

# Improving the Efficiency of Parsing Discontinuous Constituents

Mike Daniels  
daniels@ling.osu.edu  
Draft of 6 May 2003

## 1 Introduction

A prominent tradition within the framework of Head-Driven Phrase Structure Grammar (HPSG, Pollard and Sag 1994) has argued on linguistic grounds for analyses which license so-called discontinuous constituents (Reape 1993; Kathol 1995; Richter and Sailer 2001; Müller 1999a; Penn 1999; Donohue and Sag 1999; Bonami et al. 1999), joining researchers in other linguistic frameworks, including Dependency Grammar (Bröker 1998; Plátek et al. 2001), Tree Adjoining grammar (Kroch and Joshi 1987; Rambow and Joshi 1994), Categorical Grammar (Dowty 1996; Hepple 1994; Morrill 1995), and those positing tangled trees (McCawley 1982; Huck 1985; Ojeda 1987; Blevins 1990) in rejecting string concatenation as the fundamental mode of constituent combination.<sup>1</sup>

More recently, Müller (2003) argues that HPSG grammars for German which license discontinuous constituents should also be preferred on computational grounds. The idea underlying Müller's argument is that in order to license the many word order possibilities (as, for instance, are found in the so-called *Mittelfeld*), a large number of phrase-structure rules or equivalent specifications are needed, resulting in a large number of potential constituents. Since there is no need to distinguish these different word orders in terms of the resulting semantics, positing different rules for each order results only in wasted computational effort.<sup>2</sup>

At the same time, the parsing technology that has been developed to license discontinuous constituents (Johnson 1985; Reape 1991a; van Noord 1991; Covington 1990, 1992; Müller 1996) is far less efficient than Earley's algorithm, the standard parsing algorithm for context-free grammars (Earley 1970), and so the cost of processing such kinds of grammars in practice has outweighed the significant reduction in potential constituents that results from the licensure of discontinuous constituents. The reason for this inefficiency is that discontinuous constituents must be characterized by the subset of the input they cover,<sup>3</sup> while the parsing algorithms for context-free grammars can characterize potential constituents by the string interval they cover (since all potential constituents in a context-free grammar are contiguous). This presents a challenge to efficiency, since the number of possible subsets grows at a much higher rate than the number of possible subintervals as the size of the input increases. The following chart illustrates the problem:

---

<sup>0</sup> I would like to thank Chris Brew, and Bob Levine for their helpful comments. This paper reports joint work done with Detmar Meurers and is an extended and revised version of Daniels and Meurers (2002).

<sup>1</sup> Interestingly, discontinuous constituents are also assumed in the two German treebanks (Skut et al. 1997; Hinrichs et al. 2000).

<sup>2</sup> It has been argued that such different word orders correspond to (subtle) semantic differences (see, for instance, Lenerz 2001). However, until a theory of these differences has been worked out, the only option is to license the indistinguishable word order variations as instances of the same semantic form. For most computational purposes this is also likely to be sufficient in general.

<sup>3</sup> Kasper et al. (1998), Müller (1999b), and Ramsay (1999) discuss some methods for restricting the search space of discontinuous constituency parsing, but the fundamental problem remains.

Length of sentence	Possible sub-intervals	Possible subsets
5	10	32
10	45	1024
15	105	32768
20	190	1048576
$n$	$O(n^2)$	$O(2^n)$

The goal of this paper is to present a general parsing algorithm which is as efficient as Earley’s algorithm for context-free grammars when enough word order information is available and then degrades gradually in efficiency in relation to the number and kind of discontinuities permitted by a grammar. This idea is closely related to the proposal by Suhre (1999). However, while he focuses on formal language properties and provides valuable worst-case complexity results, this paper focuses on the practical aspects of ensuring that order constraints are used for efficient lookup of edges in the chart during completion and to limit the number of rules considered for prediction. The algorithm described herein presents a method for compiling order constraints into two kinds of bitmasks. By using these bitmasks in edge indexing, the algorithm eliminates instances of the generate-and-test paradigm, allowing the parser to check both dominance and word order relations in a tightly integrated way.

The algorithm proposed here also makes it possible to process the daughters of a given rule in any order. As such, it extends the notion of a head-driven algorithm (Kay 1990; van Noord 1991) by additionally ordering the non-head daughters. This gives the grammar writer a finer level of control over the parsing strategy used for each rule, to the point where each rule can be presented in the order that most facilitates processing.

The paper will first introduce the grammar format assumed throughout, and then present the algorithmic building blocks for the parser. This is followed by an outline and illustration of the parsing algorithm itself; the paper then concludes with some evaluatory remarks and directions for future research.

## 2 Describing Discontinuous Consituency

The idea of discontinuous constituency was first introduced into HPSG in a series of papers by Reape (1989, 1990, 1991b, 1993, 1994, 1996), who proposed the idea that word order was determined not at the level of the local tree, but at the level of the *order domain* which could potentially span multiple local trees. Each pre-terminal has a corresponding order domain; as constituents combine to form larger constituents, so their order domains combine to form larger order domains. In particular, each daughter’s order domain enters its mother’s order domain in one of two ways: by being set-unioned in (*domain union*), or by being directly inserted as a unit in which the word order has been fixed (*domain insertion*). Later work (Kathol and Pollard 1995; Kathol 1995; Yatabe 1996) introduced the notion of *partial compaction*, in which only a portion of the daughter’s order domain is isolated prior to set-union.

Domain insertion has two effects:

- **Compaction:** The terminal yield of a domain-inserted non-terminal contains all and only the terminal yield of the nodes it dominates – there are no holes or additional strings.

- **Isolation:** Precedence statements only constrain the order among elements within a local domain. In other words, precedence constraints can never apply across domains.

To illustrate the second of these effects, consider a local tree **A** consisting of two daughters: **B** has order domain [1, 2] and **C** has order domain [3, 4]. If **B** is domain-inserted and **A** is domain-unioned, then the only possible orders for **A** are as shown in Figure 1. If both were domain-unioned, all twenty-four orderings would be licensed. Thus every domain-unioned constituent may potentially be discontinuous.

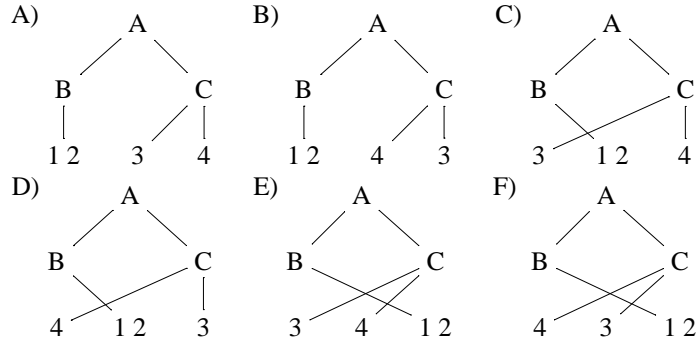


Figure 1: Domain Insertion and Domain Union

## 2.1 Grammar Format

Truly free word order is fairly rare among the world’s languages, however, and so a system for constraining word order within each order domain is needed. Following Suhre (1999), a subset of the linear specification language proposed by Götz and Penn (1997) is used to serve this purpose. In particular, the syntax of ID/LP rules (as commonly used for describing GPSGs (Gazdar et al. 1985); see, e.g. (Shieber 1984)) can be augmented with a means for specifying when and how domains are formed. The resulting grammar format is, of course, interpreted differently when discontinuous constituents are allowed, but the underlying intuition remains the same. As a result, the grammars presented here will be referred to as Generalized ID/LP (GIDL) grammars.<sup>4</sup>

A GIDL grammar consists of four parts:

- 1) A root declaration.
- 2) A set of lexical entries.
- 3) A set of grammar rules.

The root declaration states the start symbol of the grammar and those order constraints which hold in the root domain. Lexical entries have the form  $A \rightarrow t$ , linking the preterminal  $A$  to the terminal  $t$ . Grammar rules have the form  $A \rightarrow \alpha; L; I$ . Here  $A$  is a non-terminal,  $\alpha$  a list<sup>5</sup> of non-terminals,  $L$  a set of order constraints, and  $I$  a set of isolation statements.

<sup>4</sup> Suhre (1999) refers to a similar grammar class as LSL grammars, but this usage is at odds with the much more extensive LSL described by (Götz and Penn 1997).

<sup>5</sup> Note that in contrast to phrase structure rules, the order of the non-terminals in  $\alpha$  is irrelevant for the interpretation of the rule; it only determines the order of processing. This will have a significant impact on grammar development, as described in section 6.2.

$A$  and  $\alpha$  are the left-hand side (LHS) and right-hand side (RHS) of the rule,  $L$  establishes the order constraints under which the rule applies, and  $I$  defines the domains that will be created as a result. Both  $L$  and  $I$  can be empty; thus the simplest type of rule is shown in (1).

(1)  $S \rightarrow NP, VP$

This rule says that an  $S$  may immediately dominate an  $NP$  and a  $VP$ , with no constraints on the relative ordering of  $NP$  and  $VP$ . One may precede the other, or their daughters may be interleaved.

### 2.1.1 Order Constraints

Order constraints are used in two places within a grammar: on individual rules (as *rule-level* constraints) or in isolation statements (as *domain-level* constraints). In this system,<sup>6</sup> all order constraints take the form of precedence constraints: given two elements in the same domain, one must completely precede the other for the resulting parse to be valid. Precedence constraints may optionally require immediate precedence: not only must the constituents appear in a certain order, but there must also be no intervening material.

These two types of precedence are represented as follows:

- **Weak precedence:**  $A < B$ .
- **Immediate precedence:**  $A \ll B$ .

It is assumed that  $A$  and  $B$  do not overlap, nor may either dominate the other (it would otherwise be impossible to express an order constraint on a recursive rule).

Here, the symbols  $A$  and  $B$  may be descriptions or tokens. When parsing atomic categories (as assumed in this paper), a description is simply a category label; as such, a constraint involving descriptions may apply more than once within a given rule or domain. Tokens, on the other hand, are used in rule-level constraints to indicate a particular RHS member; any constraint. In this paper, tokens are represented by numbers corresponding to superscripted indices on the RHS categories. In this paper, a constraint written exclusively with tokens is called a *token-based* constraint; one involving (perhaps exclusively) descriptions is called *description-based*.

For instance, in the rule given in (2), the constraint indicates that category marked as 3 in the rule's RHS (here, the second  $NP$ ) must precede any constituents labelled  $V$  occurring in the same domain.

(2)  $A \rightarrow NP^1, V^2, NP^3 ; 3 < V$

Constraints can therefore be classified along two dimensions: their location, whether rule-level or domain-level; and their composition, description-based (involving at least one description) or token-based (involving no descriptions). Note, however, that domain-level constraints are never token-based. The constraint in (2) is an example of a rule-level, description-based constraint.

---

<sup>6</sup> Order constraints need not be based on precedence. Götz and Penn (1997), for instance, argue for an order constraint of the form “constituent X occupies  $n$ th position within this constituent”. The GIDL format does not make use of such constraints.

### 2.1.2 Isolation Statements

The fourth rule component, the set of isolation statements, establishes the domains that will be created by the rule. Each isolation statement has the form  $\langle \alpha, L, A \rangle$ , where  $\alpha$  is a list of tokens,  $L$  is a list of domain-level order constraints, and  $A$  is the resulting category. Such a statement indicates that the constituents referenced in  $\alpha$  will form a domain with category  $A$  inside which the constraints in  $L$  hold.

The nature of the isolation statement(s) on a rule therefore determines the relationship between the mother's order domain and the daughters' order domains. If a given daughter is never referenced in an isolation statement, then it will be set-unioned into its mother's domain; if it is the sole category in an isolation statement, it will be directly inserted into its mother's domain; and if it occurs along with other categories in an isolation statement, it will be part of a partially compacted element.

It is because of partial compaction that the third component in the isolation statement – the result category – is needed: if only singleton constituents could be compacted, the result category would just be the same as the compacted category. But since the domain that results from partial compaction must be referencable from order constraints,<sup>7</sup> it must have a way to respond to descriptions; this is accomplished through the result category.

Rules (3) and (4) illustrate these possibilities:

$$(3) S \rightarrow S^1, \text{Conj}^2, S^3 ; 1 \ll 2, 2 \ll 3 ; \langle [1], [], S \rangle, \langle [3], [], S \rangle$$

$$(4) VP \rightarrow V^1, NP^2, NP^3 ; ; \langle [1, 2], [], VP \rangle$$

In (3), each of the **S** categories forms its own domain; in (4), the **V** and the initial **NP** form a domain named **VP** to the exclusion of the second **NP**.

## 2.2 Grammar Examples

The grammar in (5) puts the pieces that have been presented together and illustrates how they work together.

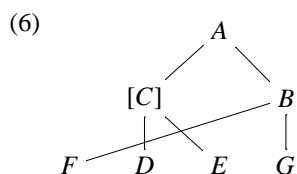
- (5) a)  $\text{root}(A, [B < D])$
- b)  $A \rightarrow B^1, C^2 ; ; \langle [2], [B < D], C \rangle$
- c)  $B \rightarrow F^1, G^2$
- d)  $C \rightarrow D^1, E^2$

(5a) is the root declaration, stating that any input string must ultimately parse as an **A** and giving the order constraints that hold in the root domain. (5b) is a grammar rule stating that an **A** may immediately dominate a **B** and a **C** in either order; it further states that the **C** forms a domain within which the constraint  $B < D$  holds. (5c) and (5d) give rules for **B** and **C**, respectively.

This grammar also illustrates the effect of isolation on constraint application: the string **FDEG** (with parse tree (6)) is allowed by the grammar.

---

<sup>7</sup> Thus while normal compaction has the effect of removing elements from higher order constraints' grasp, so to speak, partial compaction does not affect the accessibility of the non-compacted elements.



There are two non-lexical domains in this tree: the root level containing **C**, **F**, and **G**, and the inner domain containing **D** and **E**. In each of these, the constraint  $B < D$  is vacuously obeyed. Thus the string is licensed by this grammar, even though on the surface, the input material dominated by **B** does not precede that dominated by **D**. Domain formation can therefore be seen as a ‘barrier’ to LP application.

As a second example, note that any context-free grammar can be encoded in this format. For example, the context-free rule in (7) carries the immediate dominance and precedence relations made explicit in (8).

(7)  $S \rightarrow \text{Nom V Acc}$

(8)  $S \rightarrow V^1, \text{Nom}^2, \text{Acc}^3 ; 2 \ll 1, 1 \ll 3 ; \langle [1], [], V \rangle, \langle [2], [], \text{Nom} \rangle, \langle [3], [], \text{Acc} \rangle$

In GIDL terms, a context-free grammar rule implicitly requires that each element of the RHS immediately precede the next, and that each constituent be a contiguous portion of the input string.

### 3 Parser Architecture

Now that the GIDL grammar format has been completely presented, the paper can address some core concepts behind the parsing mechanism.

Traditional approaches to parsing with discontinuous constituents follow the generate-and-test model. Reape (1991a), for instance, presents as a baseline an algorithm which generates all possible permutations of the input string and tests to see which are licensed by a context-free grammar. Similarly, one can take a grammar licensing discontinuous constituents and write out the discontinuity. All non-domain introducing rules must be folded into the domain-introducing rules, and then each permutation of a freely-orderable RHS must become a fixed-order rule on its own. With a phrase-structure parser that combines a context-free backbone with a relational constraint system, the equivalent approach is to model order domains as a feature like any other, writing relations to express the word order requirements. In such a setup, the parser only checks that every part of the input string is actually part of the order domain of the root.

All of these approaches have some sort of permutative aspect to them and are therefore factorial in the worst case.<sup>8</sup> As a result, this paper approaches the task from a different viewpoint: how can a context-free parsing algorithm be adapted to handle GIDL grammars? In this case, Earley’s algorithm (Earley 1970) provides a starting point.

<sup>8</sup> It should certainly be acknowledged that various researchers have worked on making this kind of approach as efficient as possible – see, e.g. (Kasper et al. 1998). A similar approach from the viewpoint of generation (i.e. constrained permutation) can be found in the literature on ‘Shake and Bake’ machine translation (see (Brew 1992) and references cited therein).

### 3.1 Earley's algorithm

In general terms, Earley's algorithm is a memoizing algorithm for context-free parsing.<sup>9</sup> As with any parsing algorithm, the problem is to take a grammar and an input string and determine whether the grammar in question licenses the input string, or in other words, whether a series of context-free rewriting steps could transform the start symbol of the grammar into the input string.<sup>10</sup>

The operation of the algorithm description centers around two operations. In the first step, hypotheses are formed describing the potential expansions of the current target symbol; this process is referred to as *prediction*. The second step combines information describing the location of a subconstituent into an earlier hypothesis for the dominating constituent, producing either a more specific hypothesis or a well-formed substring; this is referred to as *completion*.

In traditional terms, the hypotheses are called *active edges*. They describe the potential for a category to be found covering a span that includes a given location and state the additional evidence that will be needed to confirm this hypothesis: the location of an *active element* and the remaining RHS categories. Active edges are created as the result of a prediction step, and their RHSs grow smaller after each completion step. Once the RHS of an active edge is empty, the edge is referred to as a *passive edge*, as it now encodes the information that a category has definitely been found at a certain location in the input; the passive edge is said to 'provide' that category. Not all passive edges arise from completions, however; the term *lexical edges* is used to refer to those passive edges that map lexical items in the input string to preterminal categories.

Early work on parsing (Younger 1967; Kasami and Torii 1969) made use of the notion of a well-formed substring table: a data structure containing those passive edges that had been found at any given point. Earley's algorithm extends this concept by storing both passive and active edges in a data structure known as an *active chart*.

The parse begins by predicting the root symbol of the grammar. Each time an edge is added to the chart, the parser responds by triggering further prediction and completion steps. Once the chart has finished responding to the initial prediction and its consequences, every edge in the chart spanning the entire input string whose LHS is the root symbol indicates a successful recognition.

As a result, the edges in the chart at the end of the parse can be arranged into a tree, with each edge having the active edge that created it as its parent. Each leaf in this tree represents an edge for which prediction and completion yielded no new edges. Thus all optimizations will take two forms: reducing the amount of effort needed to compute a newly-predicted or completed edge, and finding ways to make the search tree smaller. If an edge can be identified for which no children will ever contribute to a successful parse, preventing that edge from being added to the chart will prune that entire branch of the tree, leading to potentially considerable savings.

---

<sup>9</sup> The notion of a chart parser, introduced in (Kay 1980), postdates Earley's algorithm. The algorithm is explained here in chart parsing terms (rather than Earley's original terminology) to aid transparency.

<sup>10</sup> The remaining part of the parsing process deals with building the corresponding parse trees; this paper does not address this aspect of parsing.

### 3.2 Modifying Earley’s Algorithm

Before proceeding to the details of the GIDL P parsing algorithm, it is important to highlight two ways in which the GIDL P parser differs at a high level from Earley’s original formulation.

In Earley’s original algorithm, a third operation – *scanning* – is used to insert lexical edges into the chart when the active category is a terminal. The parser described here seeds the chart with all of the lexical edges before the parse begins (so that only those lexical rules are predicted whose lexical items appear in the input string). This is done to strengthen the bottom-up component, which is important considering the overall goal of parsing linearization-based HPSG grammars, where much of the information guiding parsing originates in the lexicon. Together with the ability (discussed in section 4.2.1) to determine the order in which the righthand side categories of a rule are predicted, this allows the grammar writer to, for example, specify a head-corner processing strategy.

A similar modification concerns the triggering of prediction and completion steps. In Earley’s algorithm, a completion step is triggered each time a passive edge is added to the chart: all active edges seeking the category provided by the newly-added edge are completed with that edge. Likewise, a prediction step is triggered each time an active edge is added to the chart, creating a new edge for each grammar rule that can generate the edge’s active category.

With the GIDL P parsing algorithm, however, completion must occur whenever an edge is added to the chart, regardless of whether the entered edge is active or passive. This occurs as a result of the non-linearity of GIDL P parsing – because context-free parsing effectively proceeds from left to right, Earley can assume that all possible active edges that will need a given category will be present when the passive edge is added to the chart. In GIDL P parsing, however, that assumption is invalid – an individual rule need not impose any particular order on its components, and even if it does, that order is by no means guaranteed to be compatible with the ordering imposed by any other rule. It is still the case, however, that active edges are only completed with passive edges, and passive edges only with active edges.

### 3.3 From CFG parsing to GIDL P parsing

In the abstract form presented above, the algorithm can be applied to both context-free and GIDL P grammars. To turn it into a concrete algorithm, two things must be addressed: how the coverage of an edge is encoded, and how it is determined that two edges are compatible.

For the ordinary context-free grammar case, the answers were provided by Earley (1970). Edges have the form  $_i[A \rightarrow \alpha \bullet_j \beta]$ , which indicates a parse state in which the string from  $i$  to  $j$  is covered by  $\alpha$  and an  $A$  will have been found if  $\beta$  is found immediately following  $j$ . In a passive edge,  $\beta$  is empty. An active edge, on the other hand, has a nonempty  $\beta$ , the initial element of which is the active element, with  $j$  the *active position*. Newly-predicted edges are of the form  $_i[A \rightarrow \bullet_i \beta]$ , where  $i$  is the active position of the edge triggering prediction. A passive edge  $_k[C \rightarrow \gamma \bullet_l \ ]$  is compatible with an active edge  $_i[A \rightarrow \alpha \bullet_j B\beta]$  when  $j = k$  and  $C = B$ ; the new edge  $_i[A \rightarrow \alpha B \bullet_l \beta]$  covers the string from  $i$  to  $l$ .

For GIDL P grammars, a discussion of edge coverage and edge compatibility must start by describing the data structure used to encode the coverage of an edge in light of the possibility of discontinuous constituents.

### 3.3.1 Efficient edge coverage encoding

The single interval formed by  $i$  and  $j$  that was used to encode the coverage of the edges in the context-free case is not sufficient to model edge coverage in a grammar that licenses discontinuous constituents, as each edge may potentially be covered by a discontinuous subset of the string. Johnson (1985) showed that this issue can be addressed by generalizing from single intervals to *lists of intervals*: for example, [1-3, 5] represents an edge that covers the first, second, third, and fifth words of the input.

As Johnson (1985) pointed out, such lists of intervals are conceptually equivalent to *bitvectors* – finite lists where each position holds a bit indicating whether the corresponding string position is included. Reape (1991a), in an early paper on discontinuous parsing, uses prolog lists to encode bitvectors, so that each position may be either ‘true’, ‘false’, or an anonymous prolog variable. As is known from other applications, bitvectors can also be encoded as integers by representing them as binary numbers,<sup>11</sup> with the least-significant bit of the number corresponding to the leftmost word in the input string. (Note that this representation can be confusing at first, as the leftmost word in the string is represented by the rightmost digit in the standard representation of the vector’s corresponding number.)

The merits of each style of representation can best be determined by considering a set of relevant bitvector operations necessary for GIDL parsing; these are given in the table that follows. Here,  $x$  and  $y$  are bitvectors,  $p$  is a 0-based position index,<sup>12</sup> and AND, OR, XOR, and NOT are the ordinary bitwise operators. A position set to 1 is referred to as *occupied* instead of the more common term *active*, which is avoided here in order to prevent confusion with the use of ‘active’ in the terms ‘active category’ and ‘active edge’. In the examples, the bitvectors are assumed to be five digits long.

- **SINGLETON( $p$ )**: Gives the bitvector in which only  $p$  is occupied.  
Computed as  $2^p$   
Example: singleton(1) is 00010.
- **OVERLAP( $x, y$ )**: Is there any position occupied in both  $x$  and  $y$ ?  
Computed as  $\text{AND}(x, y) \neq 0$   
Example: 10111 and 01010 have a bitwise-AND of 00010, and therefore overlap.
- **COMBINE( $x, y$ )**: Gives the union of  $x$  and  $y$ .  
Computed as  $\text{OR}(x, y)$   
Example: 10110 and 01010 combine to form 11110.
- **RBOUND( $x$ )**: Most-significant occupied bit in  $x$ .  
Computed as  $\lfloor \log_2(x) \rfloor$   
Example: the right bound of 01010 is 3, which is  $\lfloor 3.32 \rfloor$ .
- **LBOUND( $x$ )**: Least-significant occupied bit in  $x$ .  
Computed as  $\text{RBOUND}(\text{XOR}(x, x - 1))$   
Example: the left bound of 01010 is 1, which is the right bound of  $00011 = \text{XOR}(01010, 01001)$ .

---

<sup>11</sup> For example, Davis (2002) mentions the use of integer representations of bit-vectors in the context of machine translation, and some of the inspiration for the bitvector computations below stems from bitboard-based computer chess discussions on `rec.games.chess`.

<sup>12</sup> The term 0-based (in contrast with 1-based), frequently used in computer science literature, means that the first element of the list is assigned position 0.

- **PREFIX( $p$ ):** Gives the bitvector covering all positions  $\leq p$ .  
Computed as  $2^{p+1} - 1$   
Example: **PREFIX(3)** is 01111 (15).
- **SUFFIX( $p$ ):** Gives the bitvector covering all positions  $\geq p$ .  
Computed as  $\text{NOT}(2^x - 1)$   
Example: **SUFFIX(3)** is  $\text{NOT}(00111) = 11000$ .
- **PRECEDE( $x, y$ ):** Does  $x$  completely precede  $y$  (where  $x$  and  $y$  are assumed not to overlap)?  
Equivalent to  $x < y$   
Example: 00011 (3) precedes 01100 (12).
- **IPRECEDE( $x, y$ ):** Does  $x$  immediately precede  $y$  (where  $x$  and  $y$  are assumed not to overlap)?  
Equivalent to  $\text{RBOUND}(x) = \text{LBOUND}(y) - 1$   
Example: the right bound of 00011 is 1, and the left bound of 01100 is 2, so 00011 immediately precedes 01100.
- **ISOLATED( $x$ ):** Does  $x$  form a continuous unit?  
Equivalent to  $x = \text{AND}(\text{PREFIX}(\text{RBOUND}(x)), \text{SUFFIX}(\text{LBOUND}(x)))$   
Example: With the vector 01101,  $\text{AND}(01111, 11111)$  is  $01111 \neq 01101$ , so 01101 is not isolated.

The primary advantage of lists of intervals is constant-time checking for isolation: if the cardinality of the list of intervals is one, the edge is isolated. On the other hand, all of the other operations needed to retrieve and use edges in parsing, such as **OVERLAP** and **COMBINE**, are linear in the length of the lists of intervals (which, in the worst case, only differs from the length of the input string by a constant multiple). Reape cites the flexibility of the list-based representation (i.e. the possibility to mark a position as unknown by using an anonymous variable) as an advantage of that method of representation, but the list representation incurs the same efficiency penalty as the lists of intervals representation.

In contrast, with an integer representation all necessary bitvector operations can be computed as numeric expressions that require time proportional to a logarithm of the length of the input string (where the base of the logarithm is the word size of the executing processor; the exact details depend on the interaction between the processor and the language support for large integers, sometimes referred to as ‘bignums’). In the linearization parsing literature, Ramsay (1999) seems to be the only one to explore this possibility, giving definitions of **OVERLAP** and **COMBINE** and an alternative way of computing **ISOLATED**.

### 3.3.2 Bitmasks as Compiled Order Constraints

The strategy for prediction used by Suhre (1999) is to predict every rule at every position. While this strategy ensures that no possibility is overlooked, it fails to integrate and use the information provided by the word order constraints attached to the rules – in other words, the parser receives no top-down guidance. Some of the edges generated by prediction therefore fall prey to the word order constraints later, in a generate-and-test fashion. This need not be the case. Once a daughter of an active edge has been found, the other daughters should only be predicted to occur in string positions which are compatible with the word order constraints of the active edge. For example, consider the edge in (9).

$$(9) A \rightarrow B^1 \bullet C^2 ; 1 < 2$$

Recall that this notation represents the point in the parse during which the application of this rule has been predicted, and a **B** has already been located. Assuming that **B** has been found to cover the third position of a five-word string, two facts are known. From the order constraint, **C** cannot precede **B**, and from the general principle that a category may only be dominated by one mother constituent, **C** cannot overlap **B**. Thus **C** cannot cover positions one, two, or three.

A central insight of this algorithm is that the same data structure used to describe the coverage of an edge can additionally encode these kinds of restrictions on the parser’s search. This is done by adding two additional bitvectors to each edge: a *negative mask* (n-mask) and a *positive mask* (p-mask).

**Negative Masks.** The n-mask constrains the set of possible coverage vectors which could complete the edge. The 1-positions in a masking vector represent the positions that are masked out: the positions that cannot be filled when completing this edge. The 0-positions in the negative mask represent positions that may potentially be part of the edge’s coverage. For the example above, the coverage vector for the edge is 00100 since only the third word **B** has been found so far. Assuming no restrictions from a higher rule in the same domain, the n-mask for **C** is 00111, encoding the fact that the final coverage vector of the edge for **A** must be either 01000, 10000, or 11000 (that is, **C** must occupy position four, position five, or both of these positions). The negative mask in essence encodes information on where the active category cannot be found.

**Positive Masks.** The p-mask encodes information about the positions the active category **must** occupy. This knowledge arises from immediate precedence constraints. For example, consider the edge in (10).

$$(10) D \rightarrow E^1 \bullet F^2 ; 1 \ll 2$$

If **E** occupies position one, then **F** must at least occupy position two; the second position in the positive mask would therefore be occupied.

### 3.4 The Dot

It should be observed at this point that this parser retains the notion of a single active element per edge, even though any of the RHS categories might occur leftmost in the input string; this is in contrast to Suhre (1999), who essentially follows the direct ID/LP parsing tradition (see (Volk 1996) and references cited therein).

Recall that the dot in Earley’s original parser serves two purposes: (1) it indicates the portion of the *string* that has already been incorporated into chart edges; and (2) it distinguishes the *categories* that have been found from those that are left, highlighting the active category as the category to next be located. In this generalization of Earley’s algorithm, the first purpose is served by the coverage vector; thus the dot only has the second purpose.

Conceptually, using a single dot is sufficient, as for an edge to be completed, every element on the righthand side has to be found at some point. Thus a GIDL parser is free to use the RHS order for other purposes. This parser described here uses the RHS order to determine the order in which the righthand side categories are predicted. As a result, the grammar writer can use the order to specify those daughters to be searched first which are most likely to cause an early failure. For example, a rule introducing a conjunction of sentences can be specified as (11).

(11)  $S \rightarrow \text{Conj}^1, S^2, S^3 ; 2 \ll 1, 1 \ll 3 ; \langle [2], [], S \rangle, \langle [3], [], S \rangle$

This causes the parser to look for the easy-to-identify conjunction before it tries to find the potentially-complex conjunct sentences.

One might object that it is unreasonable to expect a grammar writer to take processing considerations into account. It must be observed, however, that the RHS order has no impact on the correctness of the resulting parse. Much of the work of finding the optimal ordering for the categories in a rule could be accomplished either at compile time by heuristics of varying complexity or by hand as the grammar is written. For the grammars presented in this paper, two heuristics were followed: categories should be ordered by the number of times each is mentioned in an order constraint in that rule, and exclusively pre-terminal categories should be ordered before other categories. Both of these considerations lead to ordering the **Conj** element first in (11).

### 3.5 Domains in Earley’s Algorithm

One of the primary advantages of a chart parser like Earley’s algorithm is the fact that passive edges need only be constructed once; if a given passive edge doesn’t immediately trigger a completion, it remains in the chart to be picked up during future completions. As a result, it won’t always be the case that the parser knows what domain a given passive edge will be used in. Consider the grammar in (12):

- (12) a)  $\text{root}(A, [])$   
 b)  $A \rightarrow B^1, C^2 ; 1 < 2$   
 c)  $B \rightarrow D^1$   
 d)  $C \rightarrow D^1 ; ; \langle [1], [E < F], D \rangle$   
 e)  $D \rightarrow E^1, F^2$

This grammar accepts the strings **EFEF** and **FEFF** and rejects the strings **EFFE** and **FEFE**. Consider the parse of the string **FEFE**. Given rule (12e), the last two symbols of the string constitute a **D**. By rule (12c), this **D** is also a **B**. It is not, however, a **C**: rule (12d) states that **Cs** dominate contiguous sections of the input within which all **Es** precede all **Fs**. As presented so far, a passive edge stores relatively little information: a category label and a bitvector representing the edge’s coverage. In the example above, the passive edge would simply say that a **D** has a coverage of 1100. It is impossible to tell from this alone whether such a **D** would be an acceptable **C** or not.

The parser could simply store each edge’s parse tree (leaving out daughters of compacted constituents) and examine that tree each time a new domain constraint becomes relevant (as in the example above). This is inefficient in general, though – the parse tree of a given edge might turn out to be arbitrarily deep.

Instead, the parser uses the notion of *dormant* (contrasted with *active*) order constraints. When the grammar is compiled, all domain-local LP constraints (that is, those not introduced on individual rules) are added as dormant constraints in every other domain (as long as both components are reachable within that domain; see section 4.5). A dormant constraint is still tracked and updated (as described in section 4.3.1) as normal, with the exception that a constraint violation does not prevent an edge from being created (as a violation in an active constraint does). Instead, it merely reduces the number of domains that

the edge could be completed into in the future. When a domain-introducing rule is predicted from a mother edge, if any of the dormant constraints for that domain had already been violated on that edge, that prediction will not create any new edges.

In essence, the set of dormant constraints can be seen as a ‘distilled’ version of the edge’s parse tree: the minimal amount of information needed to determine future domain memberships.

### 3.6 Non-overlapping bitvector generation

An important aspect of the parsing algorithm requires the generation of all bitvectors that do not overlap with a given bitvector. This is done in a memoizing fashion: each time a new list of vectors is requested, the parser checks to see if that list has already been created.

If not, the task is as follows: for a given bitvector, there will be a certain number of unoccupied positions, or *holes*. Each of the holes could be either filled or empty in a non-overlapping bitvector. In contrast, each occupied position in the bitvector cannot be occupied in a non-overlapping bitvector. Thus each subset of the holes in the original bitvector corresponds to one of the non-overlapping bitvectors.

The parser generates these bitvectors in a Gray code order: an order of the binary numbers in which only one digit changes at each step. Since only one position changes at a time, the full list of potential subsets is generated in an efficient manner.<sup>13</sup>

The procedure, then, for generating non-overlapping bitvectors is as follows. Consider the target bitvector and calculate the mapping between holes and bitvector positions, counting the number of holes in the process. Then generate the Gray code instructions (a list of positions to successively toggle) for that length, and use the hole:position mapping to translate these instructions into actual bitvector positions.

For example, take the vector 010100. Since the holes in this vector occur in the first, second, fourth, and sixth positions, the hole:position mapping is 0:0, 1:1, 2:3, 3:5 (in other words, hole 0 is at position 0, hole 1 is at position 1, etc.). There are four holes in total; the Gray code instructions on input 4 are (0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0) (in other words, first toggle hole 0, then hole 1, then hole 0 again, and so on). Using the hole:position mapping, the list of instructions becomes (0, 1, 0, 3, 0, 1, 0, 5, 0, 1, 0, 3, 0, 1, 0). From this list, the set of non-overlapping bitvectors can be generated: starting with the bitvector 000000, the parser toggles the positions in the list one at a time. This process is illustrated in the diagram in Figure 2, where the underlined position is the one that’s about to be toggled.<sup>14</sup>

### 3.7 Bitvector hashing

Because the number of such bitvectors grows exponentially with the length of the bitvector, there is a need to ensure that the generation procedure is constrained. This is done by working with *bitvector hashes*: smaller bitvectors such that, if  $a'$  is the hash of  $a$  and  $b'$  is the hash of  $b$ ,  $a'$  overlaps with  $b'$  only if  $a$  overlaps with  $b$ .

---

<sup>13</sup> A full discussion of the motivation and algorithms for Gray code generation can be found in Knuth (2002); my implementation of the parser adopts Algorithm L from that source to generate the Gray code.

<sup>14</sup> If the pattern of dashes looks familiar, note that it’s the solution to the Towers of Hanoi with four discs, where the rule that only one disc may move at a time corresponds to the constraint that only one bit flips at a time.

---

543210
010100
0 0 <u>00</u> → 000000
0 0 <u>01</u> → 000001
0 0 <u>11</u> → 000011
0 <u>0</u> 10 → 000010
0 <u>1</u> 10 → 001010
0 1 <u>11</u> → 001011
0 1 <u>01</u> → 001001
<u>0</u> 1 00 → 001000
1 1 <u>00</u> → 101000
1 1 <u>01</u> → 101001
1 1 <u>11</u> → 101011
1 <u>1</u> 10 → 101010
1 0 <u>10</u> → 100010
1 0 <u>11</u> → 100011
1 0 <u>01</u> → 100001
1 0 00 → 100000

Figure 2: Non-overlapping vector generation

---

The parser indexes all edges by the hash of their n-mask. When attempting to complete an edge, the parser retrieves all edges with non-overlapping hashes. For each edge so retrieved, its actual n-mask is tested for overlap with the edge triggering completion.

The hash function is as follows. Given an input vector  $v_0v_1v_2 \dots v_{x-1}$  and a target length  $m$ , the hashed vector  $h_0h_1 \dots h_{m-1}$  is desired. First compute the group width  $r = \lceil x/m \rceil$ . Then the hashed vector is described by

$$h_{(m-1)-n} = \bigwedge_{i=rn}^{rn+(r-1)} v_i$$

as  $0 \leq n < m$ .

For example, the computation of the 8-digit hash of 00111101111000010100 is given in (3).

---

00111101111000010 <u>100</u>	→ <u>0</u>
001111011110000 <u>10</u>	→ <u>00</u>
00111101111 <u>000</u>	→ <u>000</u>
00111101 <u>111</u>	→ <u>0001</u>
00111 <u>101</u>	→ <u>00010</u>
00 <u>111</u>	→ <u>000101</u>
<u>00</u>	→ <u>0001010</u>
	→ <u>00010100</u>

Figure 3: Example of hash calculation

---

In summary, the use of bitvector hashes allows the parser to replace the process on the left with that on the right in the table below.

Step	Without hashes	With hashes
1		Compute the bitvector hash.
2	Generate all bitvectors that do not overlap the target's n-mask.	Generate all bitvectors that do not overlap the target's hash.
3	Retrieve all such edges.	Retrieve all such edges.
4		See if each edge found actually overlaps with the target.
5	Test each edge found for compatibility.	Test each remaining edge for compatibility.

While step 3 returns more edges in the procedure on the right than in that on the left, step 2 is much simpler on the right than on the left. The effect of the latter far outweighs that of the former in practice, resulting in a net gain.

## 4 The GIDL parsing algorithm

Now that the building blocks have been presented, the algorithm for parsing GIDL grammar can now be given in full. This chapter presents the complete algorithm; chapter 5 will provide two example parses illustrating the steps shown below.

### 4.1 Initialization

The chart is seeded with passive edges corresponding to each word in the input. The parse then begins by predicting an isolated instance of the start symbol covering the entire input; each final completion of this edge will correspond to a successful parse.

### 4.2 Prediction

Recall that prediction takes an active edge seeking its active category and produces a new edge that will provide that category if successfully completed. The parser considers each rule in the grammar that provides the symbol being predicted, and for each rule, it generates the masks for the new edge, taking both rule-based and domain-based order constraints into account (as described in section 4.2.1). The resulting masks are checked to ensure that there is enough space in the resulting mask for the minimum number of categories required by the rule.<sup>15</sup>

#### 4.2.1 Mask Computation

To compute the new masks, the parser examines each order constraint in turn for references to the category currently being predicted. Each reference provides a component of the final mask, as indicated below:

<sup>15</sup> Note that this optimization assumes epsilon rules are absent from the grammar.

	n-mask component	p-mask component
Cat < Loc	SUFFIX(LBOUND(C))	
Loc < Cat	PREFIX(RBOUND(C))	
Cat ≪ Loc	SUFFIX(LBOUND(C))	SINGLETON(LBOUND(C) - 1)
Loc ≪ Cat	PREFIX(RBOUND(C))	SINGLETON(RBOUND(C) + 1)

Here, Cat refers to the category being predicted (in the form of either a description or a token), and Loc refers to the location of an already-located category. (Thus constraints that do not mention the category being predicted and constraints that do not refer to any already-located categories are irrelevant for prediction.)

Each constraint in turn is checked to see which (if any) of these patterns it matches, and the relevant mask components are calculated. At the end, all n-mask components are COMBINED, as are all p-mask components: these are the new masks.

For example, if the parser is predicting *verb* as token 2, it would respond to some sample constraints as follows:

	n-mask component	p-mask component
00010 < verb	00011	
3 < verb		
2 ≪ 01000	11000	00100
result	11011	00100

For the first constraint, the parser sees that categories matching the description *verb* must follow the position 00010, generating the negative mask component 00011. The second constraint doesn't contribute anything, as it does not refer to any as-yet determined locations. The third constraint requires category number 2 to immediately precede the position 01000. This generates both a negative mask component (11000), which encodes the fact that the category cannot follow the position given, and a positive mask component (00100), which encodes the fact that the category must include the third position.

Once the new masks have been calculated, the parser ensures that an edge is only entered into the chart if the n-mask has enough space for the rule being considered. For instance, a rule like (13) cannot apply within a mask that has only one unoccupied position, so such an edge would be blocked from entering the chart at this point.

(13)  $A \rightarrow B^1, C^2$

If it passes this check, the newly-predicted edge is entered into the chart with an empty coverage vector; as usual, the chart will respond to this by applying completion and prediction to this edge.

### 4.3 Completion

Recall that completion is the process of combining a compatible pair of active and passive edges to produce a new edge. The parser considers each edge compatible with the edge that triggered completion. For each such edge, the parser constructs the LP statements for the new edge (as described in section 4.3.1), the new n-mask, and the new coverage vector. If this was a final completion (the resulting edge is passive), the parser must additionally check to see that the p-mask was respected.

Edges are indexed by their n-mask hash and active element, so the parser can efficiently retrieve only those edges which are likely to yield successful completions. In particular, the parser takes the n-mask hash of the triggering edge and generates a set of non-overlapping bitvectors. All edges having such a bitvector as their n-mask hash are then examined to see if they are category-compatible, and then if their actual n-masks are non-overlapping.

### 4.3.1 Order Constraint Updating

For each edge retrieved, the parser must update the word order constraints of the active edge with the coverage of the passive edge. As edges are initially constructed from grammar rules, all order constraints are initially expressed in terms of either descriptions or tokens. As the parse proceeds, these constraints are updated in terms of the actual locations where matching constituents have been found. For example, a constraint like  $1 < 2$  (where 1 and 2 are tokens) can be updated with the information that the constituent corresponding to token 1 has been found as the first word, i.e. as position 00001.

**Activating dormant constraints.** Before this can be done, however, the parser must first check to see if any dormant constraints on the passive edge need to be activated, based on the active edge's domain. The first step of this process is to find the domain that the passive edge is being completed into. If the active element of the active edge is referred to in an isolation statement, then that domain is the relevant one. Otherwise, the passive edge is assumed to be completed into the active edge's own domain. Once the relevant domain has been determined, the constraints for that domain will be either active or dormant on the passive edge. If dormant, they must be merged with the active edge's active constraints; all other dormant constraints on the passive edge are merged with the active edge's dormant constraints. The procedure for merging constraints will be discussed below.

**Updating token-based constraints.** Each update step will take one of the following forms:

- The first time one of the categories mentioned in a precedence constraint has been found, the constraint is updated as above and tested to see whether there is enough space for the other category. For example, if, given the constraint  $1 \ll 2$ , constituent 2 is found as the first word of the string, the constraint will be impossible to fulfill.
- When the remaining category of a precedence constraint is found, the parser checks that the constraint actually holds; if it does, then that constraint will not appear as part of the word order constraint set of the resulting edge. If the constraint is discovered to have been violated, the completion step aborts and no edge is added to the chart.

**Updating and merging description-based constraints.** With description-based constraints, weak and immediate precedence constraints must be handled separately.

In a weak precedence constraint, the parser need only keep track of the most extreme matching cases (the *frontiers*), if any. For example, the constraint  $\text{NP} < \text{VP}$  needs to keep track of the rightmost **NP** and the leftmost **VP** seen in the domain so far. As long as the rightmost **NP** remains to the left of the leftmost **VP**, the constraint will never cause completion to fail. Similarly, when merging two weak precedence constraints, the most extreme version of each frontier is kept.

Immediate precedence constraints, on the other hand, need only keep track of whether zero, one, or many of each part of the constraint has been seen. The possibilities are as follows:

A << B		B		
		Zero	One	Many
A	Zero	OK	OK	OK
	One	OK	OK if precedence is respected	Failure
	Many	OK	Failure	Failure

Here, the rows represent the status of the lefthand side of the constraint, and the columns represent the status of the righthand side; the cells then encode whether that particular status indicates a constraint violation. For instance, if the parser knows that NP << VP in this domain, it is acceptable for there to be several VPs, as long as no NPs are present (since, by definition, a category cannot immediately precede multiple locations).

To summarize, immediate precedence constraint violations can be detected by keeping count of the occurrences of each side of the constraint; the one exception is the situation represented by the center cell, where each side has been observed once. In this case, the parser must check whether the locations are properly adjacent.

The corresponding chart for merging two count values is straightforward:

A + B		B		
		Zero	One	Many
A	Zero	Zero	One	Many
	One	One	Many	Many
	Many	Many	Many	Many

**Final Steps.** Once the word order constraints have been successfully updated, the rest of the new edge is easy to compute: the category of the edge is the category of the active edge, the missing righthand side is the tail of the active edge's righthand side, and the coverage vector is the bitwise or of the two edges' coverage vectors. Finally, if this was a final completion (that is, the edge being created is passive), the parser checks to see if the p-mask was respected: in other words, every occupied bit in the p-mask must be occupied in the new coverage vector. The resulting edge is then added to the chart and itself triggers another round of completion and prediction.

#### 4.4 Agendas and the Search Tree

The overall search strategy of the parser is as follows: Seed the chart with passive edges for each word of the input. Initiate the parse by predicting the root category; each edge added to the chart triggers further completions and predictions.

In each of these steps, the parser will often have many possibilities to explore. For instance, there may be many rules in the grammar that provide the root category, or there may be many compatible edges in the chart to complete with. A recurring concept in parser design is that of an *agenda* – a data structure that keeps track of the as-yet unexplored possibilities at each point and decides which will be explored next. Since the current implementation of this algorithm is an all-paths parser, the nature of the agenda plays a relatively minor role, and so Prolog's call stack implicitly represents the parsing agenda. Thus the consequences of any choice are explored before alternatives to that choice are explored.

## 4.5 Grammar Compilation

As can be seen from the algorithm presented so far, the parser spends a significant time on keeping track of order constraints. It is therefore desirable to be able to ensure that each constraint stored on an edge is one that could potentially be relevant to that edge. This is accomplished through a compilation phase. This phase, along with allowing grammar writers to use a more natural input syntax (including, for instance, global order constraints), ensures that constraints only appear on domains in which both components of the constraint are reachable (i.e. could potentially appear in that domain). For example, the grammar in Figure 4 is compiled to that in Figure 5.

- 
- a)  $\text{root}(A, [E < B, J < I, J < G])$
  - b)  $A \rightarrow B^1, C^2$
  - c)  $C \rightarrow D^1, E^2, F^3 ; \langle [1,3], [D < J, E \ll F, E < B, J < I, J < G], K \rangle$
  - d)  $E \rightarrow J^1, G^2 ; 1 < 2$
  - e)  $E \rightarrow H^1, I^2$
  - f)  $F \rightarrow J^1, C^2 ; \langle [1,2], [E \ll 1, E < B, J < I, J < G], F \rangle$

Figure 4: The Effects of Grammar Compilation: Before

- 
- a)  $\text{root}(A, [E < B, J < G])$
  - b)  $A \rightarrow B^1, C^2$
  - c)  $C \rightarrow D^1, E^2, F^3 ; \langle [1,3], [D < F, E \ll F], K \rangle$
  - d)  $E \rightarrow J^1, G^2 ; 1 < 2$
  - e)  $E \rightarrow H^1, I^2$
  - f)  $F \rightarrow J^1, C^2 ; \langle [1,2], [E \ll 1, F < I, F < G], F \rangle$

Figure 5: The Effects of Grammar Compilation: After

The first step in grammar compilation is to compute the reachability of each category and domain. By definition, the LHS category reaches all non-isolated RHS categories and everything they reach. This can be calculated through an iterative procedure, as illustrated in Figure 6 for the grammar in (4).

---

A	$\{B, C\} \cup \text{reach}(C)$	$\{B, C, E, J, G\} \mid \{B, C, E, H, I\}$
C	$\{E\} \cup \text{reach}(E)$	$\{E, J, G\} \mid \{E, H, I\}$
E	$\{J, G\} \mid \{H, I\}$	$\{J, G\} \mid \{H, I\}$
F	$\emptyset$	$\emptyset$
dom[0]	$\{A\} \cup \text{reach}(A)$	$\{A, B, C, E, J, G\} \mid \{A, B, C, E, H, I\}$
dom[1]	$\{D, F\} \cup \text{reach}(F)$	$\{D, F\}$
dom[2]	$\{J, C\} \cup \text{reach}(C)$	$\{C, E, J, G\} \mid \{J, C, E, H, I\}$

Figure 6: Reachability computation

The process starts by calculating the basic reachability for each category (given in the top-left block of Figure 6). For instance, rule (4f) indicates that **F** reaches nothing, since all of its RHS categories are referenced in an isolation statement. Since there are two rules

expanding **E**, that symbol has a disjunctive reachability: one disjunct for each rule. Finally, the rules for **A** and **C** indicate that each reaches a non-terminal, so their reachabilities include a transitive component. Once the basic reachabilities have been computed, the transitive reachabilities are unioned in. The process cycles through each category until there are no changes, yielding the set of reachabilities on the right.

At this point, the reachability of each domain can be calculated. Each domain reaches the categories it contains as well as the categories they reach. Since at this point the full reachability of each category is known, the domain reachabilities can be computed in a single pass.

Now the information about each domain's reachability can be used to optimize the placement of the domain-level order constraints. For a constraint to be relevant to a given domain, both of its components must be reachable in that domain. In Figure 4, the root domain contains the constraint  $J < I$ , yet **J** and **I** are never simultaneously reachable in that domain; as a result, that constraint is absent from the root domain in Figure 5.

## 5 Sample Parses

Having described the parsing algorithm in general, the paper will now present two concrete examples: the parser's actions on a specific grammar and input sentence.

### 5.1 Relatively-free word order

This sample parse uses the grammar in (14) and the sentence in (15).<sup>16</sup>

- (14) a) root(s, []).  
 b)  $s \rightarrow \text{verb}^1, \text{nom}^2, \text{acc}^3 ; 2 < 1, 3 < 1$   
 c)  $s \rightarrow \text{verb}^1, \text{nom}^2 ; 2 < 1$   
 d)  $s \rightarrow \text{conj}^2, s^1, s^3 ; 1 \ll 2, 2 \ll 3 ; \langle [1], 1, s \rangle, \langle [3], 2, s \rangle$   
 e)  $\text{acc} \rightarrow \text{adj}^1, \text{acc}^2$   
 f) नलस्  $\rightarrow$  nom 'Nala' (a proper name)  
 g) नगरम्  $\rightarrow$  acc 'city'  
 h) अगच्छत्  $\rightarrow$  verb 'went'  
 i) च  $\rightarrow$  conj 'and'  
 j) अवदत्  $\rightarrow$  verb 'spoke'  
 k) रुचिरम्  $\rightarrow$  adj 'shining'

- (15) रुचिरम् नलस् नगरम् अगच्छत् च नलस् अवदत्  
 shining Nala city went and Nala spoke  
 'Nala went to the shining city and Nala spoke'

The grammar can be summarized as follows: A sentence may consist of a verb and either one or two arguments preceding it. A sentence may also consist of a conjunction immediately between two (conjunct) sentences, each of which forms an isolated domain. Finally, accusatives may be modified by an adjective which may occur anywhere in a sentence,

<sup>16</sup> The example has been tokenized from रुचिरं नलो नगरमगच्छत् नलो ऽवदत्.

before or after the accusative it modifies. Note also that the example sentence contains the discontinuous constituent रुचिरम् नगरम् ‘shining city’.

Before parsing, the parser seeds the chart with the lexical entries, each covering a singleton vector.

The parser’s actions at each step are illustrated in the following format: Edge lines begin with a line number, followed by a description of the step that generated that edge, the number of the resulting edge, three bitvectors representing the edge’s coverage, n-mask, and p-mask, the category of the edge, its RHS, and its order constraints. Failure lines (as number 11 below) only contain a line number and a description of the failure.

For instance, line 1 was created by scanning the input word रुचिरम्. The resulting edge, number 1, covers the leftmost word of the input (0000001); as a consequence, it has that position blocked in its n-mask, and like all lexical edges, it has an empty p-mask. The edge provides the category *adj*, is passive (since the rhs list is empty), and has no order constraints.

#	Description	E	Cover	N-Mask	P-Mask	LHS	RHS
	RLPs			Isos			
1	SCAN रुचिरम्	1	0000001	0000001	0000000	adj	[]
	[]			[]			
2	SCAN नलस्	2	0000010	0000010	0000000	nom	[]
	[]			[]			
3	SCAN नगरम्	3	0000100	0000100	0000000	acc	[]
	[]			[]			
4	SCAN अगच्छत्	4	0001000	0001000	0000000	verb	[]
	[]			[]			
5	SCAN च	5	0010000	0010000	0000000	conj	[]
	[]			[]			
6	SCAN नलस्	6	0100000	0100000	0000000	nom	[]
	[]			[]			
7	SCAN अबदत्	7	1000000	1000000	0000000	verb	[]
	[]			[]			

In these parse traces, the first row contains the line number of the parsing step and a brief description of the parser action. This is followed by either the details of the resulting edge (edge number, coverage vector, n-mask, p-mask, lhs, rhs, rule-based order constraints, isolation statements) or the reason that no edge was created.

The parse itself begins by predicting the start symbol of the grammar, here *s*. Each of the rules that can generate this symbol are considered in order. The prediction step adds a new edge to the chart based on the rule. All edges generated by prediction have coverage 0000000. The n-mask 0000000 is provided by this step’s trigger (here, the dummy edge 0), and the fact that this is a root edge sets its p-mask to 1111111 (since a successful parse must cover the entire string).

8	PRED s in 0	8	0000000	0000000	1111111	s	[verb <sup>1</sup> ,nom <sup>2</sup> ,acc <sup>3</sup> ]
	[c2 < c1,c3 < c1]			[]			

Since the category *verb* cannot be predicted (no grammar rules provide *verb*), this addition triggers completion, and the parser begins to look for passive edges providing *verb*. Here, edge 4 is the first edge in the chart for which this is the case, so edge 9 is generated by completing 8 with 4. The differences between edges 8 and 9 illustrate the process of completion: *verb*<sup>1</sup> has been removed from the list of daughters and the constraint  $c2 < c1$  has been updated to  $c2 < p8$  (representing the fact that category 1 has been found at position  $0001000_2 = 8_{10}$ ).

9	COMP 8, 4	9	0001000	0001000	1111111	s	[nom <sup>2</sup> ,acc <sup>3</sup> ]
	[c2 < p8,c3 < p8]			[]			

As before, prediction cannot do anything with *nom*, so the parser responds to edge 9 with completion. Here, a *nom* is found covering position 0000010. The constraint  $c2 < p8$  is updated to the concrete form  $p2 < p8$ . Since position 2 does in fact precede position 8, this constraint will not appear on the new edge.

10	COMP 9, 2	10	0001010	0001010	1111111	s	[acc <sup>3</sup> ]
	[c3 < p8]			[]			

The active category is now *acc*; since rule (14e) provides that category, it is used for prediction. From the order information on edge 10, the parser knows that the *acc* cannot occur after position 8, nor can it overlap with edge 10's n-mask of 0001010. These facts combine to generate an n-mask of 1111010. Since the category being predicted does not participate in any immediate precedence constraints, its p-mask is empty.

11	PRED acc in 10	11	0000000	1111010	0000000	acc	[adj <sup>1</sup> ,acc <sup>2</sup> ]
	[]			[]			

The parser can now complete edge 11 with the adjective in position 1. Note that by ordering the adjective first, the parser will not predict another application of the recursive rule without first finding an actual adjective to justify the recursion.

12	COMP 11, 1	12	0000001	1111011	0000000	acc	[acc <sup>2</sup> ]
	[]			[]			

The active category in this edge is again *acc*; this time, prediction from rule (14e) fails, as edge 12's n-mask only has one open space.

13	PRED acc in 12 fails – no room for 2 elements in 1111011						
----	----------------------------------------------------------	--	--	--	--	--	--

Now the accusative noun in position 3 is used for completion. Since the resulting edge is passive, its n-mask is reset to the edge's coverage.

14	COMP 12, 3	13	0000101	0000101	0000000	acc	[]
	[]			[]			

At this point, the parser can now try to complete edge 10 with the two-word accusative noun phrase. This fails because the resulting passive edge does not respect its p-mask. In effect, it claims to have satisfied the root symbol of the grammar, but fails to cover the entire string.

15	COMP 10, 13 fails – p-mask 1111111 $\nrightarrow$ 0001111.						
----	------------------------------------------------------------	--	--	--	--	--	--

The parser's options for completing with edge 13 are now exhausted, and since edge 13 is passive, it cannot trigger prediction. As a result, the parser backtracks to the last point where it had unfinished business; in this case, now that it has finished predicting from edge 10, it can attempt to complete with edge 10. This edge has *acc* as its active category, which edge 3 provides. The parser therefore attempts to complete edge 10 and edge 3; this attempt also fails to pass the p-mask test.

16	COMP 10, 3 fails – p-mask 1111111 $\nrightarrow$ 0001110.						
----	-----------------------------------------------------------	--	--	--	--	--	--

The parser now reverts to the task of finding the verb in edge 8, choosing to complete with the word in final position. From now on, only steps introducing new aspects of the parser will be annotated.

17	COMP 8, 7	14	1000000	1000000	1111111	s	[nom <sup>2</sup> ,acc <sup>3</sup> ]
	[c2 < p64,c3 < p64]			[]			
18	COMP 14, 2	15	1000010	1000010	1111111	s	[acc <sup>3</sup> ]
	[c3 < p64]			[]			
19	PRED acc in 15	16	0000000	1000010	0000000	acc	[adj <sup>1</sup> ,acc <sup>2</sup> ]
	[]			[]			
20	COMP 16, 1	17	0000001	1000011	0000000	acc	[acc <sup>2</sup> ]
	[]			[]			
21	PRED acc in 17	18	0000000	1000011	0000000	acc	[adj <sup>1</sup> ,acc <sup>2</sup> ]
	[]			[]			

At this point, the completion of edge 17 and edge 3 succeeds, but the result is already in the chart. The parser acknowledges this, and no new steps are triggered as a result.

22	COMP 17, 3 is already in the chart as edge 13						
23	COMP 15, 13 fails – p-mask 1111111 $\nrightarrow$ 1000111.						
24	COMP 15, 3 fails – p-mask 1111111 $\nrightarrow$ 1000110.						
25	COMP 14, 6	19	1100000	1100000	1111111	s	[acc <sup>3</sup> ]
	[c3 < p64]			[]			
26	PRED acc in 19	20	0000000	1100000	0000000	acc	[adj <sup>1</sup> ,acc <sup>2</sup> ]
	[]			[]			
27	COMP 20, 1	21	0000001	1100001	0000000	acc	[acc <sup>2</sup> ]
	[]			[]			
28	PRED acc in 21	22	0000000	1100001	0000000	acc	[adj <sup>1</sup> ,acc <sup>2</sup> ]
	[]			[]			
29	COMP 21, 3 is already in the chart as edge 13						
30	COMP 19, 13 fails – p-mask 1111111 $\nrightarrow$ 1100101.						
31	COMP 19, 3 fails – p-mask 1111111 $\nrightarrow$ 1100100.						

Here, the parser has finally returned to the point where it originally chose a rule from the grammar with which to predict the start symbol, and now turns to the second rule that generates *s*.

32	PRED s in 0	23	0000000	0000000	1111111	s	[verb <sup>1</sup> ,nom <sup>2</sup> ]
	[c2 < c1]			□			
33	COMP 23, 4	24	0001000	0001000	1111111	s	[nom <sup>2</sup> ]
	[c2 < p8]			□			
34	COMP 24, 2 fails – p-mask 1111111 ↗ 0001010.						
35	COMP 23, 7	25	1000000	1000000	1111111	s	[nom <sup>2</sup> ]
	[c2 < p64]			□			
36	COMP 25, 2 fails – p-mask 1111111 ↗ 1000010.						
37	COMP 25, 6 fails – p-mask 1111111 ↗ 1100000.						
38	PRED s in 0	26	0000000	0000000	1111111	s	[conj <sup>2</sup> ,s <sup>1</sup> ,s <sup>3</sup> ]
	[c1 ≪ c2,c2 ≪ c3]			[s:[1]:1,s:[3]:2]			
39	COMP 26, 5	27	0010000	0010000	1111111	s	[s <sup>1</sup> ,s <sup>3</sup> ]
	[c1 ≪ p16,p16 ≪ c3]			[s:[1]:1,s:[3]:2]			
40	PRED s in 27	28	0000000	1110000	0001000	s	[verb <sup>1</sup> ,nom <sup>2</sup> ,acc <sup>3</sup> ]
	[c2 < c1,c3 < c1]			□			
41	COMP 28, 4	29	0001000	1111000	0001000	s	[nom <sup>2</sup> ,acc <sup>3</sup> ]
	[c2 < p8,c3 < p8]			□			
42	COMP 29, 2	30	0001010	1111010	0001000	s	[acc <sup>3</sup> ]
	[c3 < p8]			□			
43	PRED acc in 30 is already in the chart as edge 11						
44	COMP 30, 13	31	0001111	0001111	0000000	s	□
	□			□			
45	COMP 27, 31	32	0011111	0011111	1111111	s	[s <sup>3</sup> ]
	[p16 ≪ c3]			[s:[3]:2]			

By line 46, one of the conjuncts has been found. Since only two positions are left uncovered, the remaining conjunct must fit in two positions.

46	PRED s in 32 fails – no room for 3 elements in 0011111						
47	PRED s in 32	33	0000000	0011111	0100000	s	[verb <sup>1</sup> ,nom <sup>2</sup> ]
	[c2 < c1]			□			
48	COMP 33, 7	34	1000000	1011111	0100000	s	[nom <sup>2</sup> ]
	[c2 < p64]			□			
49	COMP 34, 6	35	1100000	1100000	0000000	s	□
	□			□			
50	COMP 32, 35	36	1111111	1111111	0000000	s	□
	□			□			
51	PRED s in 32 fails – no room for 3 elements in 0011111						
52	COMP 30, 3	37	0001110	0001110	0000000	s	□
	□			□			
53	COMP 27, 37	38	0011110	0011110	1111111	s	[s <sup>3</sup> ]
	[p16 ≪ c3]			[s:[3]:2]			
54	PRED s in 38 fails – no room for 3 elements in 0011111						
55	PRED s in 38 is already in the chart as edge 33						
56	PRED s in 38 fails – no room for 3 elements in 0011111						
57	COMP 38, 35 fails – p-mask 1111111 ↗ 1111110.						
58	PRED s in 27	39	0000000	1110000	0001000	s	[verb <sup>1</sup> ,nom <sup>2</sup> ]
	[c2 < c1]			□			
59	COMP 39, 4	40	0001000	1111000	0001000	s	[nom <sup>2</sup> ]
	[c2 < p8]			□			

60	COMP 40, 2	41	0001010	0001010	0000000	s	[]
	[]			[]			

Here, an attempt to treat the second and fourth words as a conjunct fails from a lack of continuity.

61	COMP 27, 41 fails – 0001010 is not contiguous						
62	PRED s in 27	42	0000000	1110000	0001000	s	[conf <sup>2</sup> ,s <sup>1</sup> ,s <sup>3</sup> ]
	[c1 << c2,c2 << c3]			[s:[1]:1,s:[3]:2]			
63	SUCCESS: 36						

The overall search tree for this parse is given in (7). In this diagram, nodes are given in the format  $xi$ , where  $x$  indicates either (c)ompletion or (p)rediction and  $i$  is the edge created. The boldface node indicates the edge corresponding to a successful parse; its label  $c36$  indicates that it corresponds to edge 36, which was created by a completion step.

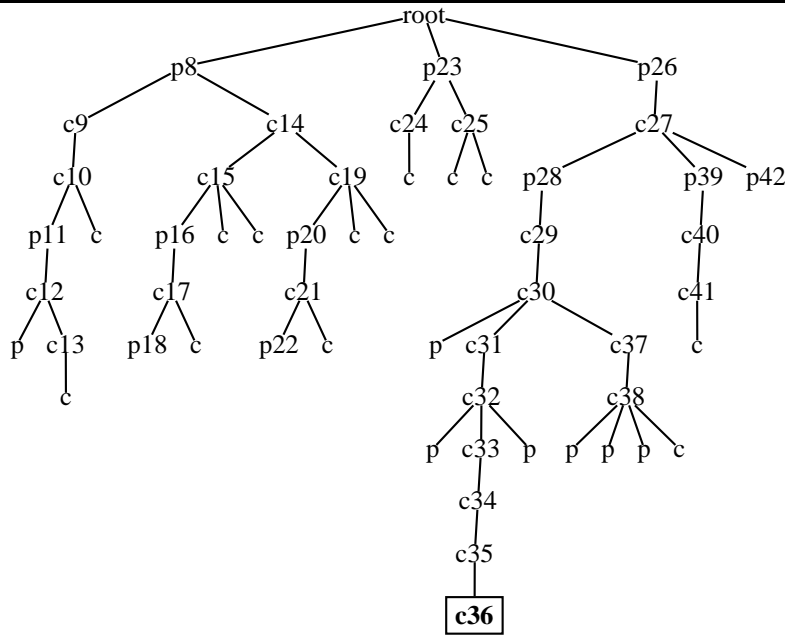


Figure 7: Parser's search tree

## 5.2 Domains and Dormant Constraints

This sample parse uses the grammar in (16) and the input sentence **fegefhi**.

- (16) a) root(a, [g << c, d < x]).  
 b)  $a \rightarrow b^1, c^2, g^3 ; 1 < 2$

- c)  $b \rightarrow d^1$
- d)  $c \rightarrow h^1, d^2, i^3 ; ; \langle [2], [e < f], d \rangle, \langle [1,3], [h < i], x \rangle$
- e)  $d \rightarrow e^1, f^2$

This grammar has been written to exercise the domain-level constraint handling abilities of the parser; it is a more complicated version of (12). Most importantly, the partial compaction in (16d) will require a constraint merger.

As before, the parse starts with a set of lexical edges.

#	Description	E	Cover	N-Mask	P-Mask	LHS	RHS
	RLPs			DLPs			
	MLPs			Isos			
1	SCAN (f)	1	0000001	0000001	0000000	f	[]
	[]			[]			
	[]			[]			
2	SCAN (e)	2	0000010	0000010	0000000	e	[]
	[]			[]			
	[]			[]			
3	SCAN (g)	3	0000100	0000100	0000000	g	[]
	[]			[]			
	[]			[]			
4	SCAN (e)	4	0001000	0001000	0000000	e	[]
	[]			[]			
	[]			[]			
5	SCAN (f)	5	0010000	0010000	0000000	f	[]
	[]			[]			
	[]			[]			
6	SCAN (h)	6	0100000	0100000	0000000	h	[]
	[]			[]			
	[]			[]			
7	SCAN (i)	7	1000000	1000000	0000000	i	[]
	[]			[]			
	[]			[]			

The parser begins by predicting the start symbol. Note that in addition to the information presented in the last example, there are (active) domain constraints (DLPs) and dormant constraints (MLPs) on each edge. Since the root is always part of domain 0, the constraints for domain 0 are active and those for domains 1 and 2 are dormant. All order constraints initially start with frontier labels  $z$  (indicating zero instances seen).

8	PRED a in 0	8	0000000	0000000	1111111	a	$[b^1, c^2, g^3]$
	$[c1 < c2]$			$[0 \rightarrow [g:z \ll c:z, d:z < x:z]]$			
	$[1 \rightarrow [e:z < f:z], 2 \rightarrow [h:z < i:z]]$			[]			

Predicting from edge 8 does not open a new domain, so the active and dormant domain constraints are simply inherited from the mother edge.

9	PRED b in 8	9	0000000	0000000	0000000	b	[d <sup>1</sup> ]
	[]			[0→[g:z ≪ c:z, d:z < x:z]]			
	[1→[e:z < f:z], 2→[h:z < i:z]]			[]			
10	PRED d in 9	10	0000000	0000000	0000000	d	[e <sup>1</sup> , f <sup>2</sup> ]
	[]			[0→[g:z ≪ c:z, d:z < x:z]]			
	[1→[e:z < f:z], 2→[h:z < i:z]]			[]			

Completing a category, the order constraints are updated to note the location of the category found; at this point, the right-most **e** found within the domain is at position  $2_{10} = 0000010_2$ .

11	COMP 10, 2	11	0000010	0000010	0000000	d	[f <sup>2</sup> ]
	[]			[0→[g:z ≪ c:z, d:z < x:z]]			
	[1→[e:2 < f:z], 2→[h:z < i:z]]			[]			

At this point, the parser completes edge 11 with the **f** in first position. This violates the dormant constraint in domain 1, which is now updated to read *void*; any future attempts to complete this edge into domain 1 will fail. As the parser completes further edges in the same domain from edge 11, each will be ‘tainted’ in the same way.

12	COMP 11, 1	12	0000011	0000011	0000000	d	[]
	[]			[0→[g:z ≪ c:z, d:z < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[]			
13	COMP 9, 12	13	0000011	0000011	0000000	b	[]
	[]			[0→[g:z ≪ c:z, d:3 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[]			
14	COMP 8, 13	14	0000011	0000011	1111111	a	[c <sup>2</sup> , g <sup>3</sup> ]
	[p3 < c2]			[0→[g:z ≪ c:z, d:3 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[]			

The parser now begins to work on finding the **c**.

15	PRED c in 14	15	0000000	0000011	0000000	c	[h <sup>1</sup> , d <sup>2</sup> , i <sup>3</sup> ]
	[]			[0→[g:z ≪ c:z, d:3 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[d:[2]:1, x:[1, 3]:2]			

Finding an **h** at sixth position yields a state in which only some of the information about the members of a domain is known; this is referred to as an ‘in-progress isolation statement’. Here, the statement indicates that the location of constituent 3, COMBINED with the position vector 0100000, will form a contiguous unit. In-progress isolation statements need to store their own sets of active and dormant order constraints, since the active and dormant constraints on the edge itself are for the LHS category’s domain. Thus **h** is listed as as-yet unseen on the edge itself, but has the frontier value of  $32_{10} = 0100000_2$  inside the in-progress isolation statement.

16	COMP 15, 6	16	0100000	0100011	0000000	c	[d <sup>2</sup> , i <sup>3</sup> ]
	[]			[0→[g:z ≪ c:z, d:3 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[d:[2]:1, [x:[3]+0100000, [2→[h:32 < i:z]]], [0→[g:z ≪ c:z, d:z < x:z], 1→[e:z < f:z]]]			

Predicting the **d** in edge 16 moves the parser into a new domain, so the active and dormant constraints are re-initialized; now the constraint for domain 1 is active and those for domains 0 and 2 are dormant.

17	PRED d in 16	17	0000000	0100011	0000000	d	[e <sup>1</sup> , f <sup>2</sup> ]
	[]			[1→[e:z < f:z]]			
	[0→[g:z << c:z, d:z < x:z], 2→[h:z < i:z]]			[]			
18	COMP 17, 4	18	0001000	0101011	0000000	d	[f <sup>2</sup> ]
	[]			[1→[e:8 < f:z]]			
	[0→[g:z << c:z, d:z < x:z], 2→[h:z < i:z]]			[]			
19	COMP 18, 5	19	0011000	0011000	0000000	d	[]
	[]			[1→[e:8 < f:16]]			
	[0→[g:z << c:z, d:z < x:z], 2→[h:z < i:z]]			[]			

Even though this **d** was originally predicted as part of the process of locating a **c**, it will first be picked up by edge 9. Note that in contrast to the handling of rule-level constraints, the constraint in domain 1 remains a part of the edge. This arises from the fact that there are no limits on how often a description may be satisfied in any domain.

20	COMP 9, 19	20	0011000	0011000	0000000	b	[]
	[]			[0→[g:z << c:z, d:24 < x:z]]			
	[1→[e:8 < f:16], 2→[h:z < i:z]]			[]			
21	COMP 8, 20	21	0011000	0011000	1111111	a	[c <sup>2</sup> , g <sup>3</sup> ]
	[p24 < c2]			[0→[g:z << c:z, d:24 < x:z]]			
	[1→[e:8 < f:16], 2→[h:z < i:z]]			[]			
22	PRED c in 21 fails – no room for 3 elements in 0011111						

At this point, the parser can complete the **d** it found into edge 16. Notice that the frontier value of **d** has been updated from 3<sub>10</sub> to 24<sub>10</sub> – the fact that **d** forms a domain only means that its constituents are not visible to higher constraints.

23	COMP 16, 19	22	0111000	0111011	0000000	c	[r <sup>3</sup> ]
	[]			[0→[g:z << c:z, d:24 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[[x:[3]+0100000, [2→[h:32 < i:z]], [0→[g:z << c:z, d:z < x:z], 1→[e:z < f:z]]]			

Finding the **i** at position 64<sub>10</sub> finishes the in-progress isolation statement: the position vector 1100000 is indeed contiguous, and 32 < 64, so domain 2's constraints are satisfied. The resulting edge is updated to indicate an **x** found at position 96<sub>10</sub>.

24	COMP 22, 7	23	1111000	1111000	0000000	c	[]
	[]			[0→[g:z << c:z, d:24 < x:96]]			
	[1→[void], 2→[h:z < i:z]]			[]			

The **c** located, the original prediction can be advanced. Note the form of the updated immediate precedence statement on edge 24, indicating that (o)ne **c** has been found at 120<sub>10</sub>.

25	COMP 14, 23	24	1111011	1111011	1111111	a	[g <sup>3</sup> ]
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:o(120), d:24 < x:96]]			
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:o(120), d:24 < x:96]]			

After completing with the remaining **g**, the success edge is obtained. The remainder of the parse investigates the remaining choicepoints to see if any other analyses are possible.

26	COMP 24, 3	25	1111111	1111111	0000000	a	[ ]
	[1→[void], 2→[h:z < i:z]]			[0→[g:o(4) ≪ c:o(120), d:24 < x:96]]			
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:24 < x:96]]			
27	COMP 11, 5	26	0010010	0010010	0000000	d	[ ]
	[1→[e:2 < f:16], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:24 < x:96]]			
	[1→[e:2 < f:16], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:24 < x:96]]			
28	COMP 9, 26	27	0010010	0010010	0000000	b	[ ]
	[1→[e:2 < f:16], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:18 < x:z]]			
	[1→[e:2 < f:16], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:18 < x:z]]			
29	COMP 8, 27	28	0010010	0010010	1111111	a	[c <sup>2</sup> , g <sup>3</sup> ]
	[p18 < c2]			[0→[g:z ≪ c:z, d:18 < x:z]]			
	[1→[e:2 < f:16], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:18 < x:z]]			
30	PRED c in 28 fails – no room for 3 elements in 0011111						
31	COMP 10, 4	29	0001000	0001000	0000000	d	[f <sup>2</sup> ]
	[1→[e:8 < f:z], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
	[1→[e:8 < f:z], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
32	COMP 29, 1	30	0001001	0001001	0000000	d	[ ]
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
33	COMP 9, 30	31	0001001	0001001	0000000	b	[ ]
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
34	COMP 8, 31	32	0001001	0001001	1111111	a	[c <sup>2</sup> , g <sup>3</sup> ]
	[p9 < c2]			[0→[g:z ≪ c:z, d:9 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
35	PRED c in 32	33	0000000	0001111	0000000	c	[h <sup>1</sup> , d <sup>2</sup> , i <sup>3</sup> ]
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[d:2]:1, x:[1, 3]:2			
36	COMP 33, 6	34	0100000	0101111	0000000	c	[d <sup>2</sup> , i <sup>3</sup> ]
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:9 < x:z]]			
	[1→[void], 2→[h:z < i:z]]			[d:2]:1, [x:3]+0100000, [2→[h:32 < i:z]], [0→[g:z ≪ c:z, d:z < x:z], 1→[e:z < f:z]]]			
	[1→[void], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:z < x:z], 1→[e:z < f:z]]]			
37	PRED d in 34	35	0000000	0101111	0000000	d	[e <sup>1</sup> , f <sup>2</sup> ]
	[1→[e:z < f:z]]			[1→[e:z < f:z]]			
	[0→[g:z ≪ c:z, d:z < x:z], 2→[h:z < i:z]]			[1→[e:z < f:z]]			
	[0→[g:z ≪ c:z, d:z < x:z], 2→[h:z < i:z]]			[1→[e:z < f:z]]			
38	COMP 29, 5	36	0011000	0011000	0000000	d	[ ]
	[1→[e:8 < f:16], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:z < x:z]]			
	[1→[e:8 < f:16], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:z < x:z]]			
	[1→[e:8 < f:16], 2→[h:z < i:z]]			[0→[g:z ≪ c:z, d:z < x:z]]			
39	COMP 9, 36 is already in the chart as edge 20						
40	COMP 16, 36 is already in the chart as edge 22						
41	SUCCESS: 25						

## 6 Evaluation

In general, a new parsing algorithm should contribute to the state of the art in either (or both) of two ways: it is more efficient than previous approaches, or it allows grammars to express concepts more transparently. The paper now addresses each of these areas in turn.<sup>17</sup>

### 6.1 Efficiency

Suhre (1999) shows that the membership problem for a formally-equivalent grammar formalism is NP-complete, both when considering the grammar plus the string as input (general membership problem) as well as when only the string is considered as input (fixed membership problem). It has been known since Huynh (1983) that the general membership problem for unordered context-free grammars (ID/LP grammars without LP statements) is also NP-complete, so Suhre's first result is not surprising. That the fixed membership problem for GIDL grammar is also NP-complete is less straightforward; fortunately, Suhre (1999, 61ff) demonstrates that it stems from the potential for recursive growth of discontinuities. As a result, when the parser can assume an upper bound on the number of discontinuities in any given constituent, the fixed membership problem becomes polynomial. Formally, this can be achieved by requiring that the number of discontinuities introduced by a recursive non-terminal is bounded by some constant.

Interestingly, a related practical proposal based on linguistic argumentation is discussed by Müller (1999b). He proposes a continuity constraint for linearization-based HPSG which requires saturated phrasal elements (that is, maximal projections) to be continuous.<sup>18</sup> Müller shows that adding his continuity constraint results in a significant reduction in the number of passive edges and thereby significant improvements in parsing performance.

This continuity constraint is weaker than Suhre's condition in that recursion on the level of adjunction is not restricted. It is, however, interesting to note in this context that a grammar incorporating the  $\bar{X}$ -schema (Jackendoff 1977) will require all non-head constituents to be maximal projections. In sum, Müller's result strongly suggests that further research on linguistically-motivated continuity constraints can result in efficient parsing of those GIDL grammars which include such constraints.

This raises the question of how the parsing algorithm proposed in this paper performs when used to parse grammars incorporating linear precedence and isolation constraints (since the worst-case performance results are based on the absence of such constraints). As mentioned earlier, the GIDL grammars form a superset of the context-free grammars. Thus it would be desirable for a GIDL parser to be just as efficient as a context-free parser when presented with a context-free grammar encoded in the GIDL format.

To investigate this, the parser's performance has been tested with the three types of context-free grammars discussed in Earley (1970) – those that require linear, quadratic, and cubic

---

<sup>17</sup> An interesting point of reference is the ID/LP parsing literature. Volk (1996) showed that in terms of efficiency and expressivity, it is advantageous to be able to combine ID/LP and ordinary context-free rules in one grammar. While this paper has focused on the issue of discontinuity, the GIDL parsing algorithm does seamlessly integrate context-free and (G)ID/LP rules.

<sup>18</sup> If extraposition is handled via discontinuous constituents, a more complex constraint is required.

time for strings to be recognized; following Earley, the number of edge insertion attempts (whether successful or unsuccessful) is used as a metric (represented as **cost** in the charts that follow).

Earley uses the context-free grammar in (17) (presented here with its GIDL P equivalent) to test the linear and quadratic aspects of the algorithm.

- (17)  $\text{root}(X)$              $\text{root}(X, [])$   
 $X \rightarrow A$              $X \rightarrow A^1 ; ; \langle [1], [], A \rangle$   
 $X \rightarrow X B$          $X \rightarrow X^1 B^2 ; 1 \ll 2 ; \langle [1], [], X \rangle, \langle [2], [], B \rangle$   
 $X \rightarrow Y A$          $X \rightarrow Y^1 A^2 ; 1 \ll 2 ; \langle [1], [], Y \rangle, \langle [2], [], A \rangle$   
 $Y \rightarrow E$              $Y \rightarrow E^1 ; ; \langle [1], [], E \rangle$   
 $Y \rightarrow Y D Y$      $Y \rightarrow Y^1 D^2 Y^3 ; 1 \ll 2, 2 \ll 3 ; \langle [1], [], Y \rangle, \langle [2], [], D \rangle, \langle [3], [], Y \rangle$

Consider the input string  $(\mathbf{ed})^x \mathbf{eab}^y$  (where  $a^x$  abbreviates  $x$  copies of  $a$ ). With this grammar, Earley reports that the number of edge insertion attempts for his algorithm increases linearly with  $y$  and quadratically with  $x$ .

With the linear case, a constant amount of additional work is expected for each additional character. The results below show that this is obtained by the GIDL P parser.

Input	Cost	Size	$\Delta\text{Cost}$	$\Delta\text{Size}$	$\Delta\text{Cost}/\Delta\text{Size}$
<b>ededeab</b>	97	1	n/a	n/a	n/a
<b>ededeab<sup>2</sup></b>	110	2	1	13	13
<b>ededeab<sup>3</sup></b>	123	3	2	26	13
<b>ededeab<sup>5</sup></b>	149	5	4	52	13
<b>ededeab<sup>10</sup></b>	214	10	9	117	13

In these charts, size refers to the length of the relevant portion of the input string (here, the number of **bs**); all  $\Delta$  values are computed with respect to the first line of the chart.

For the quadratic case, the additional work required should be proportional to the difference of the squares of the input length. The data below diverge, indicating that this performance goal has not yet been met.

Input	Cost	Size <sup>2</sup>	$\Delta\text{Cost}$	$\Delta(\text{Size}^2)$	$\Delta\text{Cost}/\Delta(\text{Size}^2)$
<b>(ed)<sup>2</sup>ea</b>	74	4	n/a	n/a	n/a
<b>(ed)<sup>3</sup>ea</b>	132	9	58	5	11.6
<b>(ed)<sup>4</sup>ea</b>	220	16	146	12	12.2
<b>(ed)<sup>5</sup>ea</b>	347	25	273	21	13.0
<b>(ed)<sup>6</sup>ea</b>	523	36	449	32	14.0

The simplest cubic-time grammar is

- $B \rightarrow B B$   
 $B \rightarrow A$

The corresponding results show a similar divergence.

Input	Cost	Size <sup>3</sup>	$\Delta$ Cost	$\Delta$ (Size <sup>3</sup> )	$\Delta$ Cost/ $\Delta$ (Size <sup>3</sup> )
A <sup>5</sup>	105	125	n/a	n/a	n/a
A <sup>6</sup>	182	216	77	91	0.85
A <sup>7</sup>	294	343	189	218	0.87
A <sup>8</sup>	450	512	345	387	0.89
A <sup>10</sup>	935	1000	830	875	0.94

## 6.2 Natural Description of Efficient Grammars

Recall that in the GIDL grammar format defined in section 2.1, the order of the RHS of a grammar rule does not encode the terminal order of the daughters. Instead, it expresses the order in which the parser will search for these elements, as discussed in section 3.4.

To see why this is valuable, consider a grammar covering raising verbs in Icelandic. Many verbs in Icelandic assign “quirky case” (i.e. non-nominative) to their subjects; these case assignments persist when the subject is raised to be the subject or object of a matrix verb. From a parsing perspective, the embedded verb must be known before it can be determined whether a given noun phrase is an acceptable subject for the matrix verb. This is illustrated by the data in (18) – (23).

- (18) Hana virðist vanta peninga  
her.ACC seems to-lack money  
‘She seems to lack money.’
- (19) Barninu virðist hafa batnað veikin  
the-child.DAT seems to-have recovered-from the-disease  
‘The child seems to have recovered from the disease.’
- (20) Verkjanna virðist ekki gæta  
the-pains.GEN seem not to-be-noticeable  
‘The pains don’t seem to be noticeable.’
- (21) Hann telur mig vanta peninga  
he.NOM believes me.ACC to-lack money  
‘He believes that I lack money.’
- (22) Hann telur barninu hafa batnað veikin  
he believes the-child.DAT to-have recovered-from the-disease  
‘He believes the child to have recovered from the disease.’
- (23) Hann telur verkjanna ekki gæta  
he believes the-pains.GEN not to-be-noticeable  
‘He believes the pains to be not noticeable.’

In other words, the fact that the subject in (18) and (21) is accusative is a reflection of the embedded verb ‘lack’ rather than the matrix verbs ‘seem’ or ‘believe’; the same situation holds for the dative [(19), (22)] and genitive [(20), (23)] examples. In all other respects, however, the matrix verb is still the head of its clause (it must agree in number with the subject, for example).

Consider a head-driven parser (van Noord 1997): a variant of a phrase-structure parser in which a designated element (the head) is parsed before any other complement; the non-head daughters are then parsed in the usual left-to-right order. With such a parser, the grammar writer would write a rule like (24) to license the matrix clause.

$$(24) S \rightarrow NP_{\text{subj}} V^{\text{head}} VP_{\text{inf}}$$

With such a rule, the parser will first locate the head (here, the **V**), then the **NP**, and finally the **VP**. As a consequence, the constraints in the **VP** on the case of the subject will not be known until after the subject has been found. The parser will therefore try all possible **NPs** as subjects, and then see which the embedded verb phrase rejects.

With the GIDL P formalism, in contrast, the grammar writer could specify the rule as (25) to avoid this generate-and-test pattern.

$$(25) S \rightarrow V^1 VP_{\text{inf}}^2 NP_{\text{subj}}^3$$

Now the parser will not look for the subject of the clause until the embedded verb phrase has been located, and so only **NPs** with the appropriate case will even be considered.

## 7 Future Work

To advance towards the general goal of efficient parsing with linearization-based HPSG grammars, the next step is to replace atomic categories with complex categories, encoded by typed feature structures.

This move brings with it a number of complications. Apart from the well-known issues generally involved in adding complex categories to a chart-parser – for example, subsumption checking or the use of a restrictor (see, for instance, Shieber 1985; Pereira and Shieber 1987; Gerdemann 1991) – the most interesting challenge in this context is the observation by Seiffert (1991) on ID/LP parsing that word order constraints cannot always be verified when a local domain is constructed. While Seiffert addresses the issue by checking word order constraints in a second pass once the entire tree has been constructed, Morawietz (1995) shows that by explicitly retaining the relevant information, this second pass can be avoided. By making use of the co-routining capabilities of SICStus Prolog, however, it should be possible to retain the relevant information implicitly. It may also be possible to adapt the dormant constraint mechanism so that no more information is retained than necessary.

The move to complex categories also brings with it the opportunity for more practical evaluation metrics. It is generally accepted, for instance, that the dominating factor in feature structure-based algorithms is the number of unifications that must be performed; such a metric is easily calculated once one has a grammar which can be used for testing. One future offshoot of this project is the recoding of the linearization-based Babel grammar (Müller 1996), one of the most comprehensive grammars of German, as a GIDL P grammar so it can be used as a test case for the extended parser. This should allow the substantiation (or refutation) of the claim that processing comprehensive linearization grammars of natural languages is efficient once all available word order constraints are used to guide processing in a well-engineered way.

## 8 Summary

This paper has described a number of optimizations for parsing with a formalism licensing discontinuous constituents. Using bitvectors encoded as integers to model subsets of the terminal yield, the required bitvector operations can be computed in constant time. To efficiently access edges and rules in a way that makes use of word order information, two kinds of bitmasks are used to constrain possible coverage vectors, specifying the positions that can, must, and must not be covered by an edge. The algorithm can thereby take order constraints into account in a more interleaved fashion, restricting the search space of the parser.

## References

- Blevins, James (1990). Syntactic Complexity: Evidence for Discontinuity and Multidomination. Ph.D. thesis, University of Massachusetts, Amherst, MA.
- Bonami, Olivier, Danièle Godard and Jean-Marie Marandin (1999). Constituency and word order in French subject inversion. In Gosse Bouma, Erhard W. Hinrichs, Geert-Jan M. Kruijff and Richard T. Oehrle (eds.), *Constraints and Resources in Natural Language Syntax and Semantics*, Stanford, CA: CSLI Publications, Studies in Constraint-Based Lexicalism, pp. 21–40.
- Brew, Chris (1992). Letting the cat out of the bag: generation for shake-and-bake MT. In *Proceedings of the 14th International Conference on Computational Linguistics*. Nantes, France, pp. 610–616. Available from <http://arxiv.org/abs/cmp-1g/9511002>.
- Bröker, Norbert (1998). Separating Surface Order and Syntactic Relations in a Dependency Grammar. In COLING-ACL (1998), pp. 174–180.
- Bunt, Harry and Arthur van Horck (eds.) (1996). *Discontinuous Constituency*, vol. 6 of *Natural Language Processing*. Berlin and New York, NY: Mouton De Gruyter.
- COLING-ACL (1998). *Proceedings of the 17th International Conference on Computational Linguistics (COLING) and the 36th Annual meeting of the ACL (ACL)*, Montreal.
- Covington, Michael A. (1990). Parsing discontinuous constituents in dependency grammar. *Computational Linguistics* 16(4), 234–236.
- Covington, Michael A. (1992). A dependency parser for variable-word-order languages. In K. R. Billingsley, Hilton U. Brown III and Ed Derohanes (eds.), *Computer assisted modeling on the IBM 3090: Papers from the 1989 IBM Supercomputing Competition*, Athens, GA: Baldwin Press, vol. 2, pp. 799–845.
- Daniels, Michael W. and W. Detmar Meurers (2002). Improving the efficiency of parsing with discontinuous constituents. In Shuly Wintner (ed.), *Proceedings of NLULP-02: The Seventh International Workshop on Natural Language Understanding and Logic Programming*. Roskilde University, Computer Science Department, Copenhagen, Denmark, pp. 49–68.
- Davis, Paul C. (2002). Stone Soup Translation: The Linked Automata Model. Ph.D. thesis, Ohio State University, Columbus, OH.
- Donohue, Cathryn and Ivan A. Sag (1999). Domains in Warlpiri. In *Abstracts of the Sixth Int. Conference on HPSG*. Edinburgh: University of Edinburgh, pp. 101–106. <http://www-csli.stanford.edu/~sag/papers/warlpiri.ps>.

- Dowty, David R. (1996). Towards a Minimalist Theory of Syntactic Structure. In Bunt and van Horck (1996).
- Earley, Jay (1970). An Efficient Context-Free Parsing Algorithm. *Communications of the ACM* 13(2), 94–102. Also in Grosz et al. (1986).
- Gazdar, Gerald, Ewan Klein, Geoffrey K. Pullum and Ivan A. Sag (1985). *Generalized Phrase Structure Grammar*. Cambridge, MA: Harvard University Press.
- Gerdemann, Dale (1991). *Parsing and Generation of Unification Grammars*. Tech. Rep. CS-91-06, Beckman Institute, University of Illinois.
- Götz, Thilo and Gerald Penn (1997). *A Proposed Linear Specification Language*. Volume 134 in Arbeitspapiere des SFB 340., Universität Tübingen. <http://www.sfs.uni-tuebingen.de/sfb/reports/berichte/134/134abs.html>.
- Grosz, Barbara, Karen Sparck Jones and Bonnie Lynn Webber (eds.) (1986). *Readings in Natural Language Processing*. Los Altos, CA: Morgan Kaufmann.
- Hepple, Mark (1994). Discontinuity and the Lambek Calculus. In *Proceedings of the 15th Conference on Computational Linguistics (COLING-94)*. Kyoto. Available from <ftp://ftp.dcs.shef.ac.uk/home/hepple/papers/coling94.ps>.
- Hinrichs, Erhard, Julia Bartels, Yasuhiro Kawata, Valia Kordoni and Heike Telljohann (2000). The Tübingen Treebanks for Spoken German, English, and Japanese. In Wolfgang Wahlster (ed.), *Verbmobil: Foundations of Speech-to-Speech Translation*, Berlin: Springer, pp. 552–576.
- Huck, Geoffrey (1985). Exclusivity and discontinuity in phrase structure grammar. In *West Coast Conference on Formal Linguistics (WCCFL)*. Stanford University, CSLI Publications, vol. 4, pp. 92–98.
- Huck, Geoffrey and Almerindo Ojeda (eds.) (1987). *Discontinuous Constituency*. No. 20 in Syntax and Semantics. New York: Academic Press.
- Huynh, Dung T. (1983). Commutative Grammars: The Complexity of Uniform Word Problems. *Information and Control* 57(1), 21–39.
- Jackendoff, Ray (1977). *X-Bar Syntax: A Study of Phrase Structure*. Cambridge, Mass.: MIT Press.
- Johnson, Mark (1985). Parsing with discontinuous constituents. In *Proceedings of the 23rd Annual Meeting of the ACL*, Chicago, pp. 127–132.
- Kasami, T. and K. Torii (1969). A syntax-analysis procedure for unambiguous context-free grammars. *Journal of the Association for Computing Machinery* 16(3), 423–431.
- Kasper, Robert T., Mike Calcagno and Paul C. Davis (1998). Know When to Hold 'Em: Shuffling Deterministically in a Parser for Nonconcatenative Grammars. In *COLING-ACL (1998)*, pp. 663–669.
- Kathol, Andreas (1995). Linearization-Based German Syntax. Ph.D. thesis, The Ohio State University.
- Kathol, Andreas and Carl Pollard (1995). Extraposition via Complex Domain Formation. In *Proceedings of the 1995 Annual Meeting of the Association for Computational Linguistics*. pp. 174–180. Available from <http://www.linguistics.berkeley.edu/~kathol/papers.html>.

- Kay, Martin (1980). Algorithm schemata and data structures in syntactic processing. In Grosz et al. (1986), pp. 35–70.
- Kay, Martin (1990). Head-Driven Parsing. In Masaru Tomita (ed.), *Current Issues in Parsing Technology*, Dordrecht: Kluwer Academic Publishers. Previously published in the proceedings of the International Workshop on Parsing Technologies, 1989.
- Knuth, Donald E. (2002). Generating all  $n$ -tuples. Pre-fascicle 2A (revision 8) of Volume 4 of *The Art of Computer Programming*. Available from <http://sunburn.stanford.edu/~knuth/fasc2a.ps.gz>.
- Kroch, Anthony S. and Aravind K. Joshi (1987). Analyzing Extraposition in a Tree Adjoining Grammar. In Huck and Ojeda (1987).
- Lenerz, Jürgen (2001). Word Order Variation: Competition or Co-Operation. In Gereon Müller and Wolfgang Sternefeld (eds.), *Competition in Syntax*, Berlin and New York, NY: Mouton De Gruyter, pp. 249–281.
- McCawley, James D. (1982). Parentheticals and discontinuous constituent structure. *Linguistic Inquiry* 13(1), 91–106.
- Morawietz, Frank (1995). *Formalization and Parsing of Unification-Based ID/LP Grammars*. Arbeitspapiere des SFB 340. Nr. 68, Universität Tübingen. Available from <http://www.sfs.uni-tuebingen.de/sfb/reports/berichte/68/68abs.html>.
- Morrill, Glynn V. (1995). Discontinuity in categorial grammar. *Linguistics and Philosophy* 18, 175–219.
- Müller, Stefan (1996). The Babel-System – An HPSG Prolog Implementation. In *Proceedings of the Fourth International Conference on the Practical Application of Prolog*, London, pp. 263–277. Revised version available at <http://www.dfki.de/~stefan/Pub/babel.html>.
- Müller, Stefan (1999a). *Deutsche Syntax deklarativ. Head-Driven Phrase Structure Grammar für das Deutsche*. No. 394 in *Linguistische Arbeiten*. Tübingen: Max Niemeyer Verlag.
- Müller, Stefan (1999b). *Restricting Discontinuity*. Verbmobil Report 237, DFKI, Saarbrücken. Also published in the Proceedings of GLDV 99 (Frankfurt/Main). Available from [http://www.dfki.de/~stefan/Pub/e\\_restricting.html](http://www.dfki.de/~stefan/Pub/e_restricting.html).
- Müller, Stefan (2003). Continuous or Discontinuous Constituents? A Comparison between Syntactic Analyses for Constituent Order and Their Processing Systems. *Language and Computation* To appear. <http://www.dfki.de/~stefan/Pub/discont.html>.
- Ojeda, Almerindo (1987). Discontinuity, multidominances and unbounded dependency in Generalized Phrase Structure Grammar. In Huck and Ojeda (1987).
- Penn, Gerald (1999). Linearization and WH-extraction in HPSG: Evidence from Serbo-Croatian. In Robert D. Borsley and Adam Przepiórkowski (eds.), *Slavic in HPSG*, Stanford, CA: CSLI Publications, pp. 149–182.
- Pereira, Fernando and Stuart Shieber (1987). *Prolog and Natural-Language Analysis*. CSLI Lecture Notes. CSLI.

- Plátek, Martin, Tomáš Holan, Vladimír Kuboň and Karel Oliva (2001). Word-Order Relaxations and Restrictions within a Dependency Grammar. In G. Satta (ed.), *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT)*. Beijing: Tsinghua University Press, pp. 237–240.
- Pollard, Carl and Ivan Sag (1994). *Head-Driven Phrase Structure Grammar*. Chicago: University of Chicago Press.
- Rambow, Owen and Aravind Joshi (1994). A Formal Look at Dependency Grammars and Phrase-Structure Grammars, with Special Consideration of Word-Order Phenomena. In L. Wanner (ed.), *Current Issues in Meaning-Text-Theory*, London: Pinter. Available from <http://arxiv.org/abs/cmp-lg/9410007>.
- Ramsay, Allan M. (1999). Direct parsing with discontinuous phrases. *Natural Language Engineering* 5(3), 271–300.
- Reape, Mike (1989). A logical treatment of semi-free word order and bounded discontinuous constituency. In *Proceedings of the Fourth Meeting of the European Association for Computational Linguistics*. pp. 103–110.
- Reape, Mike (1990). A Theory of Word Order and Discontinuous Constituency in West Continental Germanic. In Elisabeth Engdahl and Mike Reape (eds.), *Parametric Variation in Germanic and Romance: Preliminary Investigations*, Edinburgh: Centre for Cognitive Science, University of Edinburgh, DYANA Deliverable R1.1.A, ESPRIT Basic Research Action BR 3175, pp. 25–39.
- Reape, Mike (1991a). Parsing Bounded Discontinuous Constituents: Generalisations of some common algorithms. In Reape (1991b), pp. 41–70.
- Reape, Mike (ed.) (1991b). *Word Order in Germanic and Parsing*. DYANA Deliverable R1.1.C, ESPRIT Basic Research Action BR 3175. Centre for Cognitive Science, University of Edinburgh.
- Reape, Mike (1993). A Formal Theory of Word Order: A Case Study in West Germanic. PhD thesis., University of Edinburgh.
- Reape, Mike (1994). Domain Union and Word Order Variation in German. In John Nerbonne, Klaus Netter and Carl Pollard (eds.), *German in Head-Driven Phrase Structure Grammar*, Stanford, CA: CSLI Publications, no. 46 in CSLI Lecture Notes, pp. 151–197.
- Reape, Mike (1996). Getting things in order. In Bunt and van Horck (1996), pp. 209–253. Published version of a Ms. from 1990.
- Richter, Frank and Manfred Sailer (2001). On the Left Periphery of German Finite Sentences. In W. Detmar Meurers and Tibor Kiss (eds.), *Constraint-Based Approaches to Germanic Syntax*, Stanford, CA: CSLI Publications, Studies in Constraint-Based Lexicalism, pp. 257–300.
- Seiffert, Roland (1991). Unification–ID/LP Grammars: Formalization and Parsing. In Otthein Herzog and Claus-Rolf Rollinger (eds.), *Text Understanding in LILOG*, Berlin: Springer Verlag, no. 546 in Lecture Notes in Artificial Intelligence, pp. 63–73.
- Shieber, Stuart (1985). Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proceedings of the 23rd Annual Meeting of the ACL*, Chicago, pp. 145–52.
- Shieber, Stuart M. (1984). Direct Parsing of ID/LP Grammars. *Linguistics and Philosophy* 7, 135–154.

- Skut, Wojciech, Brigitte Krenn, Thorsten Brants and Hans Uszkoreit (1997). An Annotation Scheme for Free Word Order Languages. In *Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP)*. Washington, D.C. Available from <http://www.coli.uni-sb.de/~thorsten/publications/Skut-ea-ANLP97.ps.gz>.
- Suhre, Oliver (1999). Computational Aspects of a Grammar Formalism for Languages with Freer Word Order. Diplomarbeit., Department of Computer Science, University of Tübingen. Published 2000 as Volume 154 in Arbeitspapiere des SFB 340, <http://www.sfs.uni-tuebingen.de/sfb/reports/berichte/154/154abs.html>.
- van Noord, Gertjan (1991). Head Corner Parsing for Discontinuous Constituency. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*. pp. 114–121.
- van Noord, Gertjan (1997). An Efficient Implementation of the Head-Corner Parser. *Computational Linguistics* 23(3).
- Volk, Martin (1996). Parsing with ID/LP and PS rules. In *Natural Language Processing and Speech Technology. Results of the 3rd KONVENS Conference (Bielefeld)*, Berlin: Mouton de Gruyter, pp. 342–353.
- Yatabe, Shuichi (1996). Long-distance scrambling via Partial Compaction. In Masatoshi Koizumi, Masayuki Oishi and Uli Sauerland (eds.), *Formal Approaches to Japanese Linguistics 2*, Cambridge, MA: MITWPL, pp. 303–317. Available from <http://gamp.c.u-tokyo.ac.jp/~yatabe/fajl.pdf>.
- Younger, D. H. (1967). Recognition and parsing of context-free languages in time  $n^3$ . *Information and Control* 10, 189–208.