsing, the arc-eager algorithm can in practice be seen as a close approximation of incremental parsing.

6 Conclusion

In this paper, we have analyzed the potential for incremental processing in deterministic dependency parsing. Our first result is negative, since we have shown that strict incrementality is not achievable within the restrictive parsing framework considered here. However, we have also shown that the arc-eager parsing algorithm is optimal for incremental dependency parsing, given the constraints imposed by the overall framework. Moreover, we have shown that in practical parsing, the algorithm performs incremental processing for the majority of input structures. If we consider all sentences in the test data, the share is roughly two thirds, but if we limit our attention to well-formed output, it is almost 90%. Since deterministic dependency parsing has previously been shown to be competitive in terms of parsing accuracy (Yamada and Matsumoto, 2003; Nivre et al., 2004), we believe that this is a promising approach for situations that require parsing to be robust, efficient and (almost) incremental.

Connected components	Parser configurations Number	Percent
0	928	9.2
1	7823	77.8
2	1000	10.0
3	248	2.5
4	41	0.4
5	8	0.1
6	1	0.0
≤ 1	8751	87.1
≤ 3	9999	99.5
≤ 6	10049	100.0

Table 2: Number of connected components in (S, A_S) for well-formed trees

Acknowledgements

The work presented in this paper was supported by a grant from the Swedish Research Council (621-2002-4207). The memory-based classifiers used in the experiments were constructed using the Tilburg Memory-Based Learner (TiMBL) (Daelemans et al., 2003). Thanks to three anonymous reviewers for constructive comments on the submitted paper.

References

Steven Abney and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20:233–250.

Steven Abney. 1991. Parsing by chunks. In *Principle-Based Parsing*, pages 257–278. Kluwer.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings NAACL-2000*.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annatual Meeting of the Association for Computational Linguistics*, pages 16–23, Madrid, Spain.

Walter Daelemans, Sabine Buchholz, and Jorn Veenstra. 1999. Memory-based shallow parsing. In *Proceedings of the 3rd Conference on Computational Natural Language Learning (CoNLL)*, pages 77–89.

Walter Daelemans, Jakub Zavrel, Ko van der Sloot, and Antal van den Bosch. 2003. Timbl: Tilburg memory based learner, version 5.0, reference guide. Technical Report ILK 03-10, Tilburg University, ILK.

Jan Einarsson. 1976. Talbankens skriftspråkskonkordans. Lund University.

Lyn Frazier. 1987. Syntactic processing: Evidence from Dutch. *Natural Language and Linguistic Theory*, 5:519–559.

Matthias Trautner Kromann. 2002. Optimality parsing and local cost functions in Discontinuous Grammar. *Electronic Notes of Theoretical Computer Science*, 52.

William Marslen-Wilson. 1973. Linguistic structure and speech shadowing at very short latencies. *Nature*, 244:522–533.

Igor Mel'cuk. 1988. *Dependency Syntax: Theory and Practice*. State University of New York Press.

George A. Miller and Noam Chomsky. 1963. Finitary models of language users. In R. D. Luce, R. R. Bush, and E. Galanter, editors, *Handbook of Mathematical Psychology. Volume 2*. Wiley.

Glyn Morrill. 2000. Inremental processing and acceptability. *Computational Linguistics*, 26:319–338.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL)*, pages 49–56.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 195–206.