September 1988

Processing Crossed and Nested Dependencies: An Automaton Perspective on the Psycholinguistic Results

Aravind K. Joshi

University of Pennsylvania, joshi@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation

This paper is a revised paper originally presented at the First CUNY Workshop on Sentence Processing, CUNY, New York, February 1988.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/739
For more information, please contact libraryrepository@pobox.upenn.edu.
Processing Crossed and Nested Dependencies: An Automaton Perspective on the Psycholinguistic Results

Abstract
The clause-final verbal clusters in Dutch and German (and in general, in West Germanic languages) has been extensively studied in different syntactic theories. Standard Dutch prefers crossed dependencies (between verbs and their arguments) while Standard German prefers nested dependencies. Recently Bach, Brown, and Marslen-Wilson (1986) have investigated the consequences of these differences between Dutch and German for the processing complexity of sentences, containing either crossed or nested dependencies. Stated very simply, their results show that Dutch is 'easier' than German, thus showing that the push-down automaton (PDA) cannot be the universal basis for the human parsing mechanism. They provide an explanation for the inadequacy of PDA in terms of the kinds of partial interpretations the dependencies allow the listener to construct. Motivated by their results and their discussion of these results we introduce a principle of partial interpretation (PPI) and present an automaton, embedded push-down automaton (EPDA) which permits processing of crossed and nested dependencies consistent with PPI. We show that there are appropriate complexity measures (motivated by the discussion in Bach, Brown, and Marslen-Wilson (1986) according to which the processing of crossed dependencies is easier than the processing of nested dependencies. This EPDA characterization of the processing of crossed and nested dependencies is significant because EPDAs are known to be exactly equivalent to Tree Adjoining Grammars (TAG), which are also capable of providing a linguistically motivated analysis for the crossed dependencies of Dutch (Kroch and Santorini 1988). This significance is further enhanced by the fact that two other grammatical formalisms, (Head Grammars (Pollard, 1984) and Combinatory Grammars (Steedman, 1987), also capable of providing analysis for crossed dependencies of Dutch, have been recently shown to be equivalent to TAGS in their generative power. We have also briefly discussed some issues concerning the degree to which grammars directly encode the processing by automata, in accordance with PPI.

Comments
This paper is a revised paper orginally presented at the First CUNY Workshop on Sentence Processing, CUNY, New York, February 1988.
PROCESSING CROSSED AND NESTED DEPENDENCIES:
AN AUTOMATION PERSPECTIVE ON THE PSYCHOLINGUISTIC RESULTS
Aravind K. Joshi
MS-CIS-88-70
LINC LAB 128

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

September 1988

This paper is a revised paper originally presented at the First CUNY Workshop on Sentence Processing, CUNY, New York, February 1988.

Acknowledgements: This research was supported in part by DARPA grant N00014-85-K-0018, NSF grants MCS-8219196-CER, IRI84-10413-AO2, MCS 82-07294, DCR 84-10413 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.
PROCESSING CROSSED AND NESTED DEPENDENCIES:
AN AUTOMATON PERSPECTIVE ON THE
PSYCHOLINGUISTIC RESULTS*†‡

Aravind K. Joshi
Department of Computer and Information Science
Room 555, Moore School
University of Pennsylvania
Philadelphia, PA 19104

August 15, 1988

*This work is partially supported by NSF Grants MCS 82-19116-CER, MCS 82-07294, and DCR 84-10413.
†This paper is a revised version of a paper originally presented at the First CUNY Workshop on Sentence Processing, CUNY, New York, February 1988.
‡I want to thank Jack Hoeksema, Tony Kroch, Mitch Marcus, Mark Steedman, and David Weir for their valuable comments. Discussions with Tony Kroch and Mitch Marcus were crucial to the development of the complexity measures in Section 5. I have also profitted greatly from a talk on crossed dependencies presented by Emmon Bach at the Cognitive Science Colloquium at the University of Pennsylvania, October 1987.
Abstract

The clause-final verbal clusters in Dutch and German (and in general, in West Germanic languages) has been extensively studied in different syntactic theories. Standard Dutch prefers crossed dependencies (between verbs and their arguments) while Standard German prefers nested dependencies. Recently Bach, Brown, and Marslen-Wilson (1986) have investigated the consequences of these differences between Dutch and German for the processing complexity of sentences, containing either crossed or nested dependencies. Stated very simply, their results show that Dutch is 'easier' than German, thus showing that the push-down automaton (PDA) cannot be the universal basis for the human parsing mechanism. They provide an explanation for the inadequacy of PDA in terms of the kinds of partial interpretations the dependencies allow the listener to construct. Motivated by their results and their discussion of these results we introduce a principle of partial interpretation (PPI) and present an automaton, embedded push-down automaton (EPDA) which permits processing of crossed and nested dependencies consistent with PPI. We show that there are appropriate complexity measures (motivated by the discussion in Bach, Brown, and Marslen-Wilson (1986)) according to which the processing of crossed dependencies is easier than the processing of nested dependencies. This EPDA characterization of the processing of crossed and nested dependencies is significant because EPDAs are known to be exactly equivalent to Tree Adjoining Grammars (TAG), which are also capable of providing a linguistically motivated analysis for the crossed dependencies of Dutch (Kroch and Santorini 1988). This significance is further enhanced by the fact that two other grammatical formalisms, (Head Grammars (Pollard, 1984) and Combinatory Grammars (Steedman, 1987), also capable of providing analysis for crossed dependencies of Dutch, have been recently shown to be equivalent to TAGs in their generative power. We have also briefly discussed some issues concerning the degree to which grammars directly encode the processing by automata, in accordance with PPI.
1 Introduction

The clause-final verbal clusters in Dutch and German (and, in general, in West Germanic languages) have been extensively studied in different syntactic theories both from the point of view of their syntactic variation as well as on their own account (Evers, 1975; Zaenen, 1979; den Besten and Edmonson, 1983; Breman, Kaplan, Peters, and Zaenen, 1983; Ades and Steedman, 1985; Haegeman and van Riemsdijk, 1986; Kroch and Santorini, 1988, among others). The main observation for our purpose is that Standard Dutch prefers crossed dependencies (between verbs and their arguments), while Standard German prefers nested dependencies. Thus in Dutch we have

\[
\begin{array}{cccccc}
\text{NP}_1 & \text{NP}_2 & \text{NP}_3 & \text{V}_1 & \text{V}_2 & \text{V}_3 \\
\hline
\text{Jan} & \text{Piet} & \text{Marie} & \text{zag} & \text{laten} & \text{zwemmen} \\
\text{Jan} & \text{Piet} & \text{Marie} & \text{saw} & \text{make} & \text{swim} \\
\text{Jan saw Piet make Marie swim} \\
\end{array}
\]

In (1) \( NP_3 \) is an argument of \( V_3 \), \( NP_2 \) and \( S \) are arguments of \( V_2 \), and \( NP_1 \) and \( S \) are arguments of \( V_1 \). The dependencies between \( V_1, V_2, V_3 \) and their \( NP \) arguments, \( NP_1, NP_2, \) and \( NP_3 \) are \textit{crossed} as shown in (1). In contrast, in German we have

\[
\begin{array}{cccccc}
\text{NP}_1 & \text{NP}_2 & \text{NP}_3 & \text{V}_3 & \text{V}_2 & \text{V}_1 \\
\hline
\text{Hans} & \text{Peter} & \text{Marie} & \text{schwimmen} & \text{lassen} & \text{sah} \\
\text{Hans} & \text{Peter} & \text{Marie} & \text{swim} & \text{make} & \text{saw} \\
\text{Hans saw Peter make Marie swim} \\
\end{array}
\]

The dependencies between \( V_1, V_2, \) and \( V_3 \) and their \( NP \) arguments, \( NP_1, NP_2, \) and \( NP_3 \) are \textit{nested} as shown in (2).
Recently Bach, Brown, and Marslen-Wilson (1986) have investigated the consequences of these differences between Dutch and German for the processing complexity of sentences, containing either crossed or nested dependencies. Stated very simply, their results show that Dutch is ‘easier’ than German. More specifically, in their study “German and Dutch subjects performed two tasks—ratings of comprehensibility and a test of successful comprehension—on matched sets of sentences which varied in complexity from a simple sentence to one containing three levels of embedding.” Their results show “no difference between Dutch and German for sentences within the normal range (up to one level of embedding), but with a significant preference emerging for the Dutch crossed order for the more complex strings.” Based on these results they argue that “this rules out the push-down stack as the universal basis for the human parsing mechanism.” The following table (Table 1) summarizes some of their results. Note that levels in Table 1 refer to the number of verbs in the sentences and thus the level will be one more than the level of embedding in the sentence. The level for sentences (1) and (2) above is 3. Henceforth, this is what we mean by level, which is also in accordance with the notation in Bach, Brown, and Marslen-Wilson.

<table>
<thead>
<tr>
<th>Level of embedding</th>
<th>Dutch</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.14</td>
<td>1.16</td>
</tr>
<tr>
<td>2</td>
<td>2.34</td>
<td>2.58</td>
</tr>
<tr>
<td></td>
<td>(0.23)</td>
<td>(0.42)</td>
</tr>
<tr>
<td>3</td>
<td>5.42</td>
<td>5.80</td>
</tr>
<tr>
<td></td>
<td>(1.36)</td>
<td>(1.79)</td>
</tr>
<tr>
<td>4</td>
<td>7.66</td>
<td>7.86</td>
</tr>
<tr>
<td></td>
<td>(1.72)</td>
<td>(2.04)</td>
</tr>
</tbody>
</table>

Mean rating of comprehensibility (Test)

Difference in mean Test/Paraphrase ratings (numbers in parentheses)

As is evident from Table 1, Dutch is easier than German. At level 1, there is no difference; at level 2, the difference is small, still favoring Dutch; beyond level 2, Dutch is definitely easier than German. These results of Bach, Brown, and Marslen-Wilson confirm the intuitive claim of Evers (1975), in his syntactic study of these structures, that the crossed structures of Dutch are easier to process than the nested structures of German. Hoeksema (1981) made a similar claim in his study of Dutch vis-à-vis Frisian, which has nested dependencies.

These results of Bach, Brown, and Marslen-Wilson thus show that the push-down automaton (PDA) cannot be the universal basis for the human parsing mechanism. Bach, Brown, and Marslen-Wilson offer an explanation for the inadequacy of PDA based on the kinds of partial interpretations.
that the crossed and nested dependencies allow the listener to construct. Their main suggestion is "that the most important variable in successful parsing and interpretation is not simply when information becomes available, but also what you can do with that information when you get it." Thus in (2) (German example), when the deepest NP and V are reached, i.e., NP3 V3 (Marie schwimmen), we have a verb and its argument, however, we do not know at this stage where this structure belongs, i.e., we do not have a higher structure into which we can integrate this information. Hence, we must hold this information until a higher structure becomes available. The same consideration holds for NP2 V2. In contrast, in (1) (Dutch example), we can begin to build the matrix of higher verbs as soon as the verb cluster begins and the NP arguments can be integrated, without creating intermediate structures that do not have a place for them to fit into. The nested dependencies in German permit integration of structures (innermost to outermost) in a context-free manner (hence processed by a PDA) but it is not possible to decide what to do with this information until the higher verb(s) becomes available.

Motivated by the results of Bach, Brown, and Marslen-Wilson and their discussion of these results with respect to the inadequacies of PDA, we will introduce a principle of partial interpretation (PPI) which should be obeyed by an automaton if it is to be considered as a possible candidate for a universal mechanism for human sentence processing. PPI, as stated below, is an attempt to make some of the intuitions of Bach, Brown, and Marslen-Wilson more precise.

In an automaton, if a structure is popped, i.e., no longer stored by the automaton but discharged, possibly to another processor for further processing, the following conditions must hold:

1. The structure should be a properly integrated structure (with respect to the predicate-argument structure) and there should be a place for it to go, if it is expected to fit into another structure, i.e., the structure into which it will fit must have been popped already.

2. If a structure which has a slot for receiving another structure has been popped then the structure that will fill this slot will be popped next.

In this paper, we will present an automaton, embedded push-down automaton (EPDA), which will permit processing of crossed and nested dependencies consistent with PPI. We then show that there are appropriate complexity measures (also motivated by the discussion in Bach, Brown, and Marslen-Wilson), according to which the processing of crossed dependencies is easier than the processing of nested dependencies, thus correctly predicting the main results of Bach, Brown, and Marslen-Wilson. This EPDA characterization of the processing of crossed and nested dependencies is significant because EPDAs are known to be exactly equivalent to Tree Adjoining Grammars (TAG) in the sense that for any TAG, G, there is an EPDA, M, such that the language recognized by M, L(M) is exactly the language generated by G, L(G), and conversely for any EPDA, M', there is a TAG, G', such that L(M') = L(G'). TAGs were first introduced in Joshi, Levy and Takahashi.
(1975) and have been actively investigated since 1983 (e.g., Joshi, 1983; Kroch and Joshi, 1987; Joshi, 1987; Kroch, 1987; Vijay-Shanker, 1987; Weir, 1988; Joshi, Vijay-Shanker, and Weir. 1988).

TAGs are more powerful than context-free grammars, but only 'mildly' so, and are capable of providing a linguistically motivated analysis for the crossed dependencies in Dutch (Joshi, 1983; Kroch and Santorini, 1988). The significance of the EPDA characterization of crossed dependencies is enhanced even further because two other grammatical formalisms, (Head Grammars (Pollard, 1984) and Combinatory Categorial Grammars (Steedman, 1985, 1987)), based on principles completely different from those embodied in TAGs, which are also capable of providing analysis for crossed dependencies, have been shown to be equivalent to TAG in their generative power (Vijay-Shanker, Weir, and Joshi, 1985; Weir and Joshi, 1988).

The plan for the rest of the paper is as follows. In Sections 2 and 3, we will present a brief description of a push-down automaton (PDA) and an embedded push-down automaton (EPDA) respectively. In Section 4, we will show how EPDAs can process crossed and nested dependencies. Then in Section 5, we will consider some complexity measures for EPDAs, motivated by some of the discussion in Bach, Brown, and Marslen-Wilson, and show that with respect to both these measures, the crossed dependencies are easier to process than the nested dependencies. In Section 6, we will examine the relationship between EPDAs for processing crossed and nested dependencies, and their associated grammars. We will also briefly discuss some issues concerning the degree to which grammars directly encode the processing by automata, in accordance with PPI.

2 Push-Down Automaton (PDA)

A PDA, $M$, consists of a finite control (with a finite number of states), an input tape which is scanned from left to right, and a push-down store (pds), or stack for short. The pds or stack discipline is as follows: In a given move of $M$, a specified string of symbols can be written on top of the stack, or the top symbol of the stack can be popped.

$M$ starts in the initial state $S_0$ and the input head is on the leftmost symbol of the string on the input tape. The stack head is always on the top symbol of the stack. $Z_0$ is a special symbol marking the bottom of the stack. The behavior of $M$ is specified by a transition function, $\delta$, which, for some given input symbol and the state of the finite control and the stack symbol (i.e., the topmost symbol on the stack), specifies the new state, and whether the stack is pushed or popped. If pushed, then the transition function specifies the string pushed on the stack. If popped, then the topmost symbol of the stack is removed. The input head either moves one symbol to the right or stays on the current symbol. Thus

$$\delta (\text{input symbol, current state, stack symbol}) = (\text{new state, push/pop})$$
If $M$ is nondeterministic, then with a given input symbol, current state, and the stack symbol, more than one (new state, push/pop) pairs could be associated.

A string of symbols on the input tape is recognized (parsed, accepted) by $M$, if starting in the initial state and with the input head on the leftmost symbol of the input string, if there is a sequence of moves, as specified by $\delta$, such that the input head moves past the rightmost symbol on the input tape and the stack is empty. There are alternate ways of defining recognition, e.g., by $M$ entering one of the final states of $M$ after the input head has moved past the leftmost symbol on the input; however, in this paper, we will define acceptance by empty stack. It is well-known that these two definitions are equivalent.

PDA

[Diagram]

3 Embedded Push-Down Automaton (EPDA)

An EPDA, $M'$, is very similar to a PDA, except that the push-down store is not necessarily just one stack but a sequence of stacks. The overall stack discipline is similar to a PDA, i.e., the stack head will be always at the top symbol of the top stack, and if the stack head ever reaches the bottom of a stack, then the stack head automatically moves to the top of the stack below (or to the left of) the current stack, if there is one (Vijay-Shanker, 1987; Joshi, 1987; Joshi, Vijay-Shanker, and Weir, 1988).

Initially, $M'$ starts with only one stack, but unlike a PDA, an EPDA may create new stacks above and below (right and left of) the current stack. The behavior of $M$ is specified by a transition function, $\delta'$, which for a given input symbol, the state of the finite control, and the stack symbol, specifies the new state, and whether the current stack is pushed or popped; it also specifies new stacks to be created above and below the current stack. The number of stacks to be created above and below the current stack are specified by the move. Also, in each one of the newly created stacks, some specified finite strings of symbols can be written (pushed). Thus:

$$\delta' \text{ (input symbol, current state, stack symbol)} =$$

$$(\text{new state, } sb_1, sb_2, \ldots, sb_m, \text{push/pop on current stack, } st_1, st_2, \ldots, st_n)$$
where \( s_{b1}, s_{b2}, \ldots, s_{bm} \) are the stacks introduced below the current stack, and \( s_{t1}, s_{t2}, \ldots, s_{tn} \) are the stacks introduced above the current stack. As in the case of a PDA, an EPDA can be nondeterministic also.

A string of symbols on the input tape is recognized (parsed, accepted) by \( M' \), if starting in the initial state and with the input head on the leftmost symbol of the string on the input tape, if there is a sequence of moves as specified by \( \delta' \) such that the input head moves past the rightmost symbol on the input tape and the current stack is empty, and there are no more stacks below the current stack.

The following two diagrams illustrate moves of an EPDA, \( M' \).

Given the initial configuration as shown in (1), let us assume that for the given input symbol, the current state of the finite control, and the stack symbol, \( \delta' \) specifies the following move as shown in (2):

In this move, 2 stacks have been created above the current stack (which is shown by dotted lines), and 3 stacks have been created below the current stack. \( W \) has been pushed on the current stack.
stack, $X_0, X_1$ have been pushed on the stacks introduced above the current stack and $Y_0, Y_1, Y_2$ have been pushed on the stacks created below the current stack. The stack head has moved to the top of top stack, so now the topmost stack is the current stack.
Let us assume that in the next move the configuration is as shown in (3) below:

In this move, 1 stack has been created below the current stack (which is shown by dotted lines) with \( V_0 \) pushed on it, 2 stacks have been created above the current stack with \( T_0, T_1 \) pushed on them. \( V \) is pushed on the current stack. The stack head has again moved to the topmost of top stack.

Thus in an EPDA in a given configuration there is a sequence of stacks; however, the stack head is always at the top of the top stack at the end of a move. Thus although, unlike a PDA, there is a sequence of stack in a given configuration, the overall stack discipline is the same as in a PDA. PDAs are special cases of EPDAs, where in each move no new stacks are created, only a push/pop is carried out on the current stack.

4 Crossed and Nested Dependencies

We will now illustrate how EPDAs can process crosse and nested dependencies consistent with the principle of partial interpretation (PPI) described in Section 1.
Rather than defining the EPDA, $M_d$, formally, (i.e. specifying the transition function completely), we will simply describe the moves $M_d$ goes through during the processing of the input string. The symbols in the input string are indexed so as to bring out the dependencies explicitly and is thus for convenience only. Also NPs are treated as single symbols. In the initial configuration, the input head is on $NP_1$ and the stack head is on top of the current stack. The first three moves of $M_d$, i.e., moves 1, 2, and 3, push $NP_1, NP_2, NP_3$ on the stack. At the end of the third
move, the current stack has \( NP_1, NP_2, \) and \( NP_3 \) on it and the input head is on \( V_1 \). No new stacks have been created in these moves. In move 4, \( NP_3 \) is popped from the current stack and a new stack has been created below the current stack and \( V_3' \) is pushed on this stack. (The stack symbols have been indexed to show explicitly the relationship between the input symbols and stack symbols, thus this indexing is for convenience only). The symbol \( V_3' \) is assumed to encode the \( NP_3 \) argument together with the variable of type verb (which takes an \( NP \) argument), i.e., to encode a structure \( V(NP) \). In move 4, \( M_d \) has packaged \( NP_3 \) with a variable of type verb whose binding it expects to find later. At the end of move 4, the stack head is on top of the topmost stack, i.e., on \( NP_2 \) and the input head stays at \( V_1 \). Moves 5 and 6 are similar to move 4. In move 5, \( NP_2 \) is popped from the current stack and a new stack with \( V_3 \) on it is created below the current stack. Thus the stack containing \( V_3' \) appears between the stack containing \( V_3' \) and the current stack. \( V_3' \) encodes the \( NP_2 \) argument and a variable of type verb (which takes \( NP \) and an \( S \) as arguments), i.e., it encodes a structure \( V(NP, S) \). The input head stays at \( V_1 \). Similarly, in move 6, \( NP_1 \) is popped from the current stack and a new stack is created below the current stack, and \( V_4' \) is pushed on it. \( V_4' \) encodes the \( NP_1 \) argument and a variable of type verb (which takes \( NP \) and \( S \) arguments), i.e., it encodes a structure \( V(NP, S) \). The input head stays at \( V_1 \). The current stack is now empty and since there are stacks below the current stack, the stack head moves to the top of topmost stack below the empty current stack, i.e., it is on \( V_1' \). In move 7, \( V_1' \) is popped. In effect, we have matched \( V_1 \) from the input to \( V_1' \) and the structure \( V_1(NP_1, S) \) is now popped and is no longer held by \( M_d \). Note that this structure has its predicate and one argument filled in, and it has a slot for an \( S \) type argument, which will be filled in by the next package that is popped by \( M_d \). Thus we are following the principle of partial interpretation (PPI), as described in Section 1. Similarly in move 8, \( V_2 \) and \( V_2' \) are matched and \( V_2' \) is popped, i.e., the structure \( V_2(NP_2, S) \) is popped. This structure now fills in the \( S \) argument of the structure popped earlier, and it itself is ready to receive a structure to fill its \( S \) argument. In move 9, \( V_3 \) and \( V_3' \) are matched and \( V_3' \) is popped, i.e., the structure \( V_3(NP_3) \) is popped, which fills in the \( S \) argument of the structure previously popped. During the moves 7, 8, and 9, the input head moves one symbol to the right. Hence at the end of move 9, the input head is past the rightmost symbol on the input tape; also, the current stack is empty and there are no stacks below the current stack. Hence, the input string has been successfully recognized (parsed).

---

1 Although we are encoding a structure, only a bounded amount of information is stored in the EPDA stacks, the symbols \( NP, S, \) etc. are all atomic symbols. In an EPDA behaving as a parser, these symbols can be regarded as pointers to relevant structures, already constructed, and outside the EPDA.
Once again, we will describe the various moves of the EPDA, $M_g$, during the processing of the input string. We assume as before that an appropriate transition function has been defined for $M_g$ licensing the moves described below. Note that a PDA can process nested dependencies, but as discussed in Section 1, processing of nested dependencies by a PDA does not obey the principle of partial interpretation (PPI) in Section 1. The EPDA, $M_g$, described here does obey PPI.

In the initial configuration, the input head is on $NP_1$ and the stack head is on top of the current
stack. The first three moves of $M_g$ push $NP_1, NP_2,$ and $NP_3$ on the current stack. No new stacks are created in these moves. During these moves, the input head moves to the right one symbol at a time, so that at the end of move 3, the input head is on $V_3$. The first three moves of $M_d$ are similar to the first three moves of $M_d$, described earlier. In move 4, $NP_3$ is popped from the current stack, a new stack is created below the current stack with $V_3^*$ pushed on it, and the input head moves to $V_2$ on the input string. $V_3^*$ encoding the $NP_3$ argument together with the verb $V_3$ (which takes an $NP$ argument), i.e., it encodes the structure $V_3(NP_3)$. Note that $V_3^*$ is like $V_2^*$ in $M_d$ except that in $V_3^*$, we had a variable of type verb, while in $V_3^*$, we have the verb $V_3$ from the input. Thus in move 4, we have packaged $V_3$ and its argument and put it on a stack below the current stack. Moves 5, and 6 are similar to move 4. In move 5, $NP_2$ is popped from the current stack, a new stack is created below the current stack with $V_2^*$ encoding the $NP_2$ argument and the verb $V_2$ (which takes NP and S as arguments), i.e., $V_2^*$ encodes the structure $V_2(NP_2,S)$. In move 6, $NP_1$ is popped from the current stack, a new stack is created below the current stack with $V_1^*$ pushed on it, and the input head moves to the right of $V_1$. $V_1^*$ encodes the $NP_1$ argument and the verb $V_1$ (which takes NP and S as arguments), i.e., $V_1^*$ encodes the structure $V_1(NP_1,S)$. At the end of move 6, the current stack is empty. Since there are stacks below the current stack, the stack head moves to the top of topmost stack below the current stack, i.e., it will be on $V_1^*$. The input head is to the right of $V_1$ on the input tape. During moves 7, 8, and 9, the input head will stay where it is, $V_1^*, V_2^*$, and $V_3^*$ will be popped in that order, the stack head moving from $V_1^*$ to $V_2^*$ to $V_3^*$. In move 7, $V_1^*$ is popped, i.e., the structure $V_1(NP_1,S)$ is popped and it is no longer held by $M_g$. This structure has its predicate and one argument filled in, and it has a slot for S type argument, which will be filled in by the next package that is popped by $M_g$. This is consistent with the PPI. Similarly, in move 8, $V_2^*$ is popped, i.e., the structure $V_2(NP_2,S)$ is popped. This structure has its predicate and one argument filled and it has a slot for an S type argument. This structure itself fills in the S slot in the structure popped in move 7. In move 9, $V_3^*$ is popped, i.e., the structure $V_3(NP_3)$ is popped. This structure has its predicate and its argument filled in, and it itself fills in the S slot in the structure popped in move 8. At the end of move 9, the current stack is empty and there are stacks below the current stack and the input head is to the right of the rightmost symbol in the input tape, hence $M_g$ has successfully recognized (parsed) the input string, and interpretation has been built consistent with PPI.

5 Complexity of Processing

In Section 3, we have shown how an EPDA can process both the crossed and nested dependencies in accordance with PPI. The major result of Bach, Brown, and Marslen-Wilson (1986) as summarized in Section 1, is that the processing of crossed dependencies is ‘easier’ than the processing of nested
dependencies, as illustrated in Table 1 in Section 1. In this Section, we will show that if a suitable measure of complexity of processing is defined for an EPDA (in the spirit of the discussion in Bach, Brown, and Marslen-Wilson (1986)), the processing of crossed dependencies is indeed, 'easier' than the processing of 'nested' dependencies, thus suggesting that EPDAs can model the processing of crossed and nested dependencies consistent with the experimental results of Bach, Brown, and Marslen-Wilson (1986). The main significance of this result is not just that there is an automaton with the appropriate behavior but rather this behavior is achieved by a class of automata that exactly corresponds to a class of grammars (called Tree Adjoining Grammars (TAG)) which are adequate to characterize both the crossed and nested dependencies. We will discuss this topic later in some detail.

What sort of complexity measure is appropriate? Let us consider the EPDAs $M_d$ and $M_g$ in Figures 1 and 2. If we measure the complexity just in terms of the total number of moves, then there is no distinction between the processing of crossed and nested dependencies. In each case, we have exactly 9 moves, (we have 3 levels of embedding here), and similarly the number of moves for both cases will be the same for other levels of embedding.

Motivated by the discussion in Bach, Brown, and Marslen-Wilson (1986), we will consider a measure which involves only the number of items from the input that the EPDA has to store (we will not attach any cost to the moves themselves, i.e., consider them (nearly) instantaneous). In particular, the measure will be the maximum number of input items stored during the entire computation. Thus in Figure 1, $M_d$ stores 1, 2, and 3 items after moves 1, 2, and 3 respectively. After move 4, the number of items stored is still 3 because although $NP_3$ is popped, $V_3^+$ has $NP_3$ integrated in it (the $V$ in $V_3^+$ is not bound to any input item yet). Thus, after each one of the moves 5 and 6, we also have 3 items from input stored in $M_d$. After move 7, only 2 items are stored in $M_d$, after move 8, only 1, and after move 9, none. Thus the maximum number of input times stored during the entire computation is 3.

A similar computation shows that the maximum number of input items stored in the computation of $M_g$ (for the nested case as shown in Figure 2) is 5. After move 3 (as in the case of $M_d$), $M_g$ has stored 3 input items. After move 4, $M_g$ has stored 4 items because $V_3^+$ not only has $NP_3$ integrated in it but also $V_3$ from the input. Thus after move 5, $M_g$ has stored 5 items, and after move 6, 6 items. After move 7, only 4 items are stored, after move 8, only 2, and after move 9, none. Thus the maximum number of items stored is 6. However, it is possible to integrate moves 6 and 7, so that in move 6, we can immediately pop $V_4^+$, there is no need to first store it and then pop it in the next move. Thus after this newly defined move 6, $M_g$ has stored only 4 items. Hence, the maximum number of input items stored in the entire computation is 5. (We have followed here a strategy of redefining a move of $M_g$ to minimize its complexity. The idea is that by giving all the help which we can to $M_g$ and by not giving any extra help to $M_d$, if it still turns out that
the complexity of \( M_d \) is less than that of \( M_g \), then we will have succeeded in making a stronger argument for our automaton model.

Table 2 below summarizes the complexity of processing as measured by the maximum number of input items stored during the entire computation.

<table>
<thead>
<tr>
<th>Level of Embedding</th>
<th>Dutch</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

In Table 2, we have shown the relevant numbers for levels of embedding up to 4 only. It is possible to derive an exact formula for these numbers, but there is not much point in describing that formula because Bach, Brown, and Marslen-Wilson (1986) give their numbers up to level 4 only. It is unlikely that reliable experimental data can be obtained for levels beyond 4. So any complexity numbers beyond level 4 will be only mathematical curiosities.

In Table 2, the complexity numbers for \( M_d \) and \( M_g \) for level 1 are the same as one would expect. For level 2, the complexity for \( M_g \) is greater than the complexity for \( M_d \). In Table 1 (in Section 1), the complexity of processing nested dependencies is only slightly more than that of crossed dependencies. In our case, the difference is not insignificant. Thus in our model, the difficulty of processing nested dependencies shows up even at level 2.

We will now consider a somewhat more fine-grained measure of complexity, still in terms of the number of input items stored. Instead of just counting the number of input items stored, we will also pay attention to the number of time units an input item \( i \) is stored, the time unit is in terms of the movement of the input head and not in terms of machine operations. (As before, we will not attach any cost to the moves themselves, i.e., consider them (nearly) instantaneous).

Let us consider Figure 1 again. \( NP_1 \) is stored in move 1, i.e., after the input head moves past \( NP_1 \), it will continue to be stored until the input head is past \( NP_3 \). During moves 4, 5, 6, the input stays on \( V_1 \). In move 7, \( V'_1 \) (and therefore \( NP_1 \)) is popped. Thus if the time unit is measured in terms of the movement of the input head, \( NP_1 \) is stored for 3 time units. Similarly \( NP_2 \) and \( NP_3 \) are each stored for 3 time units. \( V_1, V_2, \) and \( V_3 \) are each stored for zero units. Hence, \( \sum_i T(i) = 9 \), where \( T(i) \) is the number of time units input item \( i \) is stored.

Now consider Figure 2. \( NP_1 \) is stored in move 1, i.e., after the input head moves past \( NP_1 \). It will continue to be stored until the input moves past \( V_1 \). During moves 7, 8, and 9, the input head
does not move. Hence, if the time unit is counted in terms of the movement of the input head, \( NP_1 \) is stored 6 time units. Similarly \( NP_2 \) is stored for 5 time units, \( NP_3 \) for 4 units, \( V_3 \) for 3 units, \( V_2 \) for 2 units, and \( V_1 \) for 1 unit. Thus \( \sum_i T(i) = 21 \). Once again, we can combine moves 6 and 7, i.e., there is no need to first store \( V_1^* \) and then pop it, we can pop it immediately. Thus the number of time units \( NP_1 \) is stored is reduced to 5, the number of time units for \( NP_2, NP_3, V_3, \) and \( V_2 \) are not affected. The number of time units \( V_1 \) is stored become zero. Hence, \( \sum_i T(i) = 19 \).

As before, we have followed the strategy of redefining moves of \( M_g \) to help reduce its complexity and not help \( M_d \) correspondingly. The reason for doing this is the same as before, i.e., even after helping \( M_g \) in this way, if we can show that the complexity of \( M_d \) is less than \( M_g \), then we will have succeeded in making a stronger argument for our automaton model. Table 3 summarizes the complexity of processing according to the \( \sum_i T(i) \) measure for different levels of embedding up to level 4. Once again, an exact formula for these numbers can be worked out for any level but there is not much point in presenting it, as the experimental data does not go beyond level 4. It can be easily seen that the overall behavior of our automaton model with respect to this somewhat fine-grained complexity measure is about the same as in Table 3.

### Table 3

\[
\sum_i T(i), \text{ where } T(i) \text{ is the number of time units an input item } i \text{ is stored}
\]

<table>
<thead>
<tr>
<th>Level of Embedding</th>
<th>Dutch</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>34</td>
</tr>
</tbody>
</table>

6 EPDA and the associated grammars

In Section 4, we have shown how crossed and nested dependencies can be processed by EPDAs, in accordance with the principle of partial interpretation (PPI), as described in Section 1. This result has a larger significance because EPDAs are exactly equivalent to the Tree Adjoining Grammars (TAG), which are capable of providing a linguistically motivated analysis for Dutch crossed dependencies (Joshi 1985, Kroch and Santorini 1988).

The fundamental insight on which the TAG formalism is based is that local co-occurrence relations can be factored apart from the expression of recursion and unbounded dependencies. A TAG consists of a set of elementary trees on which local dependencies are stated and an adjunction
operation, which composes elementary trees with one another to yield complex structures. The
elementary trees of a TAG are divided into initial trees and auxiliary trees (Fig. 3). Initial trees
have the form of the tree in α. The root node of an initial tree is labeled S or $\bar{S}$, its internal nodes
are all nonterminals (phrasal categories), and its frontier nodes are all lexical categories. Auxiliary
trees have the form of the tree $\beta$. The root node of an auxiliary tree is a phrasal category, which
we have labeled X, a nonterminal. Its frontier nodes are all lexical nodes except for one phrasal
node which has the same category label as the root node.

We now define adjunction as follows. Let $\alpha$ be an elementary tree with a nonterminal node
labeled $X$, and let $\beta$ be an auxiliary tree with a root node $X$, the foot node, by definition has the
label $X$ also. The tree $\gamma$ obtained by adjoining $\beta$ to $\alpha$ at the node labeled $X$ is defined as follows.
The subtree at $X$ in $\alpha$ is detached, the auxiliary tree $\beta$ is attached to $X$, and then the detached
subtree is attached to the foot node of $\beta$, in short, $\beta$ is inserted at $X$ in $\alpha$. Adjunction, so defined,
can be extended to derived trees in an obvious manner. There are other details such as constraints
on adjoining, but for our purpose the short description given above is adequate.

The following TAG, $G$, allows derivations of crossed dependencies. $\alpha$ is an elementary tree
and $\beta$ is an auxiliary tree. Note that in each tree the verb is “raised.” $\gamma_0, \gamma_1,$ and $\gamma_2$ describe
the derivation of (1) in Section 1. Indexing of $NP$s and $Vs$ are for convenience only. Note that
NP1, NP2, and NP3 are crossed with respect to V1, V2, and V3 but nested with respect to V1, V2, and V3
and V1 they are nested with respect to V1, V2, and V3, thus the crossed dependencies
between the $NP$s and (lexical) $Vs$ is achieved by pair of nested dependencies which are coordinated
through the (primed) $Vs$, which can be interpreted as traces. The moves of the EPDA, $M_d$ in Fig.
1 in Section 4 reflects the structure of the grammar $G$, quite directly.
(See Joshi 83 and Kroch and Santorini 87 or further details. We have slightly simplified the grammar given by Kroch and Santorini, without sacrificing the essential characteristics of the grammar).

For nested dependencies of German, Kroch and Santorini give the following TAG, $G'$ (Fig. 5). Note that here we do not have verb "raising". The derivation of (2) in Section 1 then consists of adjoining $\beta$ to the root $\alpha$, deriving a tree $\gamma$ and then adjoining $\beta$ to the root of $\gamma$, deriving a tree $\gamma'$. 

Fig. 4
resulting in the desired structure. We have not shown this derivation in Fig. 5. PDAs are special cases of EPDAs. Thus an EPDA, essentially following the discipline of a PDA, can process \( G' \), however, this EPDA will not be in accordance with the principle of partial interpretation (PPI). In Section 4, Fig. 2 we have presented an EPDA, \( M_2 \) which processes nested dependencies of German, in accordance with PPI. The TAG, \( G' \) does not directly map onto \( M_2 \). However, since for every EPDA there is an equivalent TAG, there is a TAG which is equivalent to \( M_2 \), say \( G'' \). (See Fig. 5 for \( G' \) and Fig. 6 for \( G'' \)). The derivation \( G'' \) consists of adjoining \( \beta \) to the interior \( S \) node \( \alpha \), marked by an arrow (and not to the root node of \( \alpha \) as in the case of \( G' \)), deriving a tree \( \gamma \) and then adjoining \( \beta \) to \( \gamma \) as before deriving \( \gamma' \), resulting in the derived structure. We have not shown this derivation in Fig. 6.

\[
G': \quad \alpha = S \\
\begin{array}{c}
NP \\
N \\
Marie
\end{array} \\
\begin{array}{c}
VP \\
V \\
schwimmen
\end{array}
\]

\[
\beta = S \\
\begin{array}{c}
NP \\
N \\
Peter
\end{array} \\
\begin{array}{c}
VP \\
S \\
Hans
\end{array} \\
\begin{array}{c}
V \\
sah
\end{array}
\]

Fig. 5

\[
G'': \quad \alpha = S \\
\begin{array}{c}
NP \\
N \\
Marie
\end{array} \\
\begin{array}{c}
VP \\
V \\
schwimmen
\end{array} \\
\begin{array}{c}
\text{arrow}
\end{array} \\
\begin{array}{c}
\gamma^* \\
S \\
V^*
\end{array}
\]

\[
\beta = S \\
\begin{array}{c}
NP \\
N \\
Peter
\end{array} \\
\begin{array}{c}
VP \\
S \\
Hans
\end{array} \\
\begin{array}{c}
V \\
sah
\end{array}
\]

Fig. 6
$G''$ is clearly a TAG and it is a TAG for the nested dependencies. This grammar directly maps into $M_g$ in the sense that it encodes, in a way, the behavior of $M_g$, which corresponds to withholding the $NPVP$s until the top level structure is reached and then discharging them in the reverse order. $G''$ differs from $G'$ only in this respect. Clearly, $G'$ is a kind of grammar a linguist would write (based on the usual distributional considerations). $G''$ is like $G'$ except that the structure represented by the right daughter of the root node and the associated link (co-indexing) with the middle $S$ node is an encoding (in the grammar) of that part of the behavior of $M_g$ just mentioned above. $G''$ is not the kind of grammar a linguist would write (based on distributional considerations), however, it is closely related to $G'$.

For the Dutch case, TAG, $G$ maps more directly onto the EPDA, $M_d$ as compared to the mapping between the TAG, $G'$ for German and the EPDA, $M_g$. The TAG $G''$ for German maps more directly onto $M_g$. Thus, in some sense, the TAG, $G$ for Dutch reflects more directly the principle of partial interpretation (PPI) than the TAG, $G'$ for German. We need to construct $G''$ for German if we want a more direct encoding of PPI in the grammar itself. Another way of saying this is that the 'natural' grammar for the crossed dependencies of Dutch reflects PPI more directly than the 'natural' grammar for the nested dependencies of German. A reflex of this disparity is then the relative processing difficulty of nested dependencies.

7 Conclusion

Motivated by the results of Bach, Brown, and Marslen-Wilson (1986) and their discussion of these results concerning processing of crossed and nested dependencies, we have shown that embedded push-down automaton (EPDA) permits processing of crossed and nested dependencies consistent with the principle of partial interpretation. We have shown that there are appropriate complexity measures according to which the processing of crossed dependencies is easier than the processing of nested dependencies. This EPDA characterization is significant because the EPDAs are exactly equivalent to Tree Adjoining Grammars, which are capable of providing a linguistically motivated analysis for the crossed dependencies. The significance of EPDA characterization is further enhanced because two other formalisms (Head Grammars and Combinatory Categorial Grammars), based on principles completely different from those embodied in TAGs, which are also capable of providing analysis for crossed dependencies, are known to be equivalent to TAGs. We have also briefly discussed some issues concerning the degree to which grammars directly encode the processing by automata, in accordance with the principle of partial interpretation.

2By a grammar we mean the set of elementary trees and the constraints that specify what auxiliary trees are adjoinable at what nodes in each elementary tree. These constraints are not shown explicitly in $G, G'$, and $G''$. 

21
References


