Know When to Hold 'Em: Shuffling Deterministically in a Parser for Nonconcatenative Grammars*

Robert T. Kasper, Mike Calcagno, and **Paul C. Davis**
Department of Linguistics, Ohio State University
222 Oxley Hall
1712 Neil Avenue
Columbus, OH 43210 U.S.A.
Email: {kasper,calcagno,pcdavis}@ling.ohio-state.edu

Abstract

Nonconcatenative constraints, such as the shuffle relation, are frequently employed in grammatical analyses of languages that have more flexible ordering of constituents than English. We show how it is possible to avoid searching the large space of permutations that results from a nondeterministic application of shuffle constraints. The results of our implementation demonstrate that deterministic application of shuffle constraints yields a dramatic improvement in the overall performance of a head-corner parser for German using an HPSG-style grammar.

1 Introduction

Although there has been a considerable amount of research on parsing for constraint-based grammars in the HPSG (Head-driven Phrase Structure Grammar) framework, most computational implementations embody the limiting assumption that the constituents of phrases are combined only by concatenation. The few parsing algorithms that have been proposed to handle more flexible linearization constraints have not yet been applied to nontrivial grammars using nonconcatenative constraints. For example, van Noord (1991; 1994) suggests that the head-corner parsing strategy should be particularly well-suited for parsing with grammars that admit discontinuous constituency, illustrated with what he calls a "tiny" fragment of Dutch, but his more recent development of the head-corner parser (van Noord, 1997) only documents its use with purely concatenative grammars. The conventional wisdom has been that the large search space resulting from the use of such constraints (e.g., the shuffle relation) makes parsing too inefficient for most practical applications. On the other hand, grammatical analyses of languages that have more flexible ordering of constituents than English make frequent use of constraints of this type. For example, in recent work by Dowty (1996), Reape (1996), and Kathol (1995), in which linear order constraints are taken to apply to domains distinct from the local trees formed by syntactic combination, the nonconcatenative `shuffle` relation is the basic operation by which these word order domains are formed. Reape and Kathol apply this approach to various flexible word-order constructions in German.

A small sampling of other nonconcatenative operations that have often been employed in linguistic descriptions includes Bach's (1979) wrapping operations, Pollard's (1984) head-wrapping operations, and Moortgat's (1996) extraction and infixation operations in (categorial) type-logical grammar.

What is common to the proposals of Dowty, Reape, and Kathol, and to the particular analysis implemented here, is the characterization of natural language syntax in terms of two interrelated but in principle distinct sets of constraints: (a) constraints on an unordered *hierarchical structure*, projected from (grammatical-relational or semantic) valence properties of lexical items; and (b) constraints on the linear order in which elements appear. In this type of framework, constraints on linear order may place conditions on the the relative order of constituents that are not siblings in the hierarchical structure. To this end, we follow Reape and Kathol and utilize *order domains*, which are associated with each node of the hierarchical structure, and serve as the domain of application for linearization constraints.

In this paper, we show how it is possible to avoid searching the large space of permutations that results from a nondeterministic application of shuffle constraints. By delaying the application of shuffle constraints until the linear position of each element is known, and by using an efficient encoding of the portions of the input covered by each element of an order domain, shuffle constraints can be applied deterministically. The results of our implementation demonstrate that this optimization of shuffle constraints yields a dramatic improvement in the overall performance of a head-corner parser for German.

The remainder of the paper is organized as follows: §2 introduces the nonconcatenative fragment

* This research was sponsored in part by National Science Foundation grant SBR-9410532, and in part by a seed grant from the Ohio State University Office of Research; the opinions expressed here are solely those of the authors.

(1) Seiner Freundin liess er ihn helfen
his(DAT) friend(FEM) allows he(NOM) him(ACC) help
'He allows him to help his friend.'

(2) Hilft sie ihr schnell
help she(NOM) her(DAT) quickly
'Does she help her quickly?'

(3) Der Vater denkt dass sie ihr seinen Sohn helfen liess
The(NOM) father thinks that she(NOM) her(DAT) his(ACC) son help allows
'The father thinks that she allows his son to help her.'

(4)
$$\left[\begin{array}{l} \textit{r\_decl} \\ \text{DOM} \left\langle \left[\begin{array}{l}\textit{dom\_obj}\\ \text{PHON } \textit{seiner Freundin}\\ \text{SYNSEM } \textit{NP}\\ \text{TOPO } \textit{vf}\end{array}\right], \left[\begin{array}{l}\textit{dom\_obj}\\ \text{PHON } \textit{liess}\\ \text{SYNSEM } \textit{V}\\ \text{TOPO } \textit{cf}\end{array}\right], \left[\begin{array}{l}\textit{dom\_obj}\\ \text{PHON } \textit{er}\\ \text{SYNSEM } \textit{NP}\\ \text{TOPO } \textit{mf}\end{array}\right], \left[\begin{array}{l}\textit{dom\_obj}\\ \text{PHON } \textit{ihn}\\ \text{SYNSEM } \textit{NP}\\ \text{TOPO } \textit{mf}\end{array}\right], \left[\begin{array}{l}\textit{dom\_obj}\\ \text{PHON } \textit{helfen}\\ \text{SYNSEM } \textit{V}\\ \text{TOPO } \textit{vc}\end{array}\right] \right\rangle \\ \ldots \end{array}\right]$$

(5) $[\text{TOPO } vf] \prec [\text{TOPO } cf] \prec [\text{TOPO } mf] \prec [\text{TOPO } vc] \prec [\text{TOPO } nf]$

Figure 1: **Linear order of German clauses.**

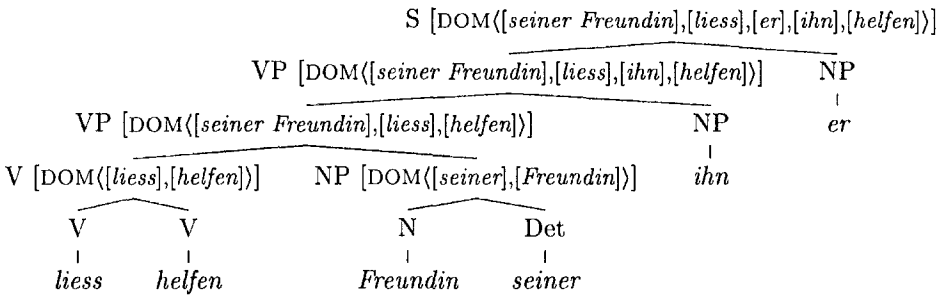

Figure 2: **Hierarchical structure of sentence (1).**

of German which forms the basis of our study; §3 describes the head-corner parsing algorithm that we use in our implementation; §4 discusses details of the implementation, and the optimization of the shuffle constraint is explained in §5; §6 compares the performance of the optimized and non-optimized parsers.

2 A German Grammar Fragment

The fragment is based on the analysis of German in Kathol's (1995) dissertation. Kathol's approach is a variant of HPSG, which merges insights from both Reape's work and from descriptive accounts of German syntax using topological fields (linear position classes). The fragment covers (1) root declarative (verb-second) sentences, (2) polar interrogative (verb-first) clauses and (3) embedded subordinate (verb-final) clauses, as exemplified in Figure 1.

The linear order of constituents in a clause is represented by an order domain (DOM), which is a list of domain objects, whose relative order must satisfy a set of linear precedence (LP) constraints. The order domain for example (1) is shown in (4). Notice that each domain object contains a TOPO attribute, whose value specifies a topological field that partially determines the object's linear position in the list. Kathol defines five topological fields for German clauses: Vorfeld (*vf*), Comp/Left Sentence Bracket (*cf*), Mittelfeld (*mf*), Verb Cluster/Right Sentence Bracket (*vc*), and Nachfeld (*nf*). These fields are ordered according to the LP constraints shown in (5).

The hierarchical structure of a sentence, on the other hand, is constrained by a set of immediate dominance (ID) schemata, three of which are included in our fragment: *Head-Argument* (where "Argument" subsumes complements, subjects, and specifiers), *Adjunct-Head*, and *Marker-Head*. The Head-

Argument schema is shown below, along with the constraints on the order domain of the mother constituent. In all three schemata, the domain of a non-head daughter is compacted into a single domain object, which is shuffled together with the domain of the head daughter to form the domain of the mother.

(6) Head-Argument Schema (simplified)

$$\left[\begin{array}{l} \text{SYNSEM}\ \left[\begin{array}{l}\text{HEAD}\ \boxed{1}\\ \text{SUBCAT}\ \boxed{3}\end{array}\right] \\ \text{DOM}\ \boxed{6} \\ \text{HEAD-DTR}\ \left[\begin{array}{l}\text{SYNSEM}\ \left[\begin{array}{l}\text{HEAD}\ \boxed{1}\\ \text{SUBCAT}\ \langle \boxed{2} \| \boxed{3} \rangle\end{array}\right] \\ \text{DOM}\ \boxed{4}\end{array}\right] \\ \text{ARG-DTR}\ \boxed{5}\left[\text{SYNSEM}\ \boxed{2}\right] \end{array}\right]$$

$$\wedge\ \texttt{shuffle}(\boxed{4},\ \texttt{compaction}(\boxed{5}),\ \boxed{6})$$

$$\wedge\ \texttt{order\_constraints}(\boxed{6})$$

The hierarchical structure of (1) is shown by the unordered tree of Figure 2, where head daughters appear on the left at each branch. Focusing on the NP *seiner Freundin* in the tree, it is compacted into a single domain object, and must remain so, but its position is not fixed relative to the other arguments of *liess* (which include the raised arguments of *helfen*). The shuffle constraint allows this single, compacted domain object to be realized in various permutations with respect to the other arguments, subject to the LP constraints, which are implemented by the `order_constraints` predicate in (6). Each NP argument may be assigned either *vf* or *mf* as its TOPO value, subject to the constraint that root declarative clauses must contain exactly one element in the *vf* field. In this case, *seiner Freundin* is assigned *vf*, while the other NP arguments of *liess* are in *mf*. However, the following permutations of (1) are also grammatical, in which *er* and *ihn* are assigned to the *vf* field instead:

(7) a. *Er liess ihn seiner Freundin helfen.*
b. *Ihn liess er seiner Freundin helfen.*

Comparing the hierarchical structure in Figure 2 with the linear order domain in (4), we see that some daughters in the hierarchical structure are realized discontinuously in the order domain for the clause (e.g., the verbal complex *liess helfen*). In such cases, nonconcatenative constraints, such as shuffle, can provide a more succinct analysis than concatenative rules. This situation is quite common in languages like German and Japanese, where word order is not totally fixed by grammatical relations.

3 Head-Corner Parsing

The grammar described above has a number of properties relevant to the choice of a parsing strategy. First, as in HPSG and other constraint-based grammars, the lexicon is information-rich, and the combinatory or phrase structure rules are highly schematic. We would thus expect a purely top-down algorithm to be inefficient for a grammar of this type, and it may even fail to terminate, for the simple reason that the search space would not be adequately constrained by the highly general combinatory rules.

Second, the grammar is essentially nonconcatenative, i.e., constituents of the grammar may appear discontinuously in the string. This suggests that a strict left-to-right or right-to-left approach may be less efficient than a bidirectional or non-directional approach.

Lastly, the grammar is head-driven, and we would thus expect the most appropriate parsing algorithm to take advantage of the information that a semantic head provides. For example, a head usually provides information about the remaining daughters that the parser must find, and (since the head daughter in a construction is in many ways similar to its mother category) effective top-down identification of candidate heads should be possible.

One type of parser that we believe to be particularly well-suited to this type of grammar is the head-corner parser, introduced by van Noord (1991; 1994) based on one of the parsing strategies explored by Kay (1989). The head-corner parser can be thought of as a generalization of a left-corner parser (Rosenkrantz and Lewis-II, 1970; Matsumoto et al., 1983; Pereira and Shieber, 1987).[1]

The outstanding features of parsers of this type are that they are head-driven, of course, and that they process the string bidirectionally, starting from a lexical head and working outward. The key ingredients of the parsing algorithm are as follows:

- Each grammar rule contains a distinguished daughter which is identified as the head of the rule.[2]
- The relation *head-corner* is defined as the reflexive and transitive closure of the head relation.
- In order to prove that an input string can be parsed as some (potentially complex) goal category, the parser nondeterministically selects a potential head of the string and proves that this head is the head-corner of the goal.
- Parsing proceeds from the head, with a rule being chosen whose head daughter can be instantiated by the selected head word. The other daughters of the rule are parsed recursively in a bidirectional fashion, with the result being a slightly larger head-corner.

[1] In fact, a head-corner parser for a grammar in which the head daughter in each rule is the leftmost daughter will function as a left-corner parser.

[2] Note that the fragment of the previous section has this property.

- The process succeeds when a head-corner is constructed which dominates the entire input string.

4 Implementation

We have implemented the German grammar and head-corner parsing algorithm described in §2 and §3 using the ConTroll formalism (Götz and Meurers, 1997). ConTroll is a constraint logic programming system for typed feature structures, which supports a direct implementation of HPSG. Several properties of the formalism are crucial for the approach to linearization that we are investigating: it does not require the grammar to have a context-free backbone; it includes definite relations, enabling the definition of nonconcatenative constraints, such as `shuffle`; and it supports delayed evaluation of constraints. The ability to control when relational contraints are evaluated is especially important in the optimization of `shuffle` to be discussed next (§5). ConTroll also allows a parsing strategy to be specified within the same formalism as the grammar.[3] Our implementation of the head-corner parser adapts van Noord's (1997) parser to the ConTroll environment.

5 Shuffling Deterministically

A standard definition of the shuffle relation is given below as a Prolog predicate.

```
% shuffle (unoptimized version)
shuffle([],[],[]).
shuffle([X|S1], S2, [X|S3]) :-
        shuffle(S1,S2,S3).
shuffle(S1, [X|S2], [X|S3]) :-
        shuffle(S1,S2,S3).
```

The use of a shuffle constraint reflects the fact that several permutations of constituents may be grammatical. If we parse in a bottom-up fashion, and the order domains of two daughter constituents are combined as the first two arguments of `shuffle`, multiple solutions will be possible for the mother domain (the third argument of `shuffle`). For example, in the structure shown earlier in Figure 2, when the domain ⟨[*liess*],[*helfen*]⟩ is combined with the compacted domain element ⟨[*seiner Freundin*]⟩, `shuffle` will produce three solutions:

(8) a. ⟨[*liess*],[*helfen*],[*seiner Freundin*] ⟩
b. ⟨[*liess*],[*seiner Freundin*],[*helfen*] ⟩
c. ⟨[*seiner Freundin*],[*liess*],[*helfen*] ⟩

This set of possible solutions is further constrained in two ways: it must be consistent with the linear precedence constraints defined by the grammar, and it must yield a sequence of words that is identical to the input sequence that was given to the parser. However, as it stands, the correspondence with the input sequence is only checked after an order domain is proposed for the entire sentence. The order domains of intermediate phrases in the hierarchical structure are not directly constrained by the grammar, since they may involve discontinuous subsequences of the input sentence. The shuffle constraint is acting as a generator of possible order domains, which are then filtered first by LP constraints and ultimately by the order of the words in the input sentence. Although each possible order domain that satisfies the LP constraints is a grammatical sequence, it is useless, in the context of parsing, to consider those permutations whose order diverges from that of the input sentence. In order to avoid this very inefficient generate-and-test behavior, we need to provide a way for the input positions covered by each proposed constituent to be considered sooner, so that the only solutions produced by the shuffle constraint will be those that correspond to the order of words in the actual input sequence.

Since the portion of the input string covered by an order domain may be discontinuous, we cannot just use a pair of endpoints for each constituent as in chart parsers or DCGs. Instead, we adapt a technique described by Reape (1991), and use bitstring codes to represent the portions of the input covered by each element in an order domain. If the input string contains n words, the code value for each constituent will be a bitstring of length n. If element i of the bitstring is 1, the constituent contains the ith word of the sentence, and if element i of the bitstring is 0, the constituent does not contain the ith word. Reape uses bitstring codes for a tabular parsing algorithm, different from the head-corner algorithm used here, and attributes the original idea to Johnson (1985).

The optimized version of the shuffle relation is defined below, using a notation in which the arguments are descriptions of typed feature structures. The actual implementation of relations in the ConTroll formalism uses a slightly different notation, but we use a more familiar Prolog-style notation here.[4]

[3] An interface from ConTroll to the underlying Prolog environment was also developed to support some optimizations of the parser, such as memoization and the operations over bitstrings described in §5.

[4] Symbols beginning with an upper-case letter are variables, while lower-case symbols are either attribute labels (when followed by ':') or the types of values (e.g., `ne_list`).

```
% shuffle (optimized version)
shuffle([], [], []).
shuffle((S1&ne_list), [], S1).
shuffle([], (S2&ne_list), S2).
shuffle(S1, S2, S3) :-
    S1=[(code:C1)|_], S2=[(code:C2)|_],
    code_prec(C1,C2,Bool),
    shuffle_d(Bool,S1,S2,S3).

% shuffle_d(Bool, [H1|T1], [H2|T2], List).
% Bool=true: H1 precedes H2
% Bool=false: H1 does not precede H2
shuffle_d(true, [H1|S1], S2, [H1|S3]) :-
    may_precede_all(H1,S2),
    shuffle(S1,S2,S3).
shuffle_d(false, S1, [H2|S2], [H2|S3]) :-
    may_precede_all(H2,S1),
    shuffle(S1,S2,S3).
```

This revision of the shuffle relation uses two auxiliary relations, `code_prec` and `shuffle_d`. `code_prec` compares two bitstrings, and yields a boolean value indicating whether the first string precedes the second (the details of the implementation are suppressed). The result of a comparison between the codes of the first element of each domain is used to determine which element must appear first in the resulting domain. This is implemented by using the boolean result of the code comparison to select a unique disjunct of the `shuffle_d` relation. The `shuffle_d` relation also incorporates an optimization in the checking of LP constraints. As each element is shuffled into the result, it only needs to be checked for LP acceptability with the elements of the other argument list, because the LP constraints have already been satisfied on each of the argument domains. Therefore, LP acceptability no longer needs to be checked for the entire order domain of each phrase, and the call to `order_constraints` can be eliminated from each of the phrasal schemata.

In order to achieve the desired effect of making shuffle constraints deterministic, we must delay their evaluation until the code attributes of the first element of each argument domain have been instantiated to a specific string. Using the analogy of a card game, we must hold the cards (delay shuffling) until we know what their values are (the codes must be instantiated). The delayed evaluation is enforced by the following declarations in the ConTroll system, where `argn:@type` specifies that evaluation should be delayed until the value of the nth argument of the relation has a value more specific than `type`:

```
delay(code_prec,
      (arg1:@string & arg2:@string)).
delay(shuffle_d, arg1:@bool).
```

With the addition of CODE values to each domain element, the input to the shuffle constraint in our previous example is shown below, and the unique solution for *MDom* is the one corresponding to (8c).

(9) shuffle(⟨ [PHON *liess*, CODE 001000], [PHON *helfen*, CODE 000001] ⟩, ⟨ [PHON *seiner Freundin*, CODE 110000] ⟩, *MDom*)

6 Performance Comparison

In order to evaluate the reduction in the search space that is achieved by shuffling deterministically, the parser with the optimized shuffle constraints and the parser with the nonoptimized constraints were each tested with the same grammar of German on a set of 30 sentences of varying length, complexity and clause types. Apart from the redefinition of the shuffle relation, discussed in the previous section, the only differences between the grammars used for the optimized and unoptimized tests are the addition of CODE values for each domain element in the optimized version and the constraints necessary to propagate these code values through the intermediate structures used by the parser.

A representative sample of the tested sentences is given in Table 2 (because of space limitations, English glosses are not given, but the words have all been glossed in §2), and the performance results for these 12 sentences are listed in Table 1. For each version of the parser, time, choice points, and calls are reported, as follows: The time measurement (Time)[5] is the amount of CPU seconds (on a Sun SPARCstation 5) required to search for all possible parses, choice points (ChoicePts) records the number of instances where more than one disjunct may apply at the time when a constraint is resolved, and calls (Calls) lists the number of times a constraint is unfolded. The number of calls listed includes all constraints evaluated by the parser, not only shuffle constraints. Given the nature of the ConTroll implementation, the number of calls represents the most basic number of steps performed by the parser at a logical level. Therefore, the most revealing comparison with regard to performance improvement between the optimized and nonoptimized versions is the *call factor*, given in the last column of Table 1. The call factor for each sentence is the number of nonoptimized calls divided by the number of optimized calls. For example, in T1, *Er hilft ihr*, the version using the nonoptimized shuffle was required to make 4.1 times as many calls as the version employing the optimized shuffle.

The deterministic shuffle had its most dramatic impact on longer sentences and on sentences con-

[5]The absolute time values are not very significant, because the ConTroll system is currently implemented as an interpreter running in Prolog. However, the relative time differences between sentences confirm that the number of calls roughly reflects the total work required by the parser.

Sent.	Parses	*Nonoptimized* Time(sec)	ChoicePts	Calls	*Optimized* Time(sec)	ChoicePts	Calls	Call Factor
T1	1	5.6	61	359	1.8	20	88	4.1
T2	1	10.0	80	480	3.6	29	131	3.7
T3	1	24.3	199	1362	4.9	44	200	6.8
T4	1	25.0	199	1377	5.2	45	211	6.5
T5	1	51.4	299	2757	6.2	49	241	11.4
T6	2	463.5	2308	22972	32.4	209	974	23.6
T7	2	465.1	2308	23080	26.6	172	815	28.3
T8	1	305.7	1301	9622	52.1	228	942	10.2
T9	1	270.5	1187	7201	48.0	214	1024	7.0
T10	1	2063.4	6916	44602	253.8	859	4176	10.7
T11	1	3368.9	8833	74703	176.5	536	2565	29.1
T12	1	8355.0	19235	129513	528.1	1182	4937	26.2

Table 1: Comparison of Results for Selected Sentences

T1. Er hilft ihr.
T2. Hilft er seiner Freundin?
T3. Er hilft ihr schnell.
T4. Hilft er ihr schnell?
T5. Liess er ihr ihn helfen?
T6. Er liess ihn ihr schnell helfen.
T7. Liess er ihn ihr schnell helfen?
T8. Der Vater liess seiner Freundin seinen Sohn helfen.
T9. Sie denkt dass er ihr hilft.
T10. Sie denkt dass er ihr schnell hilft.
T11. Sie denkt dass er ihr ihn helfen liess.
T12. Sie denkt dass er seiner Freundin seinen Sohn helfen liess.

Table 2: Selected Sentences

taining adjuncts. For instance, in T7, a verb-first sentence containing the adjunct *schnell*, the optimized version outperformed the nonoptimized by a call factor of 28.3. From these results, the utility of a deterministic shuffle constraint is clear. In particular, it should be noted that avoiding useless results for shuffle constraints prunes away many large branches from the overall search space of the parser, because shuffle constraints are imposed on each node of the hierarchical structure. Since we use a largely bottom-up strategy, this means that if there are n solutions to a shuffle constraint on some daughter node, then all of the constraints on its mother node have to be solved n times. If we avoid producing $n-1$ useless solutions to shuffle, then we also avoid $n-1$ attempts to construct all of the ancestors to this node in the hierarchical structure.

7 Conclusion

We have shown that eliminating the nondeterminism of shuffle constraints overcomes one of the primary inefficiencies of parsing for grammars that use discontinuous order domains. Although bitstring codes have been used before in parsers for discontinuous constituents, we are not aware of any prior research that has demonstrated the use of this technique to eliminate the nondeterminism of relational constraints on word order. Additionally, we expect that the applicability of bitstring codes is not limited to shuffle contraints, and that the technique could be straightforwardly generalized for other nonconcatenative constraints. In fact, some way of recording the input positions associated with each constituent is necessary to eliminate spurious ambiguities that arise when the input sentence contains more than one occurrence of the same word (cf. van Noord's (1994) discussion of nonminimality). For concatenative grammars, each position can be represented by a simple remainder of the input list, but a more general encoding, such as the bitstrings used here, is needed for grammars using nonconcatenative constraints.

References

Emmon Bach. 1979. Control in montague grammar. *Linguistic Inquiry*, 10:515–553.

David R. Dowty. 1996. Toward a minimalist theory of syntactic structure. In Arthur Horck and Wietske Sijtsma, editors, *Discontinuous Constituency*, Berlin. Mouton de Gruyter.

Thilo Götz and Walt Detmar Meurers. 1997. The ConTroll system as large grammar development platform. In *Proceedings of the Workshop on Computational Environments for Grammar*

Development and Linguistic Engineering (ENVGRAM) held at ACL-97, Madrid, Spain.

Mark Johnson. 1985. Parsing with discontinuous constituents. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 127–132, Chicago, IL, July.

Andreas Kathol. 1995. *Linearization-based German Syntax.* Ph.D. thesis, The Ohio State University.

Martin Kay. 1989. Head-driven parsing. In *Proceedings of the First International Workshop on Parsing Technologies.* Carnegie Mellon University.

Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. 1983. BUP: a bottom up parser embedded in prolog. *New Generation Computing*, 1(2).

Michael Moortgat. 1996. Generalized quantifiers and discontinuous type constructors. In Arthur Horck and Wietske Sijtsma, editors, *Discontinuous Constituency*, Berlin. Mouton de Gruyter.

Fernando C.N. Pereira and Stuart M. Shieber. 1987. *Prolog and Natural Language Analysis.* CSLI Lecture Notes Number 10, Stanford, CA.

Carl Pollard. 1984. *Generalized Phrase Structure Grammars, Head Grammars and Natural Language.* Ph.D. thesis, Stanford University.

Michael Reape. 1991. Parsing bounded discontinuous constituents: Generalizations of some common algorithms. In *Proceedings of the First Computational Linguistics in the Netherlands Day*, OTK, University of Utrecht.

Mike Reape. 1996. Getting things in order. In Arthur Horck and Wietske Sijtsma, editors, *Discontinuous Constituents.* Mouton de Gruyter, Berlin.

D.J. Rosenkrantz and P.M. Lewis-II. 1970. Deterministic left corner parsing. In *IEEE Conference of the 11th Annual Symposium on Switching and Automata Theory*, pages 139–152.

Gertjan van Noord. 1991. Head corner parsing for discontinuous constituency. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pages 114–121, Berkeley, CA, June.

Gertjan van Noord. 1994. Head corner parsing. In C.J. Rupp, M.A. Rosner, and R.L. Johnson, editors, *Constraints, Language and Computation*, pages 315–338. Academic Press.

Gertjan van Noord. 1997. An efficient implementation of the head-corner parser. *Computational Linguistics*, 23(3):425–456.