Learning Linear Precedence Rules

Vladimir Pericliev
Mathematical Linguistics Department
Institute of Mathematics and Computer Science, bl.8
Bulgarian Academy of Sciences, 1113 Sofia—Bulgaria
peri@bgearn.acad.bg

Abstract

A system is described which learns from examples the Linear Precedence rules in an Immediate Dominance/Linear Precedence grammar. Given a particular Immediate Dominance grammar and hierarchies of feature values potentially relevant for linearization (=the system's bias), the learner generates appropriate natural language expressions to be evaluated as positive or negative by a teacher, and produces as output Linear Precedence rules which can be directly used by the grammar.

1 Introduction

The manual compilation of a sizable grammar is a difficult and time-consuming task. An important subtask is the construction of word ordering rules in the grammar. Though some languages are proclaimed as having simple ordering rules, e.g. either complete scrambling or strictly "fixed" order, most languages exhibit quite complex regularities (Steele, 1981), and even the rigid word order languages (like English) and those with total scrambling (like Warlpiri; cf. (Hale, 1983) may show intricate rules (Kashket, 1981); hence the need for their automatic acquisition. This task however, to the best of our knowledge, has not been previously addressed.

This paper describes a program which, given a grammar with no ordering relations, produces as output a set of linearization, or Linear Precedence, rules which can be directly employed by that grammar. The learning step uses the version space algorithm, a familiar technique from machine learning for learning from examples. In contrast to most previous uses of the algorithm for various learning tasks, which rely on priorly given classified examples, our learner generates itself the training instances one at a time, and they are then classed as positive or negative by a teacher. A selective generation of training instances is employed which facilitates the learning by minimizing the number of evaluations that the teacher needs to make.

The next section describes the Immediate Dominance/Linear Precedence grammar format. In section 3, the task of learning Linear Precedence rules is interpreted as a task of learning from examples, and section 4 introduces the version space method. Section 5 is a system overview, and section 6 focuses on implementation. Finally, some limitations of the system are discussed as well as some directions for future research.

2 Immediate Dominance/Linear Precedence Grammars

A standard way of expressing the ordering of nodes in a grammar is by means of Linear Precedence rules in Immediate Dominance/Linear Precedence (ID/LP) grammars. The ID/LP format was first introduced by (Gazdar and Pullum, 1981) and (Gazdar et. al., 1985) and is usually associated with GPSG, but is also used by HPSG (Pollard and Sag, 1987) and, under different guises, by other formalisms as well.

In an ID/LP grammar, the two types of information, constituency (or, immediate dominance) and linear order, are separated. Thus, for instance, an immediate dominance rule, say, $A \rightarrow B\ C\ D$, with no Linear Precedence rules declared, stands for the mother node A expanded into its siblings occurring in any order (six Context Free Grammar rules as result of the permutations). If the LP rule $D < C$ is added, the ID rule can be expanded in the following three CFG rules: $A \rightarrow B\ D\ C$; $A \rightarrow D\ B\ C$; $A \rightarrow D\ C\ B$. ID/LP grammars capture an important ordering generalization, missed by usual CFGs, by means of the so called "Exhaustive Partial Ordering Constraint", stating that the partial ordering of any two sister nodes is constant throughout the whole grammar. That is, just one of the following three situations is valid for the ordering of any two nodes A and B: either $A < B$ (A precedes B) or $A > B$ (A follows B) or $A <> B$ (A occurs in either position with respect to B). (The last $<>$ situation is normally stated in ID/LP grammars by *not* stating an

LP rule, but we shall use it here, as we need an explicit reference to it.)

3 The Task of LP Rules Acquisition Viewed As Learning from Examples

A program which learns from examples usually reasons from very specific, low-level, instances (positive or both positive and negative) to more general, high-level, rules that adequately describe those instances. Upon a common understanding (Lea and Simon, 1974), learning from examples is a cooperative search in two spaces, the *instance space*, i.e. the space of all possible training instances, and the *rule (=hypotheses) space*, i.e. the space of all possible general rules. Besides these two spaces, two additional processes are needed, intermediating them: *interpretation* and *instance selection*. The interpretation process is needed, in moving from the instance space to the rule space, to interpret the *raw* instances, which may be far removed in form from the form of the rules, so that instances can guide the search in the rule space. Analogously, the instance selection rules serve to transform the high-level hypotheses (rules) to a representation useful for guiding the search in the instance space.

A general description of our task is as follows: *Given a specific ID grammar with no LP rules, find those LP rules.* [1] In this task we also need to reason from very specific instances of LP rules (language phrases like *small children*, **children small*) to more general LP rules (adjective < noun), therefore it can be interpreted in terms of the two-space model, described above.

Our instance space will consist of all strings generable by the given ID grammar (the size of this instance space for any non-toy grammar will be very large). The LP rules space will be an unordered set, whose elements are pairs of nodes, connected by one of the relations <, > or <>, e.g. LP set = [[A < B], [B < E], [E > C], ...]. (The size of the LP rules space will depend upon the size of the specific ID grammar whose LP rules are to be learned.)

We also need to define the interpretation and instance-selection processes. In the learning system to be described, for both purposes serves (basically) a meta-interpreter for ID/LP grammars, which can parse the concrete grammar, given at the outset, for both analysis and generation. In an interpretation phase, the meta-interpreter will parse a natural language expression outputting an LP-rules-space approriate representation, whereas in the instance-selection phase the meta-interpreter, given an LP space representation as input, will generate a language expression to be classified as positive (i.e. not violating word order rules) or negative (i.e. violating those rules) by a teacher.

[1] Though indeed this is the usual way of looking at the task, sometimes we may need to start with some LP rules already known; the program we shall describe supports both regimes.

4 The Version Space Method

There are a variety of methods in the AI literature for learning from examples. For handling our task, we have chosen the so called "version space" method (also known as the "candidate elimination algorithm"), cf. (Mitchell, 1982). So we need to have a look at this method.

The basic idea is, that in all representation languages for the rule space, there is a partial ordering of expressions according to their *generality*. This fact allows a compact representation of the set of plausible rules (=hypotheses) in the rule space, since the set of points in a partially ordered set can be represented by its *most general* and its *most specific* elements. The set of most general rules is called the *G-set*, and the set of most specific rules the *S-set*.

Figure 1 illustrates the LP rules space of a determiner of some grammatical number (singular or plural) and an adjective, expressed in predicate logic.

Viewed top-down, the hierarchy is in descending order of generality (arrows point from specific to general). The topmost LP rule is most general and covers all the other rules, since det(Num), where *Num* is a variable, covers both det(sg) and det(pl), and <> covers both < and >. Each of the rules at level 2 are neither more general nor more specific than each other, but are more general than the most specific rules at the bottom.

The *learning method* assumes a set of positive and negative examples, and its aim is to induce a rule which covers all the positive examples and none of the counterexamples. The basic algorithm is as follows:

(1) The *G-set* is instantiated to the most general rule, and the *S-set* to the first positive example (i.e. a positive is needed to start the learning process).

(2) The next training instance is accepted. If it is *positive*, from the *G-set* are removed the rules which do not cover the example, and the elements of *S-set* are generalized as little as possible, so that they cover the new instance. If the next instance is *negative*, then from the *S-set* are removed the rules that cover the counterexample, and the elements of the *G-set* are specialized as little as possible so that the counterexample is no longer covered by any of the elements of the *G-set*.

(3) The learning process terminates when the *G-set* and the *S-set* are both singleton sets which are identical. If they are different singleton sets, the training instances were inconsistent. Otherwise a new training instance is accepted.

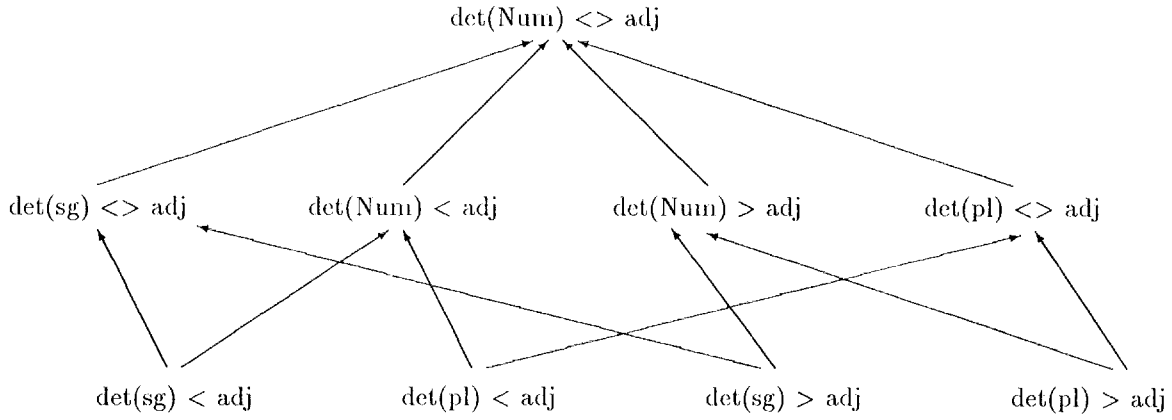


Figure 1: A Generalization hierarchy

Now, let us see how this works with the LP rules version space in Figure 1, assuming further the following classified examples ((+) means positive, and (−) negative instance):

(+) det(sg) < adj
(−) det(sg) > adj
(+) det(pl) < adj

The algorithm will instantiate the *G-set* to the most general rule in the version space, and the *S-set* to the first positive, obtaining:

G-set: [[det(Num) <> adj]]
S-set: [[det(sg) < adj]]

Then the next example will be accepted, which is negative. The current *S-set* does not cover it, so it remains the same; the *G-set* is specialized as little as possible to exclude the negative, which yields:

G-set: [[det(Num) < adj]]
S-set: [[det(sg) < adj]]

The last example is positive. The *G-set* remains the same since it covers the positive. The *S-set* however does not, so it has to be minimally generalized to cover it, obtaining:

G-set: [[det(Num) < adj]]
S-set: [[det(Num) < adj]]

These are singleton sets which are identical, and the resultant (consistent) generalization is therefore: [det(Num) < adj]. That is, a determiner of any grammatical number must precede an adjective.

5 Overview of the Learner

Our learning program has two basic modules: the version space learner which performs the elementary learning step (as described in the previous section), and a meta-interpreter for ID/LP grammars which serves the processes of interpretation and instance selection (as described in section 3). The learning proceeds in a dialog form with the teacher: for the learning of each individual LP rule, the system produces natural language phrases to be classified by the teacher until it can converge to a single concept (rule). The whole process ends when all LP rules are learned.

At the outset, the program is supplied with the specific ID grammar whose LP rules are to be acquired, and the user-provided *bias* of the system. The latter implies an explicit statement on the part of the user of what features and values are relevant to the task, by inputting the corresponding generalization hierarchies (the precedence generalization hierarchy is taken for granted).

In the particular implementation, the acceptable ID grammar format is essentially that of a logic grammar (Pereira and Warren, 1980), (Dahl and Abramson, 1990). We only use a double arrow (to avoid mixing up with the often built-in Definite Clause Grammar notation), and besides empty productions and sisters having the very same name are not allowed, since they interfere with LP rules statements, cf. e.g. (Sag, 1987), (Saint-Dizier, 1988).

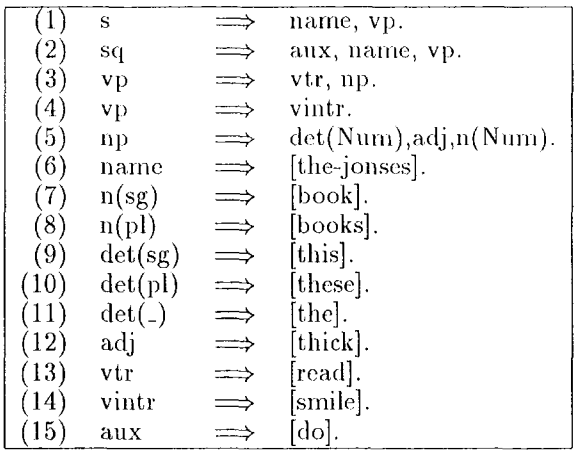
6 The Implementation

Below we discuss the basic aspects of the implementation, illustrating it with the ID grammar with no LP restrictions, given on Figure 2.

The grammar will generate simple declarative and interrogative sentences like *The Jonses read this thick book, The Jonses read these thick books, Do the Jonses smile*, etc. as well as all their (ungrammatical) permutations *Read this thick book the Jonses, The Jonses read thick this book do*, etc.

The program knows at the outset that the values "sg" and "pl" are both more specific than the variable "Num", matching any number (this is the bias of the system).

Step 1. The program determines the siblings



(1)	s	$\Longrightarrow$	name, vp.
(2)	sq	$\Longrightarrow$	aux, name, vp.
(3)	vp	$\Longrightarrow$	vtr, np.
(4)	vp	$\Longrightarrow$	vintr.
(5)	np	$\Longrightarrow$	det(Num),adj,n(Num).
(6)	name	$\Longrightarrow$	[the-jonses].
(7)	n(sg)	$\Longrightarrow$	[book].
(8)	n(pl)	$\Longrightarrow$	[books].
(9)	det(sg)	$\Longrightarrow$	[this].
(10)	det(pl)	$\Longrightarrow$	[these].
(11)	det(_)	$\Longrightarrow$	[the].
(12)	adj	$\Longrightarrow$	[thick].
(13)	vtr	$\Longrightarrow$	[read].
(14)	vintr	$\Longrightarrow$	[smile].
(15)	aux	$\Longrightarrow$	[do].

Figure 2: A simple ID grammar with no LP constraints

(=the right-hand sides of ID rules) that will later have to be linearized, by collecting them in a *partially ordered* list. Singleton right-hand sides (rule (4) above and all dictionary rules) are therefore left out, and so are cuts, and "escapes to Prolog" in curly brackets, since they are not used to represent tree nodes, but are rather constraints on such nodes. Also, if some right-hand side is a set which (properly) includes another right-hand side (as in rule (2) and rule (1) above), the latter is not added to the sibling list, since we do not want to learn twice the linearization of some two nodes ("name" and "vp" in our case). The sibling list then, after the hierarchical sorting from lower-level to higher-level nodes, becomes:

[[aux,name,vp],[vtr,np],[det(Num),adj,n(Num)]]

Now, despite the fact that the set of LP rules we need to learn is itself unordered, the order in which the program *learns* each individual LP rule may be very essential to the acquisition process. Thus, starting from the first element of the above sibling list, viz. [aux, name, vp], we will be in trouble when attempting to locate the misorderings in *any negative* example. Considering just a single negative instance, say *The Jonses read thick this book do* : What is(are) the misplacement(s) and where do they occur? In the higher-level tree nodes [aux, name, vp] or in the lower-level nodes [vtr, np] or in the still lower [det(Num),adj,n(Num)] ?

Our program solves this problem by exploiting the fact, peculiar to our application, that the nodes in a grammar are *hierarchically structured*, therefore we may try to linearize a set of nodes A and B higher up in a tree *only after all lower-level nodes dominated by both A and B have already been ordered.* Knowing these lower-level LP rules, our meta-interpreter would never generate instances like *The Jonses read thick this book do*, but only some repositionings of the nodes [aux, name, vp], their internal ordering being guaranteed to be correct. The sibling list then, after hierarchical sorting from lower-level to higher-level nodes, becomes:

[[det(Num),adj,n(Num)],[vtr,np],[aux,name,vp]]

and the first element of this list is first passed to the learning engine.

Step 2. The program now needs to produce a first positive example, as required by the version space method. Taking as input the first element of the sibling list, the ID/LP meta-interpreter generates a phrase conforming to this description and asks the teacher to re-order it correctly (if needed). In our case, the first positive example would be *this thick book*. The phrase will be reparsed in order to determine the linearization of constituents.

A word about the ID/LP parser/generator. Its analysis role is needed in processing the first positive example, and the generation role in the production of language examples for all intermediate stages of the learning process which are then evaluated by the teacher. The predicate observes two types of LP constraints: the globally valid LP rules that have been acquired by the system so far, [2] and the "transitory" LP constraints, serving to produce an ordering, as required by an intermediate stage of the learning process.

Disposing of the ordering of constituents in the positive example, the *transitive closure* of these partial orderings is computed (in our case, from [[det(Num) < adj],[adj < n(Num)]] we get [[det(Num) < adj], [adj < n(Num)], [det(Num) < n(Num)]]). This result is then cast into a representation that supports our learning process. [3]

[2] Or are priorly known, in the case when the system starts with some LP rules declared by the user.

[3] The concept we learn is actually a *conjunction* of individual LP rules, when the right-hand side of a rule consists of three or more constituents.

Dialog with user	*LP rules space (internal representation)*	*Actual LP rules*
`this thick book`	Ex.: [det,sg,<,adj,#,adj,<,n,_,#,det,_,<,n,_]	det(sg) < adj
(+)	G: [[det,N,<>,adj,#,adj,<>,n,_,#,det,_,<>,n,_]]	adj < n(sg)
	S: [[det,sg,<,adj,#,adj,<,n,_,#,det,_,<,n,_]]	det(sg) < n(sg)
(?)	Ex.: [det,sg,<,adj,#,adj,<,n,_,#,det,_,>,n,_]	det(sg) < adj
(−)	G: [[det,N,<>,adj,#,adj,<>,n,_,#,det,_,<,n,_]]	adj < n(sg)
	S: [[det,sg,<,adj,#,adj,<,n,_,#,det,_,<,n,_]]	det(sg) > n(sg)
`this book thick`	Ex.: [det,sg,<,adj,#,adj,>,n,_,#,det,_,<,n,_]	det(sg) < adj
(−)	G: [[det,N,<>,adj,#,adj,<,n,_,#,det,_,<,n,_]]	adj > n(sg)
	S: [[det,sg,<,adj,#,adj,<,n,_,#,det,_,<,n,_]]	det(sg) < n(sg)
`thick this book`	Ex.: [det,sg,>,adj,#,adj,<,n,_,#,det,_,<,n,_]	det(sg) > adj
(−)	G: [[det,N,<,adj,#,adj,<,n,_,#,det,_,<,n,_]]	adj < n(sg)
	S: [[det,sg,<,adj,#,adj,<,n,_,#,det,_,<,n,_]]	det(sg) < n(sg)
`these thick books`	Ex.: [det,pl,<,adj,#,adj,<,n,_,#,det,_,<,n,_]	det(pl) < adj
(+)	G: [[det,N,<,adj,#,adj,<,n,_,#,det,_,<,n,_]]	adj < n(pl)
	S: [[det,sg,<,adj,#,adj,<,n,_,#,det,_,<,n,_]]	det(pl) < n(pl)
	Consistent generalization:	det(N) < adj
	[det,N,<,adj,#,adj,<,n,_,#,det,_,<,n,_]	adj < n(N)
		det(N) < n(N)

Figure 3: A sample session

Step 3. The version space method is applied and the individual LP rules, resulting from finding a consistent generalization, are asserted in the ID/LP grammar database to be respected by any further generation process. [4]

Figure 3 gives a learning cycle starting from the sibling list element [det(Num),adj,n(Num)]. The first column gives the dialog with the teacher, the second the program's internal representation of the LP rules space, and the third those rules are expressed in their more familiar, and final, form that can be utilized directly by the ID grammar.

After processing the first positive (first row), the system generalizes by varying a parameter (number or precedence), verbalizes the generalization, the generated phrase is classified by the teacher, then another generalization is made, depending on the classification, it is verbalized, evaluated and so on. The process results in the three LP rules: det(Num) < adj; adj < n(Num); and det(Num) < n(Num).

A remark on notation: # delimits individual LP rules, allowing their recovery in terms of Prolog structures. The underbars, _, are merely placeholders for bound variables (in our case, those bound to "N"). Clearly, mutually dependent feature values need to be considered (i.e. varied by the program) only once, and so they occur just once in the expressions.

Several additional points regarding the learning process need to be made.

[4] Assertions are actually made after checking for consistency with LP's already present in the database. Though no contradictions may arise with *acquired* rules, they may come from LP's declared by the user in the case when the system is started with some such LP's.

The first is that after converging to a single LP rule, it is tested whether this rule covers *all* most specific instances. For doing this, the stated generalization hierarchies are taken into account alongside with the fact that in an ID/LP format a rule of the type $A > B$ logically implies the negation of its "inverse rule" $A < B$. Thus, the rule det(Num) < adj covers all potential most specific instances since the rule itself and its inverse rule det(Num) > adj cover them, which is clearly seen on the generalization hierarchy in Figure 1. If some most specific instances remain uncovered, then they are fed again to the version space algorithm for a second pass.

The second point is that when it is impossible for some structure to be verbalized due to contradictory LP statements (as in the second row), the system itself evaluates this example as negative and proceeds further.

We also need to emphasize that the program *selectively*, rather than randomly, varies the potentially relevant parameters (number and precedence, in this particular case), attempting to converge the generalization process most quickly. This is done in order to minimize the number of training instances that need to be generated, and hence to minimize the number of evaluations that the teacher needs to make. In other words, being generalization-driven, the generator never produces training instances which are superfluous to the generalization process. This, in particular, allows the program to avoid outputting all strings generable by the grammar whose LP rules are being acquired (notice, for instance, in the first column of Figure 3 that no language expression involving the dictionary rule `(11) det(_)` $\Longrightarrow$ `[the]` from Figure 2 is displayed to the user).

In this respect our approach is in sharp contrast to a learning process whose training examples are given *en bloc*, and hence the teacher would, of necessity, make a great lot of assessions that the learner would never use.

Step 4. The learning terminates successfully when all LP rules are found (i.e. all elements of the sibling list are processed) and fails when no consistent generalization may be found for some data. The latter fact needs to be interpreted in the sense that these data are not correctly describable within the ID/LP format.

7 Conclusion and Future Work

We have described a program that learns the LP rules of an ID/LP logic grammar in a form that can be directly utilized by that grammar. This task has not been addressed in previous work.

We conclude by mentioning some limitations of the system suggesting future directions for investigation.

It is known that the version space method misbehaves on encountering noisy data: an instance mistakenly classed as negative e.g. may lead to premature pruning of a search branch where the solution may actually lie. This may be a problem in our task (and perhaps in many other linguistic tasks) since our assessments of grammatical/ungrammatical word order are in some cases far from definite yes/no's. So handling uncertain input is one way our research may evolve.

Another direction for future research is addressing the learning of word order expressed in more complex formalisms than ID/LP grammars. It has been proposed in the (computational) linguistics literature (e.g. (Zwicky, 1986), Ojeda, 1988, Pericliev and Grigorov, 1994) that LP rules of the standard format may be insufficient in some cases, and need to be augmented with other ordering relations like "immediate precedence" <<, "fist", "last", etc., and more generally, that linearization needs to be stated in complex logic expressions connected by conjunction, disjunction and negation. We can trivially add the relation << to the present learner, but the other parts of such proposals seem beyond its immediate capacity, as it stands. From our previous work on word order we despose of a parser/generator that can handle complex expressions, however we shall need to modify (or perhaps, even replace) our learning method with one which is better suited to handle logic constructions like disjunction and negation.

Acknowledgements. The research reported in this paper was partly supported by contract I-526/95.

References

V. Dahl and H. Abramson. 1990. *Logic Grammars*, Springer.

G. Gazdar and G. Pullum. 1981. Subcategorization, constituent order and the notion of "head". In M. Moortgat et. al. (eds). *The Scope of Lexical Rules*, Dordrecht, Holland, pages 107-123.

G. Gazdar, E. Klein, G. Pullum and I. Sag. 1985. *Generalized Phrase Structure Grammar*. Harvard, Cambr. Mass.

K. Hale. 1983. Warlpiri and the grammar of non-configurational languages. *Natural Language and Linguistic Theory, v.1*, pages 5-49.

M. Kashket. 1987. A GB-based parser for Warlpiri, a free-word order language. MIT AI Laboratory.

G. Lea and H. Simon. 1974. Problem solving and rule induction: A unified view. In L. Gregg (ed). *Knowledge and Cognition*, Lawrence Erlbaum Associates.

T. Mitchell. 1982. Generalization as search. *Artificial Intelligence*, 18, pages 203-226.

A. Ojeda. 1988. A linear precedence account of cross-serial dependencies. *Linguistics aand Philosophy*, 11, pages 457-492.

F.C.N. Pereira and D.H.D Warren. 1980. Definite Clause Grammars for Natural Language Analysis. *Artificial Intelligence*, 13, pages 231-278.

V. Pericliev and A. Grigorov. 1994. Parsing a flexible word order language. *COLING'94*, Kyoto, pages 391-395.

C. Pollard and I. Sag. 1987. *Information-Based Syntax and Semantics*,vol.1: Fundamentals. CSLI Lecture Notes No. 13, Stanford, CA.

P. Saint-Dizier. 1988. Contextual discontinuous grammars. In *Natural Language Understanding and Logic Programming, II*, North-Holland, pages 29-43.

I. Sag. 1987. Grammatical hierarchy and linear precedence. In *Syntax and Semantics, v.12. Discontinuous Constituency*, Academic Press, pages 303-340.

Richard Steele. 1981. Word order variation. In J. Greenberg (ed). In *Universals of Language, v.4*, Stanford.

A. Zwicky. 1986. Immediate precedence in GPSG. *OSU WPL*, pages 133-138.