Head Corner Parsing for Discontinuous Constituency

Gertjan van Noord

Lehrstuhl für Computerlinguistik
Universität des Saarlandes
Im Stadtwald 15
D-6600 Saarbrücken 11, FRG
vannoord@coli.uni-sb.de

Abstract

I describe a head-driven parser for a class of grammars that handle discontinuous constituency by a richer notion of string combination than ordinary concatenation. The parser is a generalization of the left-corner parser (Matsumoto *et al.*, 1983) and can be used for grammars written in powerful formalisms such as non-concatenative versions of HPSG (Pollard, 1984; Reape, 1989).

1 Introduction

Although most formalisms in computational linguistics assume that phrases are built by string concatenation (eg. as in PATR II, GPSG, LFG and most versions of Categorial Grammar), this assumption is challenged in non-concatenative grammatical formalisms. In Pollard's dissertation several versions of "head wrapping" are defined (Pollard, 1984). In the analysis of the Australian free word-order language Guugu Yimidhirr, Mark Johnson uses a 'combine' predicate in a DCG-like grammar that corresponds to the union of words (Johnson, 1985).

Mike Reape uses an operation called 'sequence union' to analyse Germanic semi-free word order constructions (Reape, 1989; Reape, 1990a). Other examples include Tree Adjoining Grammars (Joshi *et al.*, 1975; Vijay-Shankar and Joshi, 1988), and versions of Categorial Grammar (Dowty, 1990) and references cited there.

Motivation. There are several motivations for non-concatenative grammars. First, specialized string combination operations allow elegant linguistic accounts of phenomena that are otherwise notoriously hard. Examples are the analyses of Dutch cross serial dependencies by head wrapping or sequence union (Reape, 1990a).

Furthermore, in non-concatenative grammars it is possible to relate (parts of) constituents that belong together semantically, but which are not adjacent. Hence such grammars facilitate a simple compositional semantics. In CF-based grammars such phenomena usually are treated by complex 'threading' mechanisms.

Non-concatenative grammatical formalisms may also be attractive from a computational point of view. It is easier to define generation algorithms if the semantics is built in a systematically constrained way (van Noord, 1990b). The semantic-head-driven generation strategy (van Noord, 1989; Calder *et al.*, 1989; Shieber *et al.*, 1989; van Noord, 1990a; Shieber *et al.*, 1990) faces problems in case semantic heads are 'displaced', and this displacement is analyzed using threading. However, in this paper I sketch a simple analysis of verb-second (an example of a displacement of semantic heads) by an operation similar to head wrapping which a head-driven generator processes without any problems (or extensions) at all. Clearly, there are also some computational problems, because most 'standard' parsing strategies assume context-free concatenation of strings. These problems are the subject of this paper.

The task. I will restrict the attention to a class of constraint-based formalisms, in which operations on strings are defined that are more powerful than concatenation, but which operations are restricted to be *nonerasing*, and *linear*. The resulting class of systems can be characterized as Linear Context-Free Rewriting Systems

(LCFRS), augmented with feature-structures (F-LCFRS). For a discussion of the properties of LCFRS without feature-structures, see (Vijay-Shanker *et al.*, 1987). Note though that these properties do not carry over to the current system, because of the augmention with feature structures.

As in LCFRS, the operations on strings in F-LCFRS can be characterized as follows. First, derived structures will be mapped onto a set of occurances of words; i.e. each derived structure 'knows' which words it 'dominates'. For example, each derived feature structure may contain an attribute 'phon' whose value is a list of atoms representing the string it dominates. I will write $w(F)$ for the set of occurances of words that the derived structure F dominates. Rules combine structures $D_1 \ldots D_n$ into a new structure M. Nonerasure requires that the union of w applied to each daughter is a subset of $w(M)$:

$$\bigcup_{i=1}^{n} w(D_i) \subseteq w(M)$$

Linearity requires that the difference of the cardinalities of these sets is a constant factor; i.e. a rule may only introduce a fixed number of words syncategorematically:

$$|w(M)| - |\bigcup_{i=1}^{n} w(D_i)| = c, c \ a \ constant$$

CF-based formalisms clearly fulfill this requirement, as do Head Grammars, grammars using sequence union, and TAG's. I assume in the remainder of this paper that $\bigcup_{i=1}^{n} w(D_i) = w(M)$, for all rules other than lexical entries (i.e. all words are introduced on a terminal). Note though that a simple generalization of the algorithm presented below handles the general case (along the lines of Shieber et al. (1989; 1990)by treating rules that introduce extra lexical material as non-chain-rules).

Furthermore, I will assume that each rule has a designated daughter, called the head. Although I will not impose any restrictions on the head, it will turn out that the parsing strategy to be proposed will be very sensitive to the choice of heads, with the effect that F-LCFRS's in which the notion 'head' is defined in a systematic way (Pollard's Head Grammars, Reape's version of HPSG, Dowty's version of Categorial Grammar), may be much more efficiently parsed than other grammars. The notion *seed* of a parse tree is defined recursively in terms of the head. The seed of a tree will be the seed of its head. The seed of a terminal will be that terminal itself.

Other approaches. In (Proudian and Pollard, 1985) a head-driven algorithm based on active chart parsing is described. The details of the algorithm are unclear from the paper which makes a comparison with our approach hard; it is not clear whether the parser indeed allows for example the head-wrapping operations of Pollard (1984). Reape presented two algorithms (Reape, 1990b) which are generalizations of a shift-reduce parser, and the CKY algorithm, for the same class of grammars. I present a head-driven bottom-up algorithm for F-LCFR grammars. The algorithm resembles the head-driven parser by Martin Kay (Kay, 1989), but is generalized in order to be used for this larger class of grammars. The disadvantages Kay noted for his parser do not carry over to this generalized version, as redundant search paths for CF-based grammars turn out to be genuine parts of the search space for F-LCFR grammars.

The advantage of my algorithm is that it both employs bottom-up and top-down filtering in a straightforward way. The algorithm is closely related to head-driven generators (van Noord, 1989; Calder *et al.*, 1989; Shieber *et al.*, 1989; van Noord, 1990a; Shieber *et al.*, 1990). The algorithm proceeds in a bottom-up, head-driven fashion. In modern linguistic theories very much information is defined in lexical entries, whereas rules are reduced to very general (and very uninformative) schemata. More information usually implies less search space, hence it is sensible to parse bottom-up in order to obtain useful information as soon as possible. Furthermore, in many linguistic theories a special daughter called the head determines what kind of other daughters there may be. Therefore, it is also sensible to start with the head in order to know for what else you have to look for. As the parser proceeds from head to head it is furthermore possible to use powerful top-down predictions based on the usual head feature percolations. Finally note that proceding bottom-up solves some non-termination problems, because in lexicalized theories it is often the case that information in lexical entries limit the recursive application of rules (eg. the size of the subcat list of an entry determines the depth of the derivation tree of which this entry can be the seed).

Before I present the parser in section 3, I will first present an example of a F-LCFR grammar, to obtain a flavor of the type of problems the parser handles reasonably well.

2 A sample grammar

In this section I present a simple F-LCFR grammar for a (tiny) fragment of Dutch. As a caveat I want to stress that the purpose of the current section is to provide an *example* of possible input for the parser to be defined in the next section, rather than to provide an account of phenomena that is completely satisfactory from a linguistic point of view.

Grammar rules are written as (pure) Prolog clauses.[1] Heads select arguments using a subcat list. Argument structures are specified lexically and are percolated from head to head. Syntactic features are shared between heads (hence I make the simplifying assumption that head = functor, which may have to be revised in order to treat modification). In this grammar I use revised versions of Pollard's head wrapping operations to analyse *cross serial dependency* and *verb second* constructions. For a linguistic background of these constructions and analyses, cf. Evers (1975), Koster (1975) and many others.

Rules are defined as

```
rule(Head,Mother,Other)
```

or as

```
rule(Mother)
```

(for lexical entries), where **Head** represents the designated head daughter, **Mother** the mother category and **Other** a list of the other daughters. Each category is a term

```
x(Syn,Subcat,Phon,Sem,Rule)
```

where **Syn** describes the part of speech, **Subcat** is a list of categories a category subcategorizes for, **Phon** describes the string that is dominated by this category, and **Sem** is the argument structure associated with this category. **Rule** indicates which rule (i.e. version of the combine predicate **cb** to be defined below) should be applied; it generalizes the 'Order' feature of UCG. The value of **Phon** is a term **p(Left,Head,Right)** where the fields in this term are difference lists of words. The first argument represents the string left of the head, the second argument represents the head and the third argument represents the string right of the head. Hence, the string associated with such a term is the concatenation of the three arguments from left to right. There is only one parameterized, binary branching, rule in the grammar:

```
rule(x(Syn,[x(C,L,P2,S,R)|L],P1,Sem,_),
     x(Syn,L,P,Sem,_),
     [x(C,L,P2,S,R)]) :-
          cb(R, P1, P2, P).
```

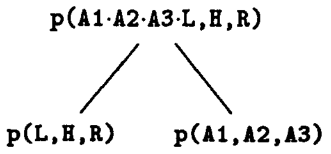
In this rule the first element of the subcategorization list of the head is selected as the (only) other daughter of the mother of the rule. The syntactic and semantic features of the mother and the head are shared. Furthermore, the strings associated with the two daughters of the rule are to be combined by the **cb** predicate. For simple (left or right) concatenation this predicate is defined as follows:

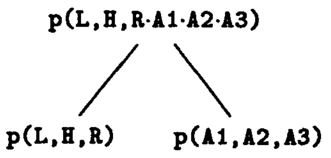
```
cb(left,  p(L4-L,H,R),
          p(L1-L2,L2-L3,L3-L4),
          p(L1-L,H,R)).

cb(right, p(L,H,R1-R2),
          p(R2-R3,R3-R4,R4-R),
          p(L,H,R1-R)).
```

Although this looks horrible for people not familiar with Prolog, the idea is really very simple. In the first case the string associated with the argument is appended to the left of the string left of the head; in the second case this string is appended to the right of the string right of the head. In a friendlier notation the examples may look like:

[1] It should be stressed though that other unification grammar formalisms can be extended quite easily to encode the same grammar. I implemented the algorithm for several grammars written in a version of PATR II without built-in string concatenation.





Lexical entries for the intransitive verb 'slaapt' (sleeps) and the transitive verb 'kust' (kisses) are defined as follows:

```
rule( x(v,[x(n,[],_,A,left)],
        p(P-P,[slaapt|T]-T,R-R),
        sleep(A),_)).

rule( x(v,[x(n,[],_,B,left),
           x(n,[],_,A,left)],
        p(P-P,[kust|T]-T,R-R),
        kiss(A,B),_)).
```

Proper nouns are defined as:

```
rule( x(n,[],p(P-P,[piet|T]-T,R-R),
        pete,_)).
```

and a top category is defined as follows (complementizers that have selected all arguments, i.e. sentences):

```
top(x(comp,[],_,_,_)).
```

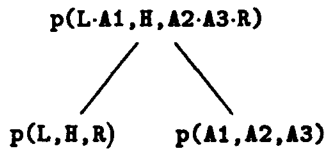
Such a complementizer, eg. 'dat' (that) is defined as:

```
rule( x(comp,[x(v,[],_,A,right)],
        p(P-P,[dat|T]-T,R-R),
        that(A),_)).
```

The choice of datastructure for the value of Phon allows a simple definition of the verb raising (vr) version of the combine predicate that may be used for Dutch cross serial dependencies:

```
cb(vr, p(L1-L2,H,R3-R),
       p(L2-L,R1-R2,R2-R3),
       p(L1-L,H,R1-R)).
```

Here the head and right string of the argument are appended to the right, whereas the left string of the argument is appended to the left. Again, an illustration might help:



A raising verb, eg. 'ziet' (sees) is defined as:

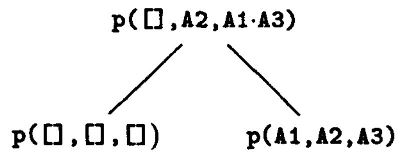
```
rule(x(v,[x(n,[],_,InfSubj,left),
          x(inf,[x(_,_,_,InfSubj,_)
                ],_,B,vr),
          x(n,[],_,A,left)],
       p(P-P,[ziet|T]-T,R-R),
       see(A,B),_)).
```

In this entry 'ziet' selects — apart from its **np**-subject — two objects, a **np** and a VP (with category **inf**). The **inf** still has an element in its subcat list; this element is controlled by the **np** (this is performed by the sharing of **InfSubj**). To derive the subordinate phrase 'dat jan piet marie ziet kussen' (that john sees pete kiss mary), the main verb 'ziet' first selects its **np**-object 'piet' resulting in the string 'piet ziet'. Then it selects the infinitival 'marie kussen'. These two strings are combined into 'piet marie ziet kussen' (using the **vr** version of the **cb** predicate). The subject is selected resulting in the string 'jan piet marie ziet kussen'. This string is selected by the complementizer, resulting in 'dat jan piet marie ziet kussen'. The argument structure will be instantiated as **that(sees(john,kiss(pete,mary)))**.

In Dutch main clauses, there usually is no overt complementizer; instead the finite verb occupies the first position (in yes-no questions), or the second position (right after the topic; ordinary declarative sentences). In the following analysis an empty complementizer selects an ordinary (finite) **v**; the resulting string is formed by the following definition of **cb**:

```
cb(v2, p(A-A,B-B,C-C),
       p(R1-R2,H,R2-R),
       p(A-A,H,R1-R)).
```

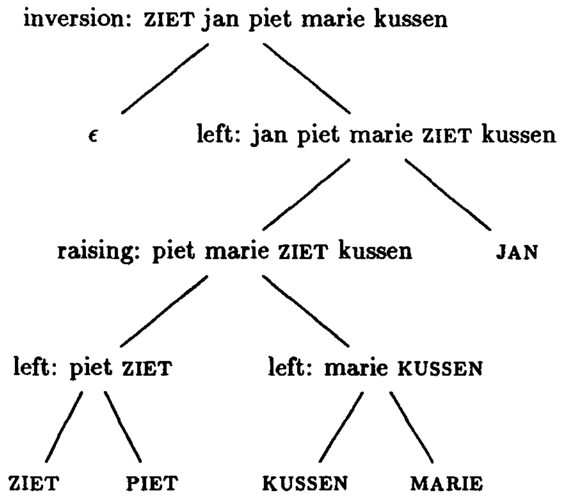
which may be illustrated with:



The finite complementizer is defined as:

```
rule(x(comp,[x(v,□,_,A,v2)],
        p(B-B,C-C,D-D),
        that(A),_)).
```

Note that this analysis captures the special relationship between complementizers and (fronted) finite verbs in Dutch. The sentence 'ziet jan piet marie kussen' is derived as follows (where the head of a string is represented in capitals):


inversion: ZIET jan piet marie kussen
ε
left: jan piet marie ZIET kussen
raising: piet marie ZIET kussen
JAN
left: piet ZIET
left: marie KUSSEN
ZIET
PIET
KUSSEN
MARIE

3 The head corner parser

This section describes the head-driven parsing algorithm for the type of grammars described above. The parser is a generalization of a left-corner parser. Such a parser, which may be called a 'head-corner' parser,[2] proceeds in a bottom-up way. Because the parser proceeds from head to head it is easy to use powerful top-down predictions based on the usual head feature percolations, and subcategorization requirements that heads require from their arguments.

In left-corner parsers (Matsumoto *et al.*, 1983) the first step of the algorithm is to select the leftmost word of a phrase. The parser then proceeds by proving that this word indeed can be the left-corner of the phrase. It does so by selecting a rule whose leftmost daughter unifies with the category of the word. It then parses other daughters of the rule recursively and then continues by connecting the mother category of that rule upwards, recursively. The left-corner algorithm can be generalized to the class of grammars under consideration if we start with the *seed* of a phrase, instead of its leftmost word. Furthermore the **connect** predicate then connects smaller categories upwards by unifying them with the *head* of a rule. The first step of the algorithm consists of the prediction step: which lexical entry is the seed of the phrase? The first thing to note is that the words introduced by this lexical entry should be part of the input string, because of the nonerasure requirement (we use the string as a 'guide' (Dymetman *et al.*, 1990) as in a left-corner parser, but we change the way in which lexical entries 'consume the guide'). Furthermore in most linguistic theories it is assumed that certain features are shared between the mother and the head. I assume that the predicate **head/2** defines these feature percolations; for the grammar of the foregoing section this predicate may be defined as:

```
head(x(Syn,_,_,Sem,_),
     x(Syn,_,_,Sem,_)).
```

As we will proceed from head to head these features will also be shared between the seed and the top-goal; hence we can use this definition to restrict lexical lookup by top-down prediction. [3] The first step in the algorithm is defined as:

```
parse(Cat,P0,P) :-
    predict_lex(Cat,SmallCat,P0,P1),
    connect(SmallCat,Cat,P1,P).

predict_lex(Cat,SmallCat,P0,P) :-
    head(Cat,SmallCat),
    rule(SmallCat),
    string(SmallCat,Words),
    subset(Words,P0,P).
```

Instead of taking the first word from the current input string, the parser may select a lexical en-

[2]This name is due to Pete Whitelock.

[3]In the general case we need to compute the transitive closure of (restrictions of) possible mother-head relationships. The predicate 'head may also be used to compile rules into the format adopted here (i.e. using the definition the compiler will identify the head of a rule).

try dominating a subset of the words occuring in the input string, provided this lexical entry can be the *seed* of the current goal. The predicate **subset(L1,L2,L3)** is true in case **L1** is a subset of **L2** with complement **L3**.[4]

The second step of the algorithm, the connect part, is identical to the connect part of the left-corner parser, but instead of selecting the left-most daughter of a rule the head-corner parser selects the head of a rule:

```
connect(X,X,P,P).
connect(Small,Big,P0,P) :-
    rule(Small, Mid, Others),
    parse_rest(Others,P0,P1),
    connect(Mid,Big,P1,P).

parse_rest([],P,P).
parse_rest([H|T],P0,P) :-
    parse(H,P0,P1),
    parse_rest(T,P1,P).
```

The predicate '**start_parse**' starts the parse process, and requires furthermore that the string associated with the category that has been found spans the input string *in the right order.*

```
start_parse(String,Cat):-
    top(Cat),
    parse(Cat,String,[]),
    string(Cat,String).
```

The definition of the predicate '**string**' depends on the way strings are encoded in the grammar. The predicate relates linguistic objects and the string they dominate (as a list of words). I assume that each grammar provides a definition of this predicate. In the current grammar **string/2** is defined as follows:

[4]In Prolog this predicate may be defined as follows:

```
subset([],P,P).
subset([H|T],P0,P):-
  selectchk(H,P0,P1),
  subset(T,P1,P).

selectchk(El,[El|P],P):-
  !.
selectchk(El,[H|P0],[H|P]):-
  selectchk(El,P0,P).
```

The cut in selectchk is necessary in case the same word occurs twice in the input string; without it the parser would not be 'minimal'; this could be changed by indexing words w.r.t. their position, but I will not assume this complication here.

```
string(x(_,_,Phon,_,_),Str):-
    copy_term(Phon,Phon2),
    str(Phon2,Str).

str(p(P-P1,P1-P2,P2-[]),P).
```

This predicate is complicated using the predicate **copy_term/2** to prevent any side-effects to happen in the category. The parser thus needs two grammar specific predicates: **head/2** and **string/2**.

Example. To parse the sentence 'dat jan slaapt', the head corner parser will proceed as follows. The first call to 'parse' will look like:

```
parse(x(comp,[],_,_,_),
      [dat,jan,slaapt],[])
```

The prediction step selects the lexical entry 'dat'. The next goal is to show that this lexical entry is the seed of the top goal; furthermore the string that still has to be covered is now **[jan,slaapt]**. Leaving details out the connect clause looks as :

```
connect(
    x(comp,[x(v,..,right)],..),
    x(comp,[],..),[jan,slaapt],[])
```

The category of **dat** has to be matched with the head of a rule. Notice that **dat** subcategorizes for a **v** with rule feature **right**. Hence the **right** version of the **cb** predicate applies, and the next goal is to parse the **v** for which this complementizer subcategorizes, with input 'jan, slaapt'. Lexical lookup selects the word **slaapt** from this string. The word **slaapt** has to be shown to be the head of this **v** node, by the **connect** predicate. This time the **left** combination rule applies and the next goal consists in parsing a **np** (for which **slaapt** subcategorizes) with input string **jan**. This goal succeeds with an empty output string. Hence the argument of the rule has been found successfully and hence we need to connect the mother of the rule up to the **v** node. This succeeds trivially, and therefore we now have found the **v** for which **dat** subcategorizes. Hence the next goal is to connect the complementizer with an empty subcat list up to the topgoal; again this succeeds trivially. Hence we obtain the instantiated version of the parse call:

```
parse(x(comp,[],p(P-P,[dat|T]-T,
          [jan,slaapt|Q]-Q),
          that(sleeps(john)),_),
          [dat,jan,slaapt],[])
```

and the predicate **start_parse** will succeed, yielding:

```
Cat = x(comp,[],p(P-P,[dat|T]-T,
        [jan,slaapt|Q]-Q),
        that(sleeps(john)),_)
```

4 Discussion and Extensions

Sound and Complete. The algorithm as it is defined is *sound* (assuming the Prolog interpreter is sound), and *complete* in the usual Prolog sense. Clearly the parser may enter an infinite loop (in case non branching rules are defined that may feed themselves or in case a grammar makes a heavy use of empty categories). However, in case the parser *does* terminate one can be sure that it has found all solutions. Furthermore the parser is *minimal* in the sense that it will return one solution for each possible derivation (of course if several derivations yield identical results the parser will return this result as often as there are derivations for it).

Efficiency. The parser turns out to be quite efficient in practice. There is one parameter that influences efficiency quite dramatically. If the notion 'syntactic head' implies that much syntactic information is shared between the head of a phrase and its mother, then the prediction step in the algorithm will be much better at 'predicting' the head of the phrase. If on the other hand the notion 'head' does not imply such feature percolations, then the parser must predict the head randomly from the input string as no top-down information is available.

Improvements. The efficiency of the parser can be improved by common Prolog and parsing techniques. Firstly, it is possible to compile the grammar rules, lexical entries and parser a bit further by (un)folding (eg. the string predicate can be applied to each lexical entry in a compilation stage). Secondly it is possible to integrate well-formed and non-well-formed subgoal tables in the parser, following the technique described by Matsumoto et al. (1983). The usefulness of this technique strongly depends on the actual grammars that are being used. Finally, the current indexing of lexical entries is very bad indeed and can easily be improved drastically.

In some grammars the string operations that are defined are not only monotonic with respect to the words they dominate, but also with respect to the order constraints that are defined between these words ('order-monotonic'). For example in Reape's sequence union operation the linear precedence constraints that are defined between elements of a daughter are by definition part of the linear precedence constraints of the mother. Note though that the analysis of verb second in the foregoing section uses a string operation that does not satisfy this restriction. For grammars that do satisfy this restriction it is possible to extend the top-down prediction possibilities by the incorporation of an extra clause in the 'connect' predicate which will check that the phrase that has been analysed up to that point can become a substring of the top string.

Acknowledgements

This research was partly supported by SFB 314, Project N3 BiLD; and by the NBBI via the Eurotra project.

I am grateful to Mike Reape for useful comments, and an anonymous reviewer of ACL, for pointing out the relevance of LCFRS.

Bibliography

Jonathan Calder, Mike Reape, and Henk Zeevat. An algorithm for generation in unification categorial grammar. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics*, pages 233–240, Manchester, 1989.

David Dowty. Towards a minimalist theory of syntactic structure. In *Proceedings of the Symposium on Discontinuous Constituency*, ITK Tilburg, 1990.

Marc Dymetman, Pierre Isabelle, and Francois Perrault. A symmetrical approach to parsing and generation. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Helsinki, 1990.

Arnold Evers. *The Transformational Cycle in Dutch and German*. PhD thesis, Rijksuniversiteit Utrecht, 1975.

Mark Johnson. Parsing with discontinuous constituents. In *23th Annual Meeting of the Association for Computational Linguistics*, Chicago, 1985.

A.K. Joshi, L.S. Levy, and M. Takahashi. Tree adjunct grammars. *Journal Computer Systems Science*, 10(1), 1975.

Martin Kay. Head driven parsing. In *Proceedings of Workshop on Parsing Technologies*, Pittsburgh, 1989.

Jan Koster. Dutch as an SOV language. *Linguistic Analysis*, 1, 1975.

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

Carl Pollard. *Generalized Context-Free Grammars, Head Grammars, and Natural Language*. PhD thesis, Stanford, 1984.

C. Proudian and C. Pollard. Parsing head-driven phrase structure grammar. In *23th Annual Meeting of the Association for Computational Linguistics*, Chicago, 1985.

Mike Reape. A logical treatment of semi-free word order and bounded discontinuous constituency. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics*, UMIST Manchester, 1989.

Mike Reape. Getting things in order. In *Proceedings of the Symposium on Discontinuous Constituency*, ITK Tilburg, 1990.

Mike Reape. Parsing bounded discontinous constituents: Generalisations of the shift-reduce and CKY algorithms, 1990. Paper presented at the first CLIN meeting, October 26, OTS Utrecht.

Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira. A semantic-head-driven generation algorithm for unification based formalisms. In *27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, 1989.

Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira. Semantic-head-driven generation. *Computational Linguistics*, 16(1), 1990.

Gertjan van Noord. BUG: A directed bottom-up generator for unification based formalisms. *Working Papers in Natural Language Processing, Katholieke Universiteit Leuven, Stichting Taaltechnologie Utrecht*, 4, 1989.

Gertjan van Noord. An overview of head-driven bottom-up generation. In Robert Dale, Chris Mellish, and Michael Zock, editors, *Current Research in Natural Language Generation*. Academic Press, 1990.

Gertjan van Noord. Reversible unification-based machine translation. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Helsinki, 1990.

K. Vijay-Shankar and A. Joshi. Feature structure based tree adjoining grammar. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING)*, Budapest, 1988.

K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Annual Meeting of the Association for Computational Linguistics*, Stanford, 1987.