Dependency Parsing Schemata and Mildly Non-Projective Dependency Parsing

Carlos Gómez-Rodríguez*
Universidade da Coruña, Spain

John Carroll**
University of Sussex, UK

David Weir†
University of Sussex, UK

We introduce dependency parsing schemata, a formal framework based on Sikkel's parsing schemata for constituency parsers, which can be used to describe, analyze, and compare dependency parsing algorithms. We use this framework to describe several well-known projective and non-projective dependency parsers, build correctness proofs, and establish formal relationships between them. We then use the framework to define new polynomial-time parsing algorithms for various mildly non-projective dependency formalisms, including well-nested structures with their gap degree bounded by a constant k *in time* $O(n^{5+2k})$*, and a new class that includes all gap degree* k *structures present in several natural language treebanks (which we call mildly ill-nested structures for gap degree* k*) in time* $O(n^{4+3k})$*. Finally, we illustrate how the parsing schema framework can be applied to Link Grammar, a dependency-related formalism.*

1. Introduction

Dependency parsing involves finding the structure of a sentence as expressed by a set of directed links (called **dependencies**) between individual words. Dependency formalisms have attracted considerable interest in recent years, having been successfully applied to tasks such as machine translation (Ding and Palmer 2005; Shen, Xu, and Weischedel 2008), textual entailment recognition (Herrera, Peñas, and Verdejo 2005), relation extraction (Culotta and Sorensen 2004; Fundel, Küffner, and Zimmer 2006), and question answering (Cui et al. 2005). Key characteristics of the dependency parsing approach are that dependency structures specify head–modifier and head–complement relationships, which form the basis of predicate–argument structure, but are not represented explicitly in constituency trees; there is no need for dependency parsers to postulate the existence of non-lexical nodes; and some variants of dependency parsers

* Facultade de Informática, Universidade da Coruña Campus de Elviña, s/n, 15071 A Coruña, Spain. E-mail: cgomezr@udc.es.
** School of Informatics, University of Sussex, Falmer, Brighton BN1 9QJ, UK. E-mail: J.A.Carroll@sussex.ac.uk.
† School of Informatics, University of Sussex, Falmer, Brighton BN1 9QJ, UK. E-mail: D.J.Weir@sussex.ac.uk.

Submission received: 21 October 2009; revised submission received: 23 December 2010; accepted for publication: 29 January 2011.

are able to represent non-projective structures (McDonald et al. 2005), which is important when parsing free word order languages where discontinuous constituents are common.

The formalism of parsing schemata, introduced by Sikkel (1997), is a useful tool for the study of constituency parsers, supporting precise, high-level descriptions of parsing algorithms. Potential applications of parsing schemata include devising correctness proofs, extending our understanding of relationships between different algorithms, deriving new variants of existing algorithms, and obtaining efficient implementations automatically (Gómez-Rodríguez, Vilares, and Alonso 2009). The formalism was originally defined for context-free grammars (CFG) and since then has been applied to other constituency-based formalisms, such as tree-adjoining grammars (Alonso et al. 1999). This article considers the application of parsing schemata to the task of dependency parsing. The contributions of this article are as follows.

- We introduce **dependency parsing schemata**, a novel adaptation of the original parsing schemata framework (see Section 2).
- We use the dependency parsing schemata to define and compare a number of existing dependency parsers (projective parsers are presented in Section 3, and their formal properties discussed in Sections 4 and 5; a number of non-projective parsers are presented in Section 6).
- We present parsing algorithms for several sets of mildly non-projective dependency structures, including a parser for a new class of structures we call mildly ill-nested, which encompasses all the structures in a number of existing dependency treebanks (see Section 7).
- We adapt the dependency parsing schema framework to the formalism of Link Grammar (Sleator and Temperley 1991, 1993) (see Section 8).

Although some of these contributions have been published previously, this article presents them in a thorough and consistent way. The definition of dependency parsing schemata was first published by Gómez-Rodríguez, Carroll, and Weir (2008), along with some of the projective schemata presented here and their associated proofs. The results concerning mildly non-projective parsing in Section 7 were first published by Gómez-Rodríguez, Weir, and Carroll (2008, 2009). On the other hand, the material on Nivre and Covington's projective parsers, as well as all the non-projective parsers and the application of the formalism to Link Grammar, are entirely new contributions of this article.

The notion of a parsing schema comes from considering parsing as a deduction process which generates intermediate results called **items**. In particular, items in parsing schemata are sets of partial constituency trees taken from the set of all partial parse trees that do not violate the constraints imposed by a grammar. A parsing schema can be used to obtain a working implementation of a parser by using deductive engines such as the ones described by Shieber et al. (1995) and Gómez-Rodríguez, Vilares, and Alonso (2009), or the Dyna language (Eisner, Goldlust, and Smith 2005).

2. Dependency Parsing Schemata

Although parsing schemata were originally defined for CFG parsers, they have since been adapted to other constituency-based grammar formalisms. This involves finding

a suitable definition of the set of structures contained in items, and a way to define deduction steps that captures the formalism's composition rules (Alonso et al. 1999). Although it is less clear how to adapt parsing schemata to dependency parsing, a number of dependency parsers have the key property of being constructive: They proceed by combining smaller structures to form larger ones, terminating when a complete parse for the input sentence is found. We show that this makes it possible to define a variant of the traditional parsing schemata framework, where the encodings of intermediate dependency structures are defined as items, and the operations used to combine them are expressed as inference rules. We begin by addressing a number of preliminary issues.

Traditional parsing schemata are used to define grammar-driven parsers, in which the parsing process is guided by some set of rules which are used to license deduction steps. For example, an Earley PREDICTOR step is tied to a particular grammar rule, and can only be executed if such a rule exists. Some dependency parsers are also grammar-driven. For example, those described by Lombardo and Lesmo (1996), Barbero et al. (1998), and Kahane, Nasr, and Rambow (1998) are based on the formalizations of dependency grammar CFG-like rules described by Hays (1964) and Gaifman (1965). However, many of the algorithms (Eisner 1996; Yamada and Matsumoto 2003) are not traditionally considered to be grammar-driven, because they do not use an explicit formal grammar; decisions about which dependencies to create are taken individually, using probabilistic models (Eisner 1996) or classifiers (Yamada and Matsumoto 2003). These are called data-driven parsers. To express such algorithms as deduction systems, we use the notion of **D-rules** (Covington 1990). D-rules have the form $(a, i) \rightarrow (b, j)$, which specifies that a word b located at position j in the input string can have the word a in position i as a dependent. Deduction steps in data-driven parsers can be associated with the D-rules corresponding to the links they create, so that parsing schemata for such parsers are defined using grammars of D-rules. In this way, we obtain a representation of some of the declarative aspects of these parsing strategies that is independent of the particular model used to make the decisions associated with each D-rule. Note that this representation is useful for designing control structures or probabilistic models for the parsers, because it makes explicit the choice points where the models will have to make probabilistic decisions, as well as the information available at each of those choice points. Additionally, D-rules allow us to use an uniform description that is valid for both data-driven and grammar-driven parsers, because D-rules can function like grammatical rules.

The fundamental structures in dependency parsing are **dependency trees**. Therefore, just as items for constituency parsers encode sets of partial constituency trees, items for dependency parsers can be defined using partial dependency trees. However, dependency trees cannot express the fact that a particular structure has been predicted, but not yet built; this is required for grammar-based algorithms such as those of Lombardo and Lesmo (1996) and Kahane, Nasr, and Rambow (1998). The formalism can be made general enough to include these parsers by using a novel way of representing intermediate states of dependency parsers based on a form of dependency trees that include nodes labelled with preterminals and terminals (Gómez-Rodríguez, Carroll, and Weir 2008; Gómez-Rodríguez 2009). For simplicity of presentation, we will only use this representation (called **extended dependency trees**) in the grammar-based algorithms that need it, and we will define the formalism and the rest of the algorithms with simple dependency trees. Some existing dependency parsing algorithms, for example, the algorithm of Eisner (1996), involve steps that connect **spans** which can represent **disconnected** dependency graphs. Such spans cannot be represented by

a single dependency tree. Therefore, our formalism allows items to be sets of **forests** of partial dependency trees, rather than sets of trees.

We are now ready to define the concepts needed to specify item sets for dependency parsers.

Definition 1
An **interval** (with endpoints i and j) is a set of natural numbers of the form $[i..j] = \{k \mid i \leq k \leq j\}$. We will use the notation $i..j$ for the ordered list of the numbers in $[i..j]$. A **dependency graph** for a string $w = w_1 \dots w_n$ is a graph $G = (V, E)$, where $V \subseteq [1..n]$ and $E \subseteq V \times V$.

The edge (i, j) is written $i \rightarrow j$, and each such edge is called a **dependency link**, encoding the fact that the word w_i is a syntactic **dependent** (or **child**) of w_j or, conversely, that w_j is the **parent**, **governor**, or **head** of w_i. We write $i \rightarrow^\star j$ to denote that there exists a (possibly empty) path from i to j. The **projection** of a node i, denoted $\lfloor i \rfloor$, is the set of reflexive-transitive dependents of i, that is, $\lfloor i \rfloor = \{j \in V \mid j \rightarrow^\star i\}$. In contexts where we refer to different dependency graphs, we use the notation $\lfloor i \rfloor_G$ to specify the projection of a node i in the graph G.

Definition 2
A dependency graph T for a string $w_1 \dots w_n$ is called a **dependency tree** for that string if it contains no cycles and all of its nodes have exactly one parent, except for one node that has none and is called the **root** or **head** of the tree T, denoted $head(T)$. The yield of a dependency tree T, denoted $yield(T)$, is the ordered list of its nodes. We will use the term **dependency forest** to refer to a set of dependency trees for the same string,[1] and the generic term **dependency structure** to refer to a dependency tree or forest.

A dependency tree is said to be a **parse tree** for a string $w_1 \dots w_n$ if its yield is $1..n$.

Definition 3
We say that a dependency graph $G = (V, E)$ for a string $w_1 \dots w_n$ is **projective** if $\lfloor i \rfloor$ is an interval for every $i \in V$.

Definition 4
Let $\delta(G)$ be the set of dependency trees which are syntactically well-formed according to a given grammar G (which may be a grammar of D-rules or of CFG-like rules, as explained previously). We define an **item set** for dependency parsing as a set $\mathcal{I} \subseteq \Pi$, where Π is a partition of the power set, $\wp(\delta(G))$, of the set $\delta(G)$. Each element of $\mathcal{I}$, called an **item**, is a set of dependency forests for strings. For example, each member of the item $[1, 5]$ in the item set of the parser by Yamada and Matsumoto (2003) that will be explained in Section 3.4 is a dependency forest with two projective trees, one with head 1 and the other with head 5, and such that the concatenation of their yields is 1..5. Figure 1 shows the three dependency forests that constitute the contents of this item under a specific grammar of D-rules.

Following Sikkel (1997), items are sets of syntactic structures and tuples are a shorthand notation for such sets, as seen in the previous example. An alternative approach,

1 Note that the trees in a dependency forest can have different yields, because the node set of a dependency tree for a string $w_1 \dots w_n$ can be any subset of $[1..n]$. In fact, all the forests used by the parsers in this article contain trees with non-overlapping yields, although this is not required by the definition.

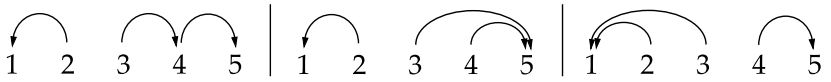

Figure 1
Contents of the item $[1, 5]$ from the Yamada and Matsumoto (2003) parsing schema under a grammar of D-rules $\{(w_2, 2) \rightarrow (w_1, 1)\ ,\ (w_3, 3) \rightarrow (w_1, 1)\ ,\ (w_3, 3) \rightarrow (w_5, 5)\ ,\ (w_3, 3) \rightarrow (w_4, 4)\ ,\ (w_4, 4) \rightarrow (w_5, 5)\}$.

following Shieber, Schabes, and Pereira (1995), would be to define items as tuples that *denote* sets of syntactic structures. Although the latter approach provides more flexibility, this makes defining the relationships between parsers less straightforward. In any case, because tuple notation is used to write schemata under both approaches, the schemata we provide are compatible with both interpretations.

Having defined an item set for dependency parsing, the remaining definitions are analogous to those in Sikkel's theory of constituency parsing (Sikkel 1997), and are not presented in full detail. A **dependency parsing system** is a deduction system $(\mathcal{I}, H, D)$ where $\mathcal{I}$ is a dependency item set as defined here, H is a set containing **initial items** or **hypotheses** (not necessarily contained in $\mathcal{I}$), and $D \subseteq (\wp(H \cup \mathcal{I}) \times \mathcal{I})$ is a set of **deduction steps** defining an inference relation $\vdash$.

Final items in this formalism will be those containing some forest F containing a parse tree for some string $w_1 \ldots w_n$. In parsers for general non-projective structures, any item containing such a tree will be called a **coherent final item** for $w_1 \ldots w_n$. In schemata for parsers that are constrained to a more restrictive class $\mathcal{T}$ of dependency trees, such as projective parsers, coherent final items will be those containing parse trees for $w_1 \ldots w_n$ that are in $\mathcal{T}$. For example, because we expect correct projective parsers to produce only projective structures, coherent final items for projective parsers will be those containing *projective* parse trees for $w_1 \ldots w_n$. Correctness proofs typically define a set of **coherent items**, such that its intersection with final items produces the set of coherent final items. The definition of coherent items depends on each particular proof.[2]

For each input string, a parsing schema's deduction steps allow us to infer a set of items, called **derivable items**, for that string.[3] A parsing schema is said to be **sound** if all derivable final items it produces for any arbitrary string are coherent for that string. A parsing schema is said to be **complete** if all coherent final items are derivable. A **correct parsing schema** is one which is both sound and complete.

Parsing schemata are located at a higher abstraction level than parsing algorithms, and formalize declarative aspects of their logic: A parsing schema specifies a set of intermediate results that are obtained by the algorithm (items) and a set of operations that can be used to obtain new such results from existing ones (deduction steps); but it makes no claim about the order in which to execute the operations or the data structures to use for storing the results.

3. Projective Schemata

In this section, we show how dependency parsing schemata can be used to describe several existing projective dependency parsers.

2 Coherent (final) items are called *correct (final) items* in the original formulation by Sikkel (1997).
3 Derivable items are called *valid items* in the original formulation by Sikkel (1997).

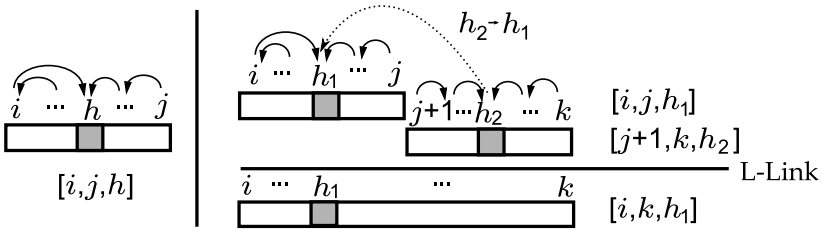

Figure 2
Representation of the $[i,j,h]$ item in Collins's parser, together with one of the dependency structures contained in it (left side); and of the antecedents and consequents of an L-LINK step (right side). White rectangles in an item represent intervals of nodes that have been assigned a head by the parser, and dark squares represent nodes that have no head.

3.1 Collins (1996)

One of the most straightforward projective dependency parsing strategies was introduced by Collins (1996), and is based on the CYK bottom–up parsing strategy (Kasami 1965; Younger 1967). Collins's parser works with dependency trees which are linked to each other by creating links between their heads. The schema for this parser maps every set of D-rules G and input string $w_1 \dots w_n$ to an instantiated dependency parsing system $(\mathcal{I}_{Col96}, \mathcal{H}, D_{Col96})$ such that:

Item set: The item set is defined as $\mathcal{I}_{Col96} = \{[i,j,h] \mid 1 \leq i \leq h \leq j \leq n\}$, where item $[i,j,h]$ is defined as the set of forests containing a single projective dependency tree T of $\delta(G)$ such that $yield(T)$ is of the form $i..j$ and $head(T) = h$ (see Figure 2, left side). From now on, we will implicitly assume that the dependency trees appearing in items of a parsing schema for a grammar G are taken from the set $\delta(G)$ of syntactically well-formed trees according to G.

This means that Collins's parser will be able to infer an item $[i,j,h]$ in the presence of an input string $w_1 \dots w_n$ if, using our set of D-rules, it is possible to build a projective dependency tree headed at h that spans the substring $w_i \dots w_j$ of the input.

Hypotheses: For an input string $w_1 \dots w_n$, the set of hypotheses is $\mathcal{H} = \{[i,i,i] \mid 0 \leq i \leq n+1\}$, where each item contains a forest with a single dependency tree having only one node i. This same set of hypotheses is used for all the parsers, so is not repeated for subsequent schemata.[4] Note that the nodes 0 and $n+1$ used in the definition do not correspond to actual input words—these are dummy nodes that we call beginning-of-sentence and end-of-sentence markers, respectively, and will be needed by several of the parsers described subsequently.

Final items: The set of final items is $\{[1,n,h] \mid 1 \leq h \leq n\}$. These items trivially represent parse trees for the input sentence whose head is some node h, expressing that the word w_h is the sentence's syntactic head.

4 Although in the parsers described in Section 7 we use a different notation for the hypotheses, they still are the same, as explained later.

Deduction steps: The set of deduction steps, D_{Col96}, is the union of the following:

$$\text{R-LINK: } \frac{[i,j,h_1] \quad [j+1,k,h_2]}{[i,k,h_2]}\ (w_{h_1},h_1) \rightarrow (w_{h_2},h_2) \qquad \text{L-LINK: } \frac{[i,j,h_1] \quad [j+1,k,h_2]}{[i,k,h_1]}\ (w_{h_2},h_2) \rightarrow (w_{h_1},h_1)$$

allowing us to join two contiguous trees by linking their heads with a rightward or leftward link, respectively. Figure 2 (right side) shows a graphical representation of how trees are joined by the L-LINK step. Note that, because this parsing strategy is data-driven, D-rules are used as **side conditions** for the parser's deduction steps. Side conditions restrict the inference relation by specifying which combinations of values are permissible for the variables appearing in the antecedents and consequent of deduction steps.

This parsing schema specifies a recognizer: Given a set of D-rules and an input string $w_1 \dots w_n$, the sentence can be parsed (projectively) under those D-rules if and only if the deduction system infers a coherent final item. When executing this schema with a deductive engine, the parse forest can be recovered by following back pointers, as in constituency parsers (Billot and Lang 1989).

This schema formalizes a parsing logic which is independent of the order and the way linking decisions are taken. Statistical models can be used to determine whether a step linking words a and b in positions i and j—i.e., having $(a,i) \rightarrow (b,j)$ as a side condition—is executed or not, and probabilities can be attached to items in order to assign different weights to different analyses of the sentence. The side conditions provide an explicit representation of the choice points where probabilistic decisions are made by the control mechanism that is executing the schema. The same principle applies to all D-rule-based parsers described in this article.

3.2 Eisner (1996)

Based on the number of free variables used in deduction steps of Collins's parser, it is apparent that its time complexity is $O(n^5)$: There are $O(n^5)$ combinations of index values with which each of its LINK steps can be executed.[5] This complexity arises because a parentless word (head) may appear in any position in the items generated by the parser; the complexity can be reduced to $O(n^3)$ by ensuring that parentless words only appear at the first or last position of an item. This is the idea behind the parser defined by Eisner (1996), which is still in wide use today (McDonald, Crammer, and Pereira 2005; Corston-Oliver et al. 2006). The parsing schema for this algorithm is defined as follows.

Item set: The item set is

$$\mathcal{I}_{Eis96} = \{[i,j,True,False] \mid 0 \leq i \leq j \leq n\} \cup \{[i,j,False,True] \mid 0 \leq i \leq j \leq n\} \cup \{[i,j,False,False] \mid 0 \leq i \leq j \leq n\},$$

where item $[i,j,True,False]$ corresponds to $[i,j,j] \in \mathcal{I}_{Col96}$, item $[i,j,False,True]$ corresponds to item $[i,j,i] \in \mathcal{I}_{Col96}$, and item $[i,j,False,False]$ is defined as the set of forests

5 For this and the rest of the complexity results in this article, we assume that the linking decision associated with a D-rule can be made in constant time.

of the form $\{T_1, T_2\}$ such that T_1, T_2 are projective, $head(T_1) = i$, $head(T_2) = j$, and there is some k ($i \leq k < j$) such that $yield(T_1) = i..k$ and $yield(T_2) = k+1..j$.

The flags b, c in $[i, j, b, c]$ indicate whether the nodes i and j, respectively, have a parent in the item. Items with one of the flags set to *True* represent dependency trees where the node i or j is the head, whereas items with both flags set to *False* represent pairs of trees headed at nodes i and j which jointly dominate the substring $w_i \ldots w_j$. Items of this kind correspond to disconnected dependency graphs.

Deduction steps: The set of deduction steps is as follows:[6]

$$\textsc{Initter:}\ \frac{[i,i,i]}{[i,i+1,i+1]} \qquad \textsc{CombineSpans:}\ \frac{[i,j,b,c]}{[j,k,not(c),d]}$$

$$\textsc{R-Link:}\ \frac{[i,j,False,False]}{[i,j,True,False]}\ (w_i,i) \rightarrow (w_j,j) \qquad \textsc{L-Link:}\ \frac{[i,j,False,False]}{[i,j,False,True]}\ (w_j,j) \rightarrow (w_i,i)$$

where the R-LINK and L-LINK steps establish a dependency link between the heads of an item containing two trees (i.e., having both flags set to *False*), producing a new item containing a single tree. The COMBINESPANS step is used to join two items that overlap at a single word, which must have a parent in only one of the items, so that the result of joining trees coming from both items (without creating any dependency link) is a well-formed dependency tree.

Final items: The set of final items is $\{[0, n, False, True]\}$. Note that these items represent dependency trees rooted at the beginning-of-sentence marker 0, which acts as a "dummy head" for the sentence. In order for the algorithm to parse sentences correctly, we need to define D-rules to allow the real sentence head to be linked to the node 0.

3.3 Eisner and Satta (1999)

Eisner and Satta (1999) define an $O(n^3)$ parser for split head automaton grammars which can be used for dependency parsing. This algorithm is conceptually simpler than Eisner's (1996) algorithm, because it only uses items representing single dependency trees, avoiding items of the form $[i, j, False, False]$.

Item set: The item set is $\mathcal{I}_{ES99} = \{[i,j,i] \mid 0 \leq i \leq j \leq n\} \cup \{[i,j,j] \mid 0 \leq i \leq j \leq n\}$, where items are defined as in Collins's parsing schema.

Deduction steps: The deduction steps for this parser are the following:

$$\textsc{R-Link:}\ \frac{[i,j,i]}{[j+1,k,k]}\ (w_i,i) \rightarrow (w_k,k) \qquad \textsc{L-Link:}\ \frac{[i,j,i]}{[j+1,k,k]}\ (w_k,k) \rightarrow (w_i,i)$$

$$\textsc{R-Combiner:}\ \frac{[i,j,i]}{[i,k,i]} \qquad \textsc{L-Combiner:}\ \frac{[i,j,j]}{[i,k,k]}$$

6 We could have used items $[i, i+1, False, False]$ as hypotheses for this parsing schema, and not require an *Initter* step, but we prefer a standard set of hypotheses valid for all parsers as it facilitates more straightforward proofs of the relations between schemata.

where LINK steps create a dependency link between two dependency trees spanning adjacent segments of the input, and COMBINER steps join two overlapping trees by a graph union operation that does not create new links. COMBINER steps follow the same mechanism as those in the algorithm of Eisner (1996), and LINK steps work analogously to those of Collins (1996), so this schema can be seen as being intermediate between those two algorithms. These relationships will be formally described in Section 4.

Final items: The set of final items is $\{[0, n, 0]\}$. By convention, parse trees have the beginning-of-sentence marker 0 as their head, as in the previous algorithm.

When described for head automaton grammars (Eisner and Satta 1999), this algorithm appears to be more complex to understand and implement than the previous one, requiring four different kinds of items to keep track of the state of the automata used by the grammars. However, this abstract representation of its underlying semantics reveals that this parsing strategy is, in fact, conceptually simpler for dependency parsing.

3.4 Yamada and Matsumoto (2003)

Yamada and Matsumoto (2003) define a deterministic, shift-reduce dependency parser guided by support vector machines, which achieves over 90% dependency accuracy on Section 23 of the Wall Street Journal Penn Treebank. Parsing schemata cannot specify control strategies that guide deterministic parsers; schemata work at an abstraction level, defining a set of operations without procedural constraints on the order in which they are applied. However, deterministic parsers can be viewed as optimizations of underlying nondeterministic algorithms, and we can represent the actions of the underlying parser as deduction steps, abstracting away from the deterministic implementation details, obtaining a potentially interesting nondeterministic dependency parser.

Actions in Yamada and Matsumoto's parser create links between two target nodes, which act as heads of neighboring dependency trees. One of the actions creates a link where the left target node becomes a child of the right one, and the head of a tree located directly to the left of the target nodes becomes the new left target node. The other action is symmetric, performing the same operation with a right-to-left link. An $O(n^3)$ nondeterministic parser generalizing this behavior can be defined as follows.

Item set: The item set is $\mathcal{I}_{YM03} = \{[i, j] \mid 0 \leq i \leq j \leq n + 1\}$, where each item $[i, j]$ corresponds to the item $[i, j, \mathit{False}, \mathit{False}]$ in $\mathcal{I}_{Eis96}$.

Deduction steps: The deduction steps are as follows:

$$\text{INITTER: } \frac{[i, i, i] \quad [i+1, i+1, i+1]}{[i, i+1]} \qquad \text{R-LINK: } \frac{[i, j] \quad [j, k]}{[i, k]}\ (w_j, j) \rightarrow (w_k, k) \qquad \text{L-LINK: } \frac{[i, j] \quad [j, k]}{[i, k]}\ (w_j, j) \rightarrow (w_i, i)$$

where a LINK step joins a pair of items containing forests with two trees each and overlapping at a head node, and creates a dependency link from their common head to one of the peripheral heads. Note that this is analogous to performing an Eisner LINK step immediately followed by an Eisner COMBINE step, as will be further analyzed in Section 4.

Final items: The set of final items is $\{[0, n + 1]\}$. For this set to be well-defined, the grammar must not have D-rules of the form $(w_i, i) \rightarrow (w_{n+1}, n + 1)$, that is, it must not

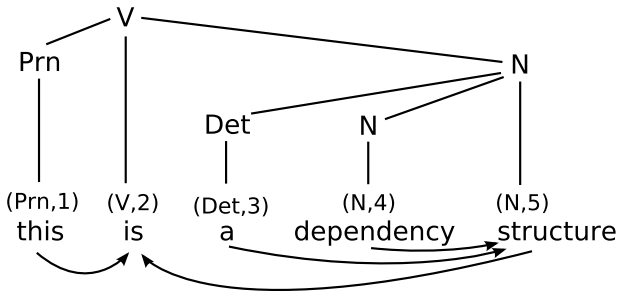

Figure 3
Grounded extended dependency tree and associated dependency structure.

allow the end-of-sentence marker to govern any words. If the grammar satisfies this condition, it is trivial to see that every forest in an item of the form $[0, n+1]$ must contain a parse tree rooted at the beginning-of-sentence marker and with yield $0..n$.

As can be seen from the schema, this algorithm requires less bookkeeping than the other parsers described here.

3.5 Lombardo and Lesmo (1996) and Other Earley-Based Parsers

The algorithms presented so far are based on making individual decisions about dependency links, represented by D-rules. Other parsers, such as that of Lombardo and Lesmo (1996), use grammars with CFG-like rules which encode the preferred order of dependents for each given governor. For example, a rule of the form $N(Det * PP)$ is used to allow N to have Det as left dependent and PP as right dependent. The algorithm by Lombardo and Lesmo is a version of Earley's CFG parser (Earley 1970) that uses Gaifman's dependency grammar (Gaifman 1965).

As this algorithm predicts dependency relations before building them, item sets contain **extended dependency trees**, trees that have two kinds of nodes: preterminal nodes and terminal nodes. Depending on whether all the preterminal nodes have been linked to terminals, extended dependency trees can either be **grounded**, in which case they are isomorphic to traditional dependency graphs (see Figure 3), or **ungrounded**, as in Figure 4, in which case they capture parser states in which some structure has been predicted, but not yet found. Note that a dependency graph can always be extracted from such a tree, but in the ungrounded case different extended trees can be associated with the same graph. Gómez-Rodríguez, Carroll, and Weir (2008) present extended dependency trees in more detail.

Item set: The item set is $\mathcal{I}_{LomLes} = \{[A(\alpha \bullet \beta), i, j] \mid A(\alpha\beta) \in P \wedge 1 \leq i \leq j+1 \leq n\}$ where α and β are strings; P is a set of CFG-like rules;[7] and each item $[A(\alpha \bullet \beta), i, j]$ represents the set of projective extended dependency trees rooted at A, where the direct children of A are $\alpha\beta$, and the subtrees rooted at α have yield $i..j$. Note that Lombardo and Lesmo's parser uses both grounded trees (in items $[A(\alpha\bullet), i, j]$) and non-grounded trees (in items

7 A CFG-like rule $A(\alpha * \beta)$ rewrites a preterminal A to strings $\alpha x \beta$ over terminals and preterminals, where α, β are strings of preterminals and x is a terminal of category A (the head of the rule). A special rule $*(S)$ is used to state that the preterminal S can act as the root of an extended dependency tree.

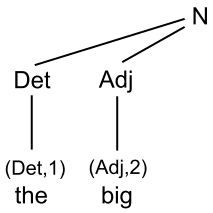

Figure 4
Non-grounded extended dependency tree: A determiner and adjective have been found and, according to the grammar, we expect a noun that will act as their common head. As this head has not been read, no dependency links have been established.

$[A(\alpha \bullet \beta), i, j]$, where β is nonempty). Items in this parser can represent infinite sets of extended dependency trees, as in Earley's CFG parser but unlike items in D-rule-based parsers, which are finite sets.

Deduction steps: The deduction steps for this parsing schema are as follows:

$$\textsc{Initter:}\ \frac{}{[(\bullet S), 1, 0]}\ *(S) \in P \qquad \textsc{Predictor:}\ \frac{[A(\alpha \bullet B\beta), i, j]}{[B(\bullet\gamma), j+1, j]}\ B(\gamma) \in P$$

$$\textsc{Scanner:}\ \frac{\begin{matrix}[A(\alpha \bullet *\beta), i, h-1] \\ [h, h, h]\end{matrix}}{[A(\alpha * \bullet\beta), i, h]}\ w_h \text{ IS } A \qquad \textsc{Completer:}\ \frac{\begin{matrix}[A(\alpha \bullet B\beta), i, j] \\ [B(\gamma\bullet), j+1, k]\end{matrix}}{[A(\alpha B \bullet \beta), i, k]}$$

Final items: The final item set is $\{[(S\bullet), 1, n]\}$.

The schema for Lombardo and Lesmo's parser is a variant of the Earley constituency parser (cf. Sikkel 1997), with minor changes to adapt it to dependency grammar (for example, the SCANNER always moves the dot over the head symbol $*$, rather than over a terminal symbol). Analogously, other dependency parsing schemata based on CFG-like rules can be obtained by modifying CFG parsing schemata of Sikkel (1997): The algorithm of Barbero et al. (1998) can be obtained from the left-corner parser, and the parser described by Courtin and Genthial (1998) is a variant of the head-corner parser.

3.6 Nivre (2003)

Nivre (2003) describes a shift-reduce algorithm for projective dependency parsing, later extended by Nivre, Hall, and Nilsson (2004). With linear-time performance and competitive parsing accuracy (Nivre et al. 2006; Nivre and McDonald 2008), it is one of the parsers included in the MaltParser system (Nivre et al. 2007), which is currently widely used (e.g., Nivre et al. 2007; Surdeanu et al. 2008).

The parser proceeds by reading the sentence from left to right, using a stack and four different kinds of transitions between configurations. The transition system defined by all the possible configurations and transitions is nondeterministic, and machine learning techniques are used to train a mechanism that produces a deterministic parser.

A deduction system describing the transitions of the parser is defined by Nivre, Hall, and Nilsson (2004), with the following set of rules that describes transitions

between configurations (we use the symbol ρ for a stack and the notation $\rho :: h$ for the stack resulting from pushing h into ρ, and β_i to represent a buffer of the form $w_i \dots w_n$):

$$\text{Initter}\ \frac{}{(\langle\rangle, \beta_0, \emptyset)} \qquad \text{Shift}\ \frac{(\rho, \beta_f, V)}{(\rho :: f, \beta_{f+1}, V)} \qquad \text{Reduce}\ \frac{(\rho :: l, \beta_f, V)}{(\rho, \beta_f, V)}\ \exists a_l \rightarrow a_k \in V$$

$$\text{L-Link}\ \frac{(\rho :: l, \beta_f, V)}{(\rho :: l :: f, \beta_{f+1}, V \cup \{a_f \rightarrow a_l\})}\ a_f \rightarrow a_l \mid \nexists a_f \rightarrow a_k \in V$$

$$\text{R-Link}\ \frac{(\rho :: l, \beta_f, V)}{(\rho, \beta_f, V \cup \{a_l \rightarrow a_f\})}\ a_l \rightarrow a_f \mid \nexists a_l \rightarrow a_k \in V$$

This set of inference rules is not a parsing schema, however, because the entities it works with are not items. Although the antecedents and consequents in this deduction system are parser configurations, they do not correspond to disjoint sets of dependency structures (several configurations may correspond to the same dependency structures), and therefore do not conform to the definition of an item set. It would be possible to define parsing schemata in a different way with a weaker definition of item sets allowing these configurations as items, but this would make it harder to formalize relations between parsers, because they rely on the properties of item sets.

A parsing schema for Nivre's parser can be obtained by abstracting away the rules in the system that are implementing control structures, however, and expressing only declarative aspects of the parser's tree building logic. To do this, we first obtain a simplified version of the deduction system. This version of the parser is obtained by storing an index f rather than the full buffer β_f in each configuration, and then grouping configurations that share common features, making them equivalent for the side conditions of the system: Instead of storing the full set of dependency links that the algorithm has constructed up to a given point (denoted by V), we only keep track of whether elements in the stack have been assigned a head or not; and we represent this by using a stack of pairs (l, b), where l is the position of a word in the string and b is a flag which is *True* if the corresponding node has been assigned a head or *False* if it has not:

$$\text{Initter}\ \frac{}{(\langle\rangle, 0)} \qquad \text{Shift}\ \frac{(\rho, f)}{(\rho :: (f, \mathit{False}), f+1)} \qquad \text{Reduce}\ \frac{(\rho :: (l, \mathit{True}), f)}{(\rho, f)}$$

$$\text{L-Link}\ \frac{(\rho :: (l, h), f)}{(\rho :: (l, h) :: (f, \mathit{True}), f+1)}\ (w_f, f) \rightarrow (w_l, l) \qquad \text{R-Link}\ \frac{(\rho :: (l, \mathit{False}), f)}{(\rho, f)}\ (w_l, l) \rightarrow (w_f, f)$$

To obtain a parsing schema from this deduction system, we retain only rules pertaining to the way in which the parser joins dependency structures and builds links between them. In particular, the *Reduce* step is just a mechanism to select which of a set of possible "linkable words" to link to the word currently being read. Two different configurations corresponding to the same dependency structure may have different lists of words in the stack depending on which *Reduce* steps have been executed. In the parsing schema, these configurations must correspond to the same item, as they involve the same dependency structures. To define an item set for this parser, we must establish which words *could* be on the stack at each configuration.

A node in a dependency graph T is **right-linkable** if it is not a dependent of any node situated to its right, and is not covered by any dependency link (j is covered by

the link $i \rightarrow k$ if $i < j < k$ or $i > j > k$). A link cannot be created between a non-right-linkable node and any node to the right of T without violating the projectivity property. When the parser is reading a particular word at position f, the following properties hold for all nodes to the left of f (nodes $0 \ldots f-1$):

- If the node i is not right-linkable, then it cannot be on the stack.
- If the node i does not have a head, then it must be on the stack. (Note that nodes that do not have a head assigned are always right-linkable.)
- If the node i has a head and it is right-linkable, then it may or may not be on the stack, depending on the transitions that we have executed.

A dependency parsing schema represents items with lists (instead of stacks) containing all the nodes found so far which are right-linkable, and a flag associated with each node indicating whether it has been assigned a head or not. Instead of using *Reduce* steps to decide which node to choose as a head of the one corresponding to the currently-read word, we allow any node in the list that does not have a headless node to its right to be the head; this is equivalent to performing several *Reduce* transitions followed by an *L-link* transition.

Item set: The item set is

$$\mathcal{I}_{Niv} = \{[i, \langle (i_1, b_1), \ldots, (i_k, b_k) \rangle] \mid 0 \leq i \leq n+1 \wedge 0 \leq i_1 \leq \ldots \leq i_k \leq n \wedge b_j \in \{False, True\}\}$$

where an item $[i, L]$ represents the set of forests of projective trees of the form $F = \{T_1, \ldots, T_w\}$ $(w > 0)$ satisfying the following:

- The concatenation of the yields of $T_1, \ldots, T_w$ is $0..i$,
- The heads of the trees $T_1, \ldots, T_{w-1}$ are the nodes j where $(j, False) \in L$; and the head of the tree T_w is the node i,
- The right-linkable nodes in the dependency graph corresponding to the union of the trees in F are the nodes j where $(j, b) \in L$, with $b \in \{False, True\}$.

Final items: The set of final items is $\{[n+1, \langle (0, False), (v_1, True), \ldots, (v_k, True) \rangle] \mid 1 \leq v_j \leq n\}$, the set of items containing a forest with a single projective dependency tree T headed at the dummy node 0, whose yield spans the whole input string, and which contains any set of right-linkable words.

Deduction steps: The deduction steps are as follows:

$$\textsc{Initter:}\ \frac{}{[0, \langle\rangle]} \qquad \textsc{Advance:}\ \frac{[i, \langle (i_1, b_1), \ldots, (i_k, b_k) \rangle]}{[i+1, \langle (i_1, b_1), \ldots, (i_k, b_k), (i, False) \rangle]}$$

$$\textsc{L-Link:}\ \frac{[i, \langle (i_1, b_1), \ldots, (i_k, b_k), (l, b), (v_1, True), \ldots, (v_r, True) \rangle]}{[i+1, \langle (i_1, b_1), \ldots, (i_k, b_k), (l, b), (i, True) \rangle]}\ (w_i, i) \rightarrow (w_l, l)$$

$$\textsc{R-Link:}\ \frac{[i, \langle (i_1, b_1), \ldots, (i_k, b_k), (h, False), (v_1, True), \ldots, (v_r, True) \rangle]}{[i, \langle (i_1, b_1), \ldots, (i_k, b_k) \rangle]}\ (w_h, h) \rightarrow (w_i, i)$$

Note that a naive nondeterministic implementation of this schema in a generic deductive engine would have exponential complexity. The linear complexity in Nivre's algorithm is achieved by using a control strategy that deterministically selects a single transition at each state.

3.7 Covington's (2001) Projective Parser

Covington (2001) defines a non-projective dependency parser, and a projective variant called Algorithm LSUP (for List-based Search with Uniqueness and Projectivity). Unfortunately, the algorithm presented in Covington (2001) is not complete: It does not parse all projective dependency structures, because when creating leftward links it assumes that the head of a node i must be a reflexive-transitive head of the node $i-1$, which is not always the case. For instance, the structure shown in Figure 5 cannot be parsed because the constraints imposed by the algorithm prevent it from finding the head of 4.

The MaltParser system (Nivre et al. 2007) includes an implementation of a complete variant of Covington's LSUP parser where these constraints have been relaxed. This implementation has the same tree building logic as the parser described by Nivre (2003), differing from it only with respect to the control structure. Thus, it can be seen as a different realization of the schema shown in Section 3.6.

4. Relations Between Dependency Parsers

The parsing schemata framework can be exploited to establish how different algorithms are related, improving our understanding of the features of these parsers, and potentially exposing new algorithms that combine characteristics of existing parsers in novel ways. Sikkel (1994) defines various relations between schemata that fall into two categories: **generalization** relations, which are used to obtain more fine-grained versions of parsers, and **filtering** relations, which can be seen as the converse of generalization and are used to reduce the number of items and/or steps needed for parsing. Informally, a parsing schema can be generalized from another via the following transformations:

- **Item refinement**: P_2 is an item refinement of P_1, written $P_1 \xrightarrow{ir} P_2$, if there is a mapping between items in both parsers such that single items in P_1 are mapped into multiple items in P_2 and individual deductions are preserved.
- **Step refinement**: $P_1 \xrightarrow{sr} P_2$ if the item set of P_1 is a subset of that of P_2 and every deduction step in P_1 can be emulated by a sequence of steps in P_2.

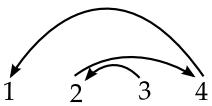

Figure 5
A projective dependency structure that cannot be parsed with Covington's LSUP algorithm.

A schema can be obtained from another by filtering in the following ways:

- **Static/dynamic filtering**: $P_1 \xrightarrow{sf/df} P_2$ if the item set of P_2 is a subset of that of P_1 and P_2 allows a subset of the direct inferences in P_1. Sikkel (1994) explains the distinction between static and dynamic filtering, which is not used here.
- **Item contraction**: The inverse of item refinement: $P_1 \xrightarrow{ic} P_2$ if $P_2 \xrightarrow{ir} P_1$.
- **Step contraction**: The inverse of step refinement: $P_1 \xrightarrow{sc} P_2$ if $P_2 \xrightarrow{sr} P_1$.

Many of the parsing schemata described in Section 3 can be related (see Figure 6), but for space reasons we sketch proofs for only the more interesting cases.

Theorem 1
Yamada and Matsumoto (2003) $\xrightarrow{sr}$ Eisner (1996).

Proof 1
It is easy to see from the schema definitions that $\mathcal{I}_{YM03} \subseteq \mathcal{I}_{Eis96}$. We must verify that every deduction step in the Yamada and Matsumoto (2003) schema can be emulated by a sequence of inferences in the Eisner (1996) schema. For the INITTER step this is trivial as the INITTERs of both parsers are equivalent. Expressing the R-LINK step of Yamada and Matsumoto's parser in the notation used for Eisner items gives:

$$\text{R-Link}\ \frac{[i, j, \mathit{False}, \mathit{False}] \qquad [j, k, \mathit{False}, \mathit{False}]}{[i, k, \mathit{False}, \mathit{False}]}\ (w_j, j) \rightarrow (w_k, k)$$

This can be emulated in Eisner's parser by an R-LINK step followed by a COMBINESPANS step:

$$[j, k, \mathit{False}, \mathit{False}] \vdash [j, k, \mathit{True}, \mathit{False}]\ \text{(by R-LINK)}$$
$$[j, k, \mathit{True}, \mathit{False}], [i, j, \mathit{False}, \mathit{False}] \vdash [i, k, \mathit{False}, \mathit{False}]\ \text{(by COMBINESPANS)}$$

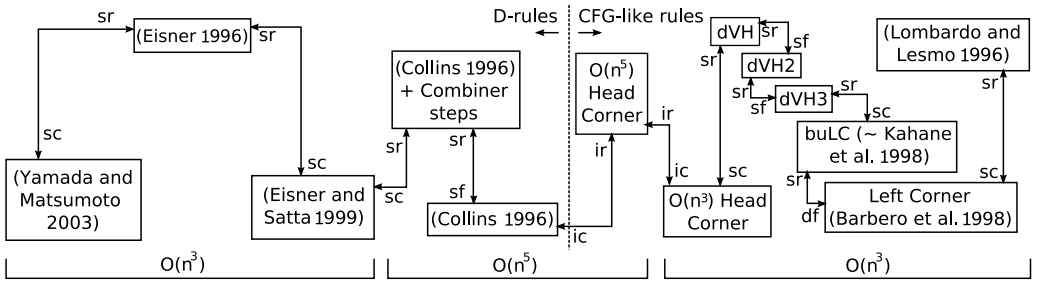

Figure 6
Relating several well-known dependency parsers. Arrows pointing up correspond to generalization relations, while those pointing down correspond to filtering. The specific subtype of relation is shown in each arrow's label, following the notation in Section 4.

Symmetrically, the L-LINK step in Yamada and Matsumoto's parser can be emulated by an L-LINK followed by a COMBINESPANS in Eisner's. ■

Theorem 2
Eisner and Satta (1999) $\xrightarrow{sr}$ Eisner (1996).

Proof 2
Writing R-LINK in Eisner and Satta's parser in the notation used for Eisner items gives

$$\text{R-LINK: } \frac{[i, j, \mathit{False}, \mathit{True}] \qquad [j+1, k, \mathit{True}, \mathit{False}]}{[i, k, \mathit{True}, \mathit{False}]} \; (w_i, i) \rightarrow (w_k, k)$$

This inference can be emulated in Eisner's parser as follows:

$$\vdash [j, j+1, \mathit{False}, \mathit{False}] \text{ (by INITTER)}$$
$$[i, j, \mathit{False}, \mathit{True}], [j, j+1, \mathit{False}, \mathit{False}] \vdash [i, j+1, \mathit{False}, \mathit{False}] \text{ (by COMBINESPANS)}$$
$$[i, j+1, \mathit{False}, \mathit{False}], [j+1, k, \mathit{True}, \mathit{False}] \vdash [i, k, \mathit{False}, \mathit{False}] \text{ (by COMBINESPANS)}$$
$$[i, k, \mathit{False}, \mathit{False}] \vdash [i, k, \mathit{True}, \mathit{False}] \text{ (by R-LINK)}$$

The proof corresponding to the L-LINK step is symmetric. As for the R-COMBINER and L-COMBINER steps in Eisner and Satta's parser, it is easy to see that they are particular cases of the COMBINESPANS step in Eisner's, and therefore can be emulated by a single application of COMBINESPANS. ■

Note that, in practice, these two relations mean that the parsers by Eisner and Satta (1999) and Yamada and Matsumoto (2003) are more efficient, at the schema level, than that of Eisner (1996), in that they generate fewer items and need fewer steps to perform the same deductions. These two parsers also have the interesting property that they use disjoint item sets (one uses items representing trees while the other uses items representing pairs of trees); and the union of these disjoint sets is the item set used by Eisner's parser. The optimization in Yamada and Matsumoto's parser comes from contracting deductions in Eisner's parser so that linking operations are immediately followed by combining operations; whereas Eisner and Satta's parser does the opposite, forcing combining operations to be followed by linking operations.

By generalizing the linking steps in Eisner and Satta's parser so that the head of each item can be in any position, we obtain an $O(n^5)$ parser which can be filtered into the parser of Collins (1996) by eliminating the COMBINER steps. From Collins's parser, we obtain an $O(n^5)$ head-corner parser based on CFG-like rules by an item refinement in which each Collins item $[i, j, h]$ is split into a set of items $[A(\alpha \bullet \beta \bullet \gamma), i, j, h]$. The refinement relation between these parsers only holds if for every D-rule $B \rightarrow A$ there is a corresponding CFG-like rule $A \rightarrow \ldots B \ldots$ in the grammar used by the head-corner parser. Although this parser uses three indices i, j, h, using CFG-like rules to guide linking decisions makes the h indices redundant. This simplification is an item contraction which results in an $O(n^3)$ head-corner parser. From here, we can follow the procedure described by Sikkel (1994) to relate this head-corner algorithm to parsers analogous to other algorithms for CFGs. In this way, we can refine the head-corner parser to a variant of the algorithm by de Vreught and Honig (1989) (Sikkel 1997), and by successive filters we reach a left-corner parser which is equivalent to the one described by Barbero et al. (1998), and a step contraction of the Earley-based dependency parser by Lombardo

and Lesmo (1996). The proofs for these relations are the same as those given by Sikkel (1994), except that the dependency variants of each algorithm are simpler (due to the absence of epsilon rules and the fact that the rules are lexicalized). The names used for schemata *dVH1*, *dVH2*, *dVH3*, and *buLC* shown in Figure 6 come from Sikkel (1994, 1997). These dependency parsing schemata are versions of the homonymous schemata whose complete description can be found in Sikkel (1997), adapted for dependency parsing. Gómez-Rodríguez (2009) gives a more thorough explanation of these relations and schemata.

5. Proving Correctness

Another use of the parsing schemata framework is that it is helpful in establishing the correctness of a parser. Furthermore, relations between schemata can be used to establish the correctness of one schema from that of related ones. In this section, we show that the schemata for Yamada and Matsumoto (2003) and Eisner and Satta (1999) are correct, and use this to prove the correctness of the schema for Eisner (1996).

Theorem 3
The Eisner and Satta (1999) parsing schema is correct.

Proof 3
To prove correctness, we must show both soundness and completeness. To verify soundness we need to check that every individual deduction step in the parser infers a coherent consequent item when applied to coherent antecedents (i.e., in this case, that steps always generate non-empty items that conform to the definition in Section 3.3). This is shown by checking that, given two antecedents of a deduction step that contain a tree licensed by a set of D-rules G, the consequent of the step also contains such a tree. The tree for the consequent is built from the trees corresponding to the antecedents: by a graph union operation, in the case of COMBINER steps; or by linking the heads of both trees with a dependency relation licensed by G, in the case of LINK steps.

To prove completeness we prove that all coherent final items are derivable by proving the stronger result that all coherent items are derivable. We show this by strong induction on the *length* of items, where the length of an item $\iota = [i, k, h]$ is defined as $\mathit{length}(\iota) = k - i + 1$. Coherent items of length 1 are the hypotheses of the schema (of the form $[i, i, i]$) which are trivially derivable. We show that, if all coherent items of length m are derivable for all $1 \leq m < l$, then items of length l are also derivable.

Let $[i, k, i]$ be an item of length l in $\mathcal{I}_{ES99}$ (thus, $l = k - i + 1$). If this item is coherent, it contains a dependency tree T such that $\mathit{yield}(T) = i..k$ and $\mathit{head}(t) = i$. By construction, the root of T is labelled i. Let j be the rightmost daughter of i in T. Because T is projective, we know that the yield of j must be of the form $l..k$, where $i < l \leq j \leq k$. If $l < j$, then l is the leftmost transitive dependent of j in T, and if $k > j$, then we know that k is the rightmost transitive dependent of j in T.

Let T_j be the subtree of T rooted at j, T_1 be the tree obtained from removing T_j by T,[8] T_2 be the tree obtained by removing all the nodes to the right of j from T_j, and T_3 be the tree obtained by removing all the nodes to the left of j from T_j. By construction,

8 Removing a subtree from a dependency tree involves removing all the nodes in the subtree from its vertex set, and all the outgoing links from nodes in the subtree from its edge set.

T_1 belongs to a coherent item $[i, l-1, i]$, T_2 belongs to a coherent item $[l, j, j]$, and T_3 belongs to a coherent item $[j, k, j]$. Because these three items have a length strictly less than l, by the inductive hypothesis, they are derivable. Thus the item $[i, k, i]$ is also derivable, as it can be obtained from these derivable items by the following inferences:

$$[i, l-1, i], [l, j, j] \vdash [i, j, i] \text{ (by the L-LINK step)}$$
$$[i, j, i], [j, k, j] \vdash [i, k, i] \quad \text{(by the L-COMBINER step)}$$

This proves that all coherent items of length l which are of the form $[i, k, i]$ are derivable under the induction hypothesis. The same can be shown for items of the form $[i, k, k]$ by symmetric reasoning. ■

Theorem 4
The Yamada and Matsumoto (2003) parsing schema is correct.

Proof 4
Soundness is verified by building forests for the consequents of steps from those corresponding to the antecedents. To prove completeness we use strong induction on the length of items, where the length of an item $[i, j]$ is defined as $j - i + 1$. The induction step involves considering any coherent item $[i, k]$ of length $l > 2$ ($l = 2$ is the base case here because items of length 2 are generated by the *Initter* step) and showing that it can be inferred from derivable antecedents of length less than l, so it is derivable. If $l > 2$, either i has at least one right dependent or k has at least one left dependent in the item. Suppose i has a right dependent; if T_1 and T_2 are the trees rooted at i and k in a forest in $[i, k]$, we call j the rightmost daughter of i and consider the following trees:

- V = the subtree of T_1 rooted at j,
- U_1 = the tree obtained by removing V from T_1,
- U_2 = the tree obtained by removing all nodes to the right of j from V,
- U_3 = the tree obtained by removing all nodes to the left of j from V.

The forest $\{U_1, U_2\}$ belongs to the coherent item $[i, j]$, and $\{U_3, T_2\}$ belongs to the coherent item $[j, k]$. From these two items, we can obtain $[i, k]$ by using the L-LINK step. Symmetric reasoning can be applied if i has no right dependents but k has at least one left dependent, analogously to the case of the previous parser. ■

Theorem 5
The Eisner (1996) parsing schema is correct.

Proof 5
By using the previous proofs and the relationships between schemata established earlier, we show that the parser of Eisner (1996) is correct: Soundness is straightforward, and completeness can be shown by using the properties of other algorithms. Because the set of final items in the Eisner (1996) and Eisner and Satta (1999) schemata are the same, and the former is a step refinement of the latter, the completeness of Eisner and Satta's parser directly implies the completeness of Eisner's parser. Alternatively, we can use Yamada and Matsumoto's parser to prove the correctness of Eisner's parser if we

redefine the set of final items in the latter to be items of the form $[0, n+1, \mathit{False}, \mathit{False}]$, which are equally valid as final items since they always contain parse trees. This idea can be applied to transfer proofs of completeness across any refinement relation. ■

6. Non-Projective Schemata

The parsing schemata presented so far define parsers that are restricted to projective dependency structures, that is, structures in which the set of reflexive-transitive dependents of each node forms a contiguous substring of the input. We now show that the dependency parsing schema formalism can also describe various non-projective parsers.

6.1 Pseudo-Projectivity

Pseudo-projective parsers generate non-projective analyses in polynomial time by using a projective parsing strategy and postprocessing the results to establish non-projective links. This projective parsing strategy can be represented by dependency parsing schemata such as those seen in Section 3. For example, the algorithm of Kahane, Nasr, and Rambow (1998) uses a strategy similar to Lombardo and Lesmo (1996), but with the following initializer step instead of the INITTER and PREDICTOR:

$$\textsc{Initter:}\ \frac{}{[A(\bullet\alpha), i, i-1]}\ A(\alpha) \in P \wedge 1 \leq i \leq n$$

The initialization step specified by Kahane, Nasr, and Rambow (1998) differs from this (directly consuming a nonterminal from the input) but this gives an incomplete algorithm. The problem can be fixed either by using the step shown here instead (bottom–up Earley strategy) or by adding an additional step turning it into a bottom–up left-corner parser.

6.2 Attardi (2006)

The non-projective parser of Attardi (2006) extends the algorithm of Yamada and Matsumoto (2003), adding additional shift and reduce actions to handle non-projective dependency structures. These extra actions allow the parser to link to nodes that are several positions deep in the stack, creating non-projective links. In particular, Attardi uses six non-projective actions: two actions to link to nodes that are two positions deep, another two actions for nodes that are three positions deep, and a third pair of actions that generalizes the previous ones to n positions deep for any n. Thus, the maximum depth in the stack to which links can be created can be configured according to the actions allowed. We use Att_d for the variant of the algorithm that allows links only up to depth d, and Att_∞ for the original, unrestricted algorithm with unlimited depth actions. A nondeterministic version of the algorithm Att_d can be described as follows.

Item set: The item set is $\mathcal{I}_{Att} = \{[h_1, h_2, \ldots, h_m] \mid 0 \leq h_1 < \ldots < h_m \leq n+1\}$ where $[h_1, h_2, \ldots, h_m]$ is the set of dependency forests of the form $\{T_1, T_2, \ldots, T_m\}$ such that: $\mathit{head}(T_i) = h_i$ for each $i \in [1..m]$; and the projections of the nodes $h_1, h_2, \ldots, h_m$ are pairwise disjoint, and their union is $[h_1..h_m]$.

Deduction steps: The set of deduction steps for Att_d is the following:

$$\textsc{Initter}: \frac{[i,i,i]}{[i,i+1]} \qquad \textsc{Combine}: \frac{[h_1,h_2,\ldots,h_m]\quad[h_m,h_{m+1},\ldots,h_p]}{[h_1,h_2,\ldots,h_p]}$$

$$\textsc{Link}: \frac{[h_1,h_2,\ldots,h_m]}{[h_1,h_2,\ldots,h_{i-1},h_{i+1},\ldots,h_m]}\ (w_{h_i},h_i) \rightarrow (w_{h_j},h_j), 1 < i < m, 1 \leq j \leq m, j \neq i, \mid j-i \mid \leq d$$

Deduction steps for Att_∞ are obtained by removing the constraint $\mid j-i \mid \leq d$ from this set (this restriction corresponds to the maximum stack depth to which dependency links can be created).

Final items: The set of final items is $\{[0,n+1]\}$. Although similar to the final item set for Yamada and Matsumoto's parser, they differ in that an Attardi item of the form $[0,n+1]$ may contain forests with non-projective dependency trees.

Given the number of indices manipulated in the schema, a nondeterministic implementation of Att_d has exponential complexity with respect to input length (though in the implementation of Attardi [2006], control structures determinize the algorithm). Soundness of the algorithm Att_∞ is shown as in the previous algorithms, and completeness can be shown by reasoning that every coherent final item $[0,n+1]$ can be obtained by first performing $n+1$ INITTER steps to obtain items $[i,i+1]$ for each $0 \leq i \leq n$, then using n COMBINERs to join all of these items into $[0,1,\ldots,n,n+1]$, and then performing the LINK steps corresponding to the links in a tree contained in $[0,n+1]$ to obtain this final item. The algorithm Att_d where d is finite is not correct with respect to the set of non-projective dependency structures, because it only parses a restricted subset of them (Attardi 2006). Note that the algorithm Att_d is a static filter of Att_{d+1} for every natural number d, since the set of deduction steps of Att_d is a subset of that of Att_{d+1}.

6.3 The MH_k Parser

We now define a novel variant of Attardi's parser with polynomial complexity by limiting the number of trees in each forest contained in an item (rather than limiting stack depth), producing a parsing schema MH_k (standing for multi-headed with at most k heads per item). Its item set is $\mathcal{I}_{MH_k} = \{[h_1,h_2,\ldots,h_m] \mid 0 \leq h_1 < \ldots < h_m \leq n+1 \wedge 2 \leq m \leq k\}$ where $[h_1,h_2,\ldots,h_m]$ is defined as in $\mathcal{I}_{Att}$, and the deduction steps are the following:

$$\textsc{Initter}: \frac{[i,i,i]}{[i,i+1]} \qquad \textsc{Combine}: \frac{[h_1,h_2,\ldots,h_m]\quad[h_m,h_{m+1},\ldots,h_p]}{[h_1,h_2,\ldots,h_p]}\ p \leq k$$

$$\textsc{Link}: \frac{[h_1,h_2,\ldots,h_m]}{[h_1,h_2,\ldots,h_{i-1},h_{i+1},\ldots,h_m]}\ (w_{h_i},h_i) \rightarrow (w_{h_j},h_j), 1 < i < m, 1 \leq j \leq m, j \neq i$$

As with the Att_d parser, MH_k parses a restricted subset of non-projective dependency structures, such that the set of structures parsed by MH_k is always a subset of those parsed by MH_{k+1}. The MH_∞ parser, obtained by assuming that the number of trees per forest is unbounded, is equivalent to Att_∞, and therefore correct with respect to

the set of non-projective dependency structures. For finite values of k, MH_{d+2} is a static filter of Att_d, because its sets of items and deduction steps are subsets of those of Att_d. Therefore, the set of structures parsed by MH_{d+2} is also a subset of those parsed by Att_d.

The complexity of the MH_k parser is $O(n^k)$. For $k = 3$, MH_3 is a step refinement of the parser by Yamada and Matsumoto (2003) that parses projective structures only, but by modifying the bound k we can define polynomial-time algorithms that parse larger sets of non-projective dependency structures. The MH_k parser has the property of being able to parse any possible dependency structure as long as we make k large enough.

6.4 MST Parser (McDonald et al. 2005)

McDonald et al. (2005) describe a parser which finds a nonprojective analysis for a sentence in $O(n^2)$ time under a strong independence assumption called an **edge-factored model**: Each dependency decision is assumed to be independent of all the others (McDonald and Satta 2007). Despite the restrictiveness of this model, this maximum spanning tree (MST) parser achieves state-of-the-art performance for projective and non-projective structures (Che et al. 2008; Nivre and McDonald 2008; Surdeanu et al. 2008). The parser considers the weighted graph formed by all the possible dependencies between pairs of input words, and applies an MST algorithm to find a dependency tree covering all the words in the sentence and maximizing the sum of weights.

The MST algorithm for directed graphs suggested by McDonald et al. (2005) is not fully constructive: It does not work by building structures and combining them into large structures until it finds the solution. Instead, the algorithm works by using a greedy strategy to select a candidate set of edges for the spanning tree, potentially creating cycles and forming an illegal dependency tree. A cycle elimination procedure is iteratively applied to this graph until a legal dependency tree is obtained. It is still possible to express declarative aspects of the parser with a parsing schema, although the importance of the control mechanism in eliminating cycles makes this schema less informative than other cases we have considered, and we will not discuss it in detail here. Gómez-Rodríguez (2009) gives a complete description and discussion of the schema for the MST parser.

6.5 Covington's (1990, 2001) Non-Projective Parser

Covington's non-projective parsing algorithm (Covington 1990, 2001) reads the input from left to right, establishing dependency links between the current word and previous words in the input. The parser maintains two lists: one with all the words encountered so far, and one with those that do not yet have a head assigned. A new word can be linked as a dependent of any of the words in the first list, and as a head of any of the words in the second list. The following parsing schema expresses this algorithm.

Item set: The item set is $\mathcal{I}_{CovNP} = \{[i, \langle h_1, h_2, \ldots, h_k \rangle] \mid 1 \leq h_1 \leq \ldots \leq h_k \leq i \leq n\}$ where an item $[i, \langle h_1, h_2, \ldots, h_k \rangle]$ represents the set of forests of the form $F = \{T_1, T_2, \ldots, T_k\}$ such that $head(T_j) = h_j$ for every T_j in F; the projections of the nodes $h_1, h_2, \ldots, h_k$ are pairwise disjoint, and their union is $[1..i]$.

Deduction steps: The set of deduction steps is as follows:

$$\textsc{Initter}: \frac{}{[1, \langle 1 \rangle]} \qquad \textsc{R-Link}: \frac{[i, \langle h_1, \ldots, h_{j-1}, h_j, h_{j+1}, \ldots, h_k \rangle]}{[i, \langle h_1, \ldots, h_{j-1}, h_{j+1}, \ldots, h_k \rangle]} \; (w_{h_j}, h_j) \rightarrow (w_i, i)(j < i)$$

$$\textsc{Advance}: \frac{[i, \langle h_1, \ldots, h_k \rangle]}{[i+1, \langle h_1, \ldots, h_k, i+1 \rangle]} \qquad \textsc{L-Link}: \frac{[i, \langle h_1, \ldots, h_k, i \rangle]}{[i, \langle h_1, \ldots, h_k \rangle]} \; (w_i, i) \rightarrow (w_j, j)(j < i)$$

Final items: The set of final items is $\{[n, \langle h \rangle] \mid 1 \leq h \leq n\}$, the set of items containing a forest with a single dependency tree T headed at an arbitrary position h of the string, and whose yield spans the whole input string. The time complexity of the algorithm is exponential in the input length n.

Note that this parsing schema is not correct, because Covington's algorithm does not prevent the generation of cycles in the dependency graphs it produces. Quoting Covington (2001, page 99),

> Because the parser operates one word at a time, unity can only be checked at the end of the whole process: did it produce a tree with a single root that comprises all of the words?

Therefore, a postprocessing mechanism is needed to determine which of the generated structures are, in fact, valid trees. In the parsing schema, this is reflected by the fact that the schema is complete but not sound. Nivre (2007) uses a variant of this algorithm in which cycle detection is used to avoid generating incorrect structures.

Other non-projective parsers not covered here can also be represented under the parsing schema framework. For example, Kuhlmann (2010) presents a deduction system for a non-projective parser which uses a grammar formalism called **regular dependency grammars**. This deduction system can easily be converted into a parsing schema by associating adequate semantics with items. However, we do not show this here for space reasons, because we would first have to explain the formalism of regular dependency grammars.

7. Mildly Non-Projective Dependency Parsing

For reasons of computational efficiency, many practical implementations of dependency parsing are restricted to *projective* structures. However, some natural language sentences appear to have non-projective syntactic structure, something that arises in many languages (Havelka 2007), and is particularly common in free word order languages such as Czech. Parsing without the projectivity constraint is computationally complex: Although it is possible to parse non-projective structures in quadratic time with respect to input length under a model in which each dependency decision is independent of all the others (as in the parser of McDonald et al. [2005], discussed in Section 6.4), the problem is intractable in the absence of this assumption (McDonald and Satta 2007).

Nivre and Nilsson (2005) observe that most non-projective dependency structures appearing in practice contain only small proportions of non-projective arcs. This has led to the study of sub-classes of the class of all non-projective dependency structures (Kuhlmann and Nivre 2006; Havelka 2007). Kuhlmann (2010) investigates several such classes, based on well-nestedness and gap degree constraints (Bodirsky, Kuhlmann, and Möhl 2005), relating them to lexicalized constituency grammar formalisms. Specifically, Kuhlmann shows that linear context-free rewriting systems

(LCFRS) with fan-out k (Vijay-Shanker, Weir, and Joshi 1987; Satta 1992) induce the set of dependency structures with gap degree at most $k-1$; coupled CFG in which the maximal rank of a nonterminal is k (Hotz and Pitsch 1996) induces the set of well-nested dependency structures with gap degree at most $k-1$; and finally, LTAG (Joshi and Schabes 1997) induces the set of well-nested dependency structures with gap degree at most 1.

These results establish that there are polynomial-time dependency parsing algorithms for well-nested structures with bounded gap degree, because such parsers exist for their corresponding lexicalized constituency-based formalisms. Developing efficient dependency parsing strategies for these sets of structures has considerable practical interest, in particular, making it possible to parse directly with dependencies in a data-driven manner rather than indirectly by constructing intermediate constituency grammars and extracting dependencies from constituency parses. In this section, we make four contributions to this enterprise.

Firstly, we define a parser for well-nested structures of gap degree 1, and prove its correctness. The parser runs in time $O(n^7)$, the same complexity as the best existing algorithms for LTAG (Eisner and Satta 2000), and can be optimized to $O(n^6)$ in the non-lexicalized case. Secondly, we generalize our algorithm to any well-nested dependency structure with gap degree at most k, resulting in an algorithm with time complexity $O(n^{5+2k})$. Thirdly, we generalize the previous parsers in order to include ill-nested structures with gap degree at most k satisfying certain constraints, giving a parser that runs in time $O(n^{4+3k})$. Note that parsing unrestricted ill-nested structures, even when the gap degree is bounded, is NP-complete: These structures are equivalent to LCFRS for which the recognition problem is NP-complete (Satta 1992). Finally, we characterize the set of structures covered by this parser, which we call **mildly ill-nested** structures, and show that it includes all the trees present in a number of dependency treebanks.

We now define the concepts of gap degree and well-nestedness (Kuhlmann and Nivre 2006). Let T be a dependency tree for the string $w_1 \dots w_n$:

Definition 5
The **gap degree** of a node k in T is the minimum $g \in (\mathbb{N} \cup \{0\})$ such that $\lfloor k \rfloor$ (the projection of the node k) can be written as the union of $g+1$ intervals, that is, the number of discontinuities in $\lfloor k \rfloor$. The gap degree of the dependency tree T is the maximum of the gap degrees of its nodes.

Note that T has gap degree 0 if and only if T is projective.

Definition 6
The **subtree** induced by the node u in a dependency tree T is the tree $T_u = (\lfloor u \rfloor, E_u)$ where $E_u = \{i \rightarrow j \in E \mid j \in \lfloor u \rfloor\}$. The subtrees induced by nodes p and q are **interleaved** if $\lfloor p \rfloor \cap \lfloor q \rfloor = \emptyset$ and there are nodes $i, j \in \lfloor p \rfloor$ and $k, l \in \lfloor q \rfloor$ such that $i < k < j < l$. A dependency tree T is **well-nested** if it does not contain two interleaved subtrees, and a tree that is not well-nested is said to be **ill-nested**.

Projective trees are always well-nested, but well-nested trees are not always projective.

7.1 The WG_1 Parser

We now define WG_1, a polynomial-time parser for well-nested dependency structures of gap degree at most 1. In this and subsequent schemata, each dependency forest in

an item is a singleton set containing a dependency tree, so we will not make explicit mention of these forests, referring directly to their trees instead. Also note that in the parsers in this section we use D-rules to express parsing decisions, so dependency trees are assumed to be taken from the set of trees licensed by a given set of D-rules. The schema for the WG_1 parser is defined as follows:

Item set: The item set is $\mathcal{I}_{WG1} = \mathcal{I}_1 \cup \mathcal{I}_2$, with

$$\mathcal{I}_1 = \{[i,j,h,\diamond,\diamond] \mid i,j,h \in \mathbb{N}, 1 \leq h \leq n, 1 \leq i \leq j \leq n, h \neq j, h \neq i-1\}$$

where each item of the form $[i,j,h,\diamond,\diamond]$ represents the set of all well-nested dependency trees with gap degree at most 1, rooted at h, and such that $\lfloor h \rfloor = \{h\} \cup [i..j]$, and

$$\mathcal{I}_2 = \{[i,j,h,l,r] \mid i,j,h,l,r \in \mathbb{N}, 1 \leq h \leq n, 1 \leq i < l \leq r < j \leq n, h \neq j, h \neq i-1, h \neq l-1, h \neq r\}$$

where each item of the form $[i,j,h,l,r]$ represents the set of all well-nested dependency trees rooted at h such that $\lfloor h \rfloor = \{h\} \cup ([i..j] \setminus [l..r])$, and all the nodes (except possibly h) have gap degree at most 1. We call items of this form **gapped items**, and the interval $[l..r]$ the **gap** of the item. Figure 7 shows two WG_1 items, one from $\mathcal{I}_1$ and the other from $\mathcal{I}_2$, together with one of the trees contained in each of them. Note that the constraints $h \neq j, h \neq i+1, h \neq l-1, h \neq r$ are added to items to avoid redundancy in the item set. Because the result of the expression $\{h\} \cup ([i..j] \setminus [l..r])$ for a given head can be the same for different sets of values of i,j,l,r, we restrict these values so that we cannot get two different items representing the same dependency structures. Items ι violating these constraints always have an alternative representation that does not violate them, which we can express with a normalizing function $nm(\iota)$ as follows:

$nm([i,j,j,l,r]) = [i,j-1,j,l,r]$ (if $r \leq j-1$ or $r = \diamond$), or $[i,l-1,j,\diamond,\diamond]$ (if $r = j-1$).

$nm([i,j,l-1,l,r]) = [i,j,l-1,l-1,r]$(if $l > i+1$), or $[r+1,j,l-1,\diamond,\diamond]$ (if $l = i+1$).

$nm([i,j,i-1,l,r]) = [i-1,j,i-1,l,r]$.

$nm([i,j,r,l,r]) = [i,j,r,l,r-1]$ (if $l < r$), or $[i,j,r,\diamond,\diamond]$ (if $l = r$).

$nm([i,j,h,l,r]) = [i,j,h,l,r]$ for all other items.

When defining the deduction steps for this and other parsers, we assume that they always produce normalized items. For clarity, we do not explicitly write this in the deduction steps, writing ι instead of $nm(\iota)$ as antecedents and consequents of steps.

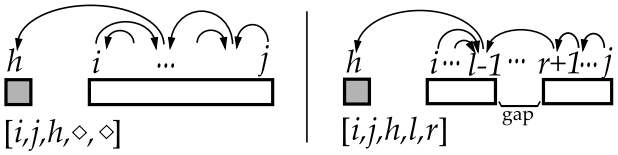

Figure 7
Representation of the WG_1 items $[i,j,h,\diamond,\diamond]$ and $[i,j,h,l,r]$, each together with one of the dependency structures contained in it.

Initial items: The set of initial items (hypotheses) is defined as the set

$$\mathcal{H} = \{[h, h, h, \diamond, \diamond] \mid h \in [1..n]\}$$

where each item $[h, h, h, \diamond, \diamond]$ represents the set containing the trivial dependency tree consisting of a single node h and no links. This is the same set of hypotheses used by the parsers defined in previous sections, but we use the notation $[h, h, h, \diamond, \diamond]$ rather than $[h, h, h]$ here for convenience when defining deduction steps. The same set of hypotheses is used for all the mildly non-projective parsers, so we do not make it explicit for subsequent schemata. Note that initial items are separate from the item set $\mathcal{I}_{WG1}$ and not subject to its constraints, so they do not require normalization.

Final items: The set of final items for strings of length n in WG_1 is defined as the set

$$\mathcal{F} = \{[1, n, h, \diamond, \diamond] \mid h \in [1..n]\},$$

which is the set of the items in $\mathcal{I}_{WG1}$ containing dependency trees for the complete input string (from position 1 to n), with their head at any position h.

Deduction steps: The deduction steps of the WG_1 parser are the following:

$$\textsc{Link Ungapped:}\ \frac{[h_1, h_1, h_1, \diamond, \diamond] \quad [i_2, j_2, h_2, \diamond, \diamond]}{[i_2, j_2, h_1, \diamond, \diamond]}\ (w_{h_2}, h_2) \rightarrow (w_{h_1}, h_1)$$

$$\text{such that } h_2 \in [i_2..j_2] \wedge h_1 \notin [i_2..j_2]$$

$$\textsc{Link Gapped:}\ \frac{[h_1, h_1, h_1, \diamond, \diamond] \quad [i_2, j_2, h_2, l_2, r_2]}{[i_2, j_2, h_1, l_2, r_2]}\ (w_{h_2}, h_2) \rightarrow (w_{h_1}, h_1)$$

$$\text{such that } h_2 \in [i_2..j_2] \setminus [l_2..r_2] \wedge h_1 \notin [i_2..j_2] \setminus [l_2..r_2]$$

$$\textsc{Combine Ungapped:}\ \frac{[i, j, h, \diamond, \diamond] \quad [j+1, k, h, \diamond, \diamond]}{[i, k, h, \diamond, \diamond]} \qquad \textsc{Combine Opening Gap:}\ \frac{[i, j, h, \diamond, \diamond] \quad [k, l, h, \diamond, \diamond]}{[i, l, h, j+1, k-1]}\ j < k-1$$

$$\textsc{Combine Keeping Gap Left:}\ \frac{[i, j, h, l, r] \quad [j+1, k, h, \diamond, \diamond]}{[i, k, h, l, r]} \qquad \textsc{Combine Keeping Gap Right:}\ \frac{[i, j, h, \diamond, \diamond] \quad [j+1, k, h, l, r]}{[i, k, h, l, r]}$$

$$\textsc{Combine Closing Gap:}\ \frac{[i, j, h, l, r] \quad [l, r, h, \diamond, \diamond]}{[i, j, h, \diamond, \diamond]} \qquad \textsc{Combine Shrinking Gap Centre:}\ \frac{[i, j, h, l, r] \quad [l, r, h, l_2, r_2]}{[i, j, h, l_2, r_2]}$$

$$\textsc{Combine Shrinking Gap Left:}\ \frac{[i, j, h, l, r] \quad [l, k, h, \diamond, \diamond]}{[i, j, h, k+1, r]} \qquad \textsc{Combine Shrinking Gap Right:}\ \frac{[i, j, h, l, r] \quad [k, r, h, \diamond, \diamond]}{[i, j, h, l, k-1]}$$

The WG_1 parser proceeds bottom–up, building dependency subtrees and combining them into larger subtrees, until a complete dependency tree for the input sentence is

found. The parser logic specifies how items corresponding to the subtree induced by a particular node are inferred, given the items for the subtrees induced by the direct dependents of that node. Suppose that, in a complete dependency analysis for a sentence $w_1 \dots w_n$, the node h has $d_1 \dots d_p$ as direct dependents (i.e., we have dependency links $d_1 \rightarrow h, \dots, d_p \rightarrow h$). The item corresponding to the subtree induced by h is obtained from the ones corresponding to the subtrees induced by $d_1 \dots d_p$ as follows.

First, apply the LINK UNGAPPED or LINK GAPPED step to each of the items corresponding to the subtrees induced by the direct dependents, and to the hypothesis $[h, h, h, \diamond, \diamond]$. We infer p items representing the result of linking each of the dependent subtrees to the new head h. Second, apply the various COMBINE steps to join all items obtained in the previous step into a single item. The COMBINE steps perform a union operation between subtrees. Therefore, the result is a dependency tree containing all the dependent subtrees, and with all of them linked to h—this is the subtree induced by h. This process is applied repeatedly to build larger subtrees, until, if the parsing process is successful, a final item is found containing a dependency tree for the complete sentence. A graphical representation of this process is shown in Figure 8.

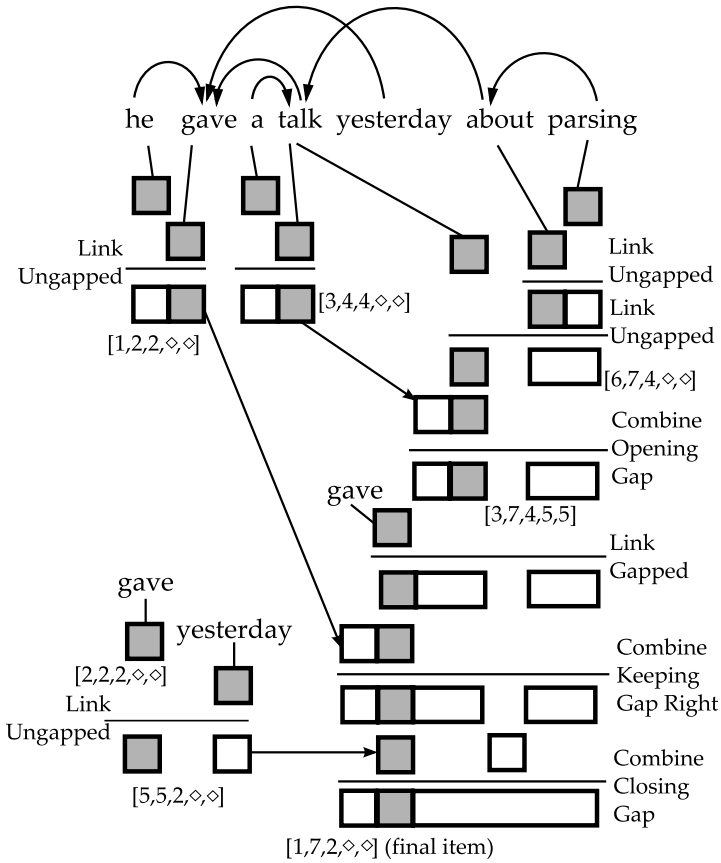

Figure 8
Example WG_1 parse, following the notation of Figure 7. LINK steps link an item to a new head, while COMBINE steps are used to join a pair of items sharing the same head. Different COMBINE steps correspond to different relative positions of items that can be joined and their gaps.

The WG_1 schema provides an abstract mechanism for finding all the dependency structures in the class of well-nested structures of gap degree at most 1, for an input string under a set of D-rules. Concrete implementations of the schema may use probabilistic models or machine learning techniques to make the linking decisions associated with the D-rules, as explained in Section 3.1. The definition of such statistical models for guiding the execution of schemata falls outside the scope of this article.

7.2 Proof of Correctness for WG_1

We define a set of **coherent** items for the schema, in such a way that final items in this set satisfy the general definition of coherent final items; and then prove the stronger claims that all derivable items are coherent and all coherent items are derivable. The full correctness proof has previously been published (Gómez-Rodríguez, Weir, and Carroll 2008; Gómez-Rodríguez 2009), so for reasons of space we only sketch the proof here.

To define the set of coherent items for WG_1, we provide a definition of the trees that these items must contain. Let T be a well-nested dependency tree headed at a node h, with all its edges licensed by our set of D-rules. We call such a tree a **properly formed tree** for the algorithm WG_1 if it satisfies the following conditions.

1. $\lfloor h \rfloor$ is either of the form $\{h\} \cup [i..j]$ or $\{h\} \cup ([i..j] \setminus [l..r])$.
2. All the nodes in T have gap degree at most 1 except for h, which can have gap degree up to 2.

An item $[i, j, h, l, r] \in \mathcal{I}_{WG1}$ is **coherent** if it contains a properly formed tree headed at h, such that $\lfloor h \rfloor = \{h\} \cup ([i..j] \setminus [l..r])$. Similarily for items of the form $[i, j, h, \diamond, \diamond]$, where $\lfloor h \rfloor = \{h\} \cup [i..j]$. A **coherent final item** $[1, n, h, \diamond, \diamond]$ for an input string contains at least one well-nested parse of gap degree ≤ 1 for that string. With these sets of coherent and coherent final items, we prove the soundness and completeness of WG_1.

Theorem 6
WG_1 is sound.

Proof 6
Proving the soundness of the WG_1 parser involves showing that all derivable final items are coherent. We do this by proving the stronger claim that all derivable items are coherent. As in previous proofs, this is done by showing that each deduction step in the parser infers a coherent consequent item when applied to coherent antecedents. We proceed step by step, showing that if each of the antecedents of a given step contains at least one properly formed tree, we obtain a properly formed tree that is an element of the corresponding consequent. In the case of LINK steps, this properly formed consequent tree is obtained by creating a dependency link between the heads of the properly formed antecedent trees; for COMBINE steps, it is obtained from the union of the antecedent trees. To prove that these consequent trees are properly formed, we show that they are well-nested, have a projection corresponding to the indices in the consequent item, and satisfy the gap degree constraint 2 required for the trees to be properly formed. Each of these properties is proven individually, based on the properties of the antecedent trees. ■

Theorem 7
WG_1 is complete.

Proof 7
Proving completeness of the WG_1 parser involves proving that all coherent final items in WG_1 are derivable. We show this by proving the following, stronger claim.

Lemma 1
If T is a dependency tree headed at a node h, which is a properly formed tree for WG_1, then:

1. If $\lfloor h \rfloor = \{h\} \cup [i..j]$, then the item $[i, j, h, \diamond, \diamond]$ containing T is a derivable item in the WG_1 parser.
2. If $\lfloor h \rfloor = \{h\} \cup ([i..j] \setminus [l..r])$, then the item $[i, j, h, l, r]$ containing T is a derivable item in the WG_1 parser.

This implies that all coherent final items are derivable, and therefore that WG_1 is complete. The lemma is proven by strong induction on the number of elements in $\lfloor h \rfloor$, which we denote $\#(\lfloor h \rfloor)$.

The base case of the induction is trivial, because the case $\#(\lfloor h \rfloor) = 1$ corresponds to a tree contained in an initial item, which is derivable by definition. For the induction step, we take T to be a properly formed dependency tree rooted at a node h, such that $\#(\lfloor h \rfloor) = N$ for some $N > 1$. Lemma 1 holds for T if it holds for every properly formed dependency tree T' rooted at h' such that $\#(\lfloor h' \rfloor) < N$. Let p be the number of direct children of h in the tree T. We have $p \geq 1$, because by hypothesis $\#(\lfloor h \rfloor) > 1$. With this, the induction step proof is divided into two cases, according to whether $p = 1$ or $p > 1$.

When p = 1, the item that Lemma 1 associates with the subtree of T induced by the single direct dependent of h is known to be derivable by the induction hypothesis. It can be shown case by case that the item corresponding to h by Lemma 1 can be inferred using LINK steps, thus completing the case for p = 1. For $p > 1$, we use the concept of **order annotations** (Kuhlmann and Möhl 2007; Kuhlmann 2010). Order annotations are strings that encode the precedence relation between the nodes of a dependency tree. The order annotation for a given node encodes the shape (with respect to this precedence relation) of the projection of each of the children of that node, that is, the number of intervals in each projection, the number of gaps, and the way in which intervals and gaps are interleaved. The concepts of projectivity, gap degree, and well-nestedness are associated with particular constraints on order annotations.

The completeness proof for $p > 1$ is divided into cases according to the order annotation of the head h. The fact that the tree T is properly formed imposes constraints on the form of this order annotation. With this information, we divide the possible order annotations into a number of cases. Using the induction hypotheses and some relevant properties of order annotations we find that, for each of this cases, we can find a sequence of COMBINE steps to infer the item corresponding to T from smaller coherent items. ■

7.3 Computational Complexity of WG_1

The time complexity of WG_1 is $O(n^7)$, as the step COMBINE SHRINKING GAP CENTRE works with seven free string positions. This complexity with respect to the length of the

input is as expected for this set of structures, because Kuhlmann (2010) shows that they are equivalent to LTAG, and the best existing parsers for this formalism also perform in $O(n^7)$ (Eisner and Satta 2000).[9] Note that the COMBINE step that is the bottleneck only uses seven indices, not any additional entities such as D-rules. Hence, the $O(n^7)$ complexity does not involve additional factors relating to grammar size.

Given unlexicalized D-rules specifying the possibility of dependencies between pairs of categories rather than pairs of words, a variant of this parser can be constructed with time complexity $O(n^6)$, as with parsers for unlexicalized TAG. We expand the item set with unlexicalized items of the form $[i, j, C, l, r]$, where C is a category, distinct from the existing items $[i, j, h, l, r]$. Steps in the parser are duplicated, to work both with lexicalized and unlexicalized items, except for the LINK steps, which always work with a lexicalized item and an unlexicalized hypothesis to produce an unlexicalized item, and the COMBINE SHRINKING GAP steps, which work only with unlexicalized items. Steps are added to obtain lexicalized items from their unlexicalized equivalents by binding the head to particular string positions. Finally, we need certain variants of the COMBINE SHRINKING GAP steps that take two unlexicalized antecedents and produce a lexicalized consequent; an example is the following:

$$\text{COMBINE SHRINKING GAP CENTRE L: } \frac{[i, j, C, l, r] \qquad [l+1, r, C, l_2, r_2]}{[i, j, l, l_2, r_2]} \; cat(w_l){=}C$$

Although this version of the algorithm reduces time complexity to $O(n^6)$, it also adds a factor related to the number of categories, as well as constant factors due to having more kinds of items and steps than the original WG_1 algorithm.

7.4 The WG_k Parser

The WG_1 parsing schema can be generalized to obtain a parser for all well-nested dependency structures with gap degree bounded by a constant k ($k \geq 1$), which we call the WG_k parser. We extend the item set so that it contains items with up to k gaps, and modify the deduction steps to work with these multi-gapped items.

Item set: The item set for the WG_k parsing schema is

$$\mathcal{I}_{WGk} = \{[i, j, h, \langle (l_1, r_1), \ldots, (l_g, r_g) \rangle]\}$$

where $i, j, h \in (\mathbb{N} \cup \{0\}), 0 \leq g \leq k, 1 \leq h \leq n, 1 \leq i \leq j \leq n, h \neq j, h \neq i-1$; and for each $p \in \{1, 2, \ldots, g\}$: $l_p, r_p \in \mathbb{N}, i < l_p \leq r_p < j, r_p < l_{p+1} - 1, h \neq l_p - 1, h \neq r_p$. An item $[i, j, h, \langle (l_1, r_1), \ldots, (l_g, r_g) \rangle]$ represents the set of all well-nested dependency trees rooted at h such that $\lfloor h \rfloor = \{h\} \cup ([i..j] \setminus \bigcup_{p=1}^{g} [l_p..r_p])$, where each interval $[l_p..r_p]$ is called a gap. The constraints $h \neq j, h \neq i+1, h \neq l_p - 1, h \neq r_p$ are added to avoid redundancy, and normalization is defined as in WG_1.

Final items: The set of final items is defined as the set $\mathcal{F} = \{[1, n, h, \langle\rangle] \mid h \in [1..n]\}$. Note that this set is the same as in WG_1, as these are the items that we denoted $[1, n, h, \diamond, \diamond]$ in that parser.

9 Although standard TAG parsing algorithms run in time $O(n^6)$ with respect to the input length, they also have a complexity factor related to grammar size. Eisner and Satta (2000) show that, in the case of lexicalized TAG, this factor is a function of the input length n; hence the additional complexity.

Deduction steps: The parser has the following deduction steps:

$$\textsc{Link:}\ \frac{[h_1, h_1, h_1, \langle\rangle] \qquad [i_2, j_2, h_2, \langle (l_1, r_1), \ldots, (l_g, r_g)\rangle]}{[i_2, j_2, h_1, \langle (l_1, r_1), \ldots, (l_g, r_g)\rangle]}\ (w_{h_2}, h_2) \rightarrow (w_{h_1}, h_1)$$

$$\text{such that } h_2 \in [i_2..j_2] \setminus \bigcup_{p=1}^{g} [l_p..r_p] \wedge h_1 \notin [i_2..j_2] \setminus \bigcup_{p=1}^{g} [l_p..r_p]$$

$$\textsc{Combine Opening Gap:}\ \frac{\begin{array}{c}[i, l_q - 1, h, \langle (l_1, r_1), \ldots, (l_{q-1}, r_{q-1})\rangle] \\ [r_q + 1, m, h, \langle (l_{q+1}, r_{q+1}), \ldots, (l_g, r_g)\rangle]\end{array}}{[i, m, h, \langle (l_1, r_1), \ldots, (l_g, r_g)\rangle]}\ g \leq k \wedge l_q \leq r_q$$

$$\textsc{Combine Keeping Gaps:}\ \frac{\begin{array}{c}[i, j, h, \langle (l_1, r_1), \ldots, (l_q, r_q)\rangle] \\ [j + 1, m, h, \langle (l_{q+1}, r_{q+1}), \ldots, (l_g, r_g)\rangle]\end{array}}{[i, m, h, \langle (l_1, r_1), \ldots, (l_g, r_g)\rangle]}\ g \leq k$$

$$\textsc{Combine Shrinking Gap Left:}\ \frac{\begin{array}{c}[i, j, h, \langle (l_1, r_1), \ldots, (l_q, r_q), (l', r_s), (l_{s+1}, r_{s+1}), \ldots, (l_g, r_g)\rangle] \\ [l', l_s - 1, h, \langle (l_{q+1}, r_{q+1}), \ldots, (l_{s-1}, r_{s-1})\rangle]\end{array}}{[i, j, h, \langle (l_1, r_1), \ldots, (l_g, r_g)\rangle]}\ g \leq k$$

$$\textsc{Combine Shrinking Gap Right:}\ \frac{\begin{array}{c}[i, j, h, \langle (l_1, r_1), \ldots, (l_{q-1}, r_{q-1}), (l_q, r'), (l_s, r_s), \ldots, (l_g, r_g)\rangle] \\ [r_q + 1, r', h, \langle (l_{q+1}, r_{q+1}), \ldots, (l_{s-1}, r_{s-1})\rangle]\end{array}}{[i, j, h, \langle (l_1, r_1), \ldots, (l_g, r_g)\rangle]}\ g \leq k$$

$$\textsc{Combine Shrinking Gap Centre:}\ \frac{\begin{array}{c}[i, j, h, \langle (l_1, r_1), \ldots, (l_q, r_q), (l', r'), (l_s, r_s), \ldots, (l_g, r_g)\rangle] \\ [l', r', h, \langle (l_{q+1}, r_{q+1}), \ldots, (l_{s-1}, r_{s-1})\rangle]\end{array}}{[i, j, h, \langle (l_1, r_1), \ldots, (l_g, r_g)\rangle]}\ g \leq k$$

As expected, the WG_1 parser corresponds to WG_k for $k = 1$. WG_k works in the same way as WG_1, except that COMBINE steps can create items with more than one gap. In all the parsers described in this section, COMBINE steps may be applied in different orders to produce the same result, causing spurious ambiguity. In WG_1 and WG_k, this can be avoided when implementing the schemata by adding flags to items so as to impose a particular order on the execution of these steps.

7.5 Proof of Correctness for WG_k

The proof of correctness for WG_k is analogous to that of WG_1, but generalizing the definition of properly formed trees to a higher gap degree. A properly formed tree in WG_k is a dependency tree T, headed at node h, such that the following hold.

1. $\lfloor h \rfloor$ is of the form $\{h\} \cup ([i..j] \setminus \bigcup_{p=1}^{g} [l_p..r_p])$, with $0 \leq g \leq k$.
2. All the nodes in T have gap degree at most k except for h, which can have gap degree up to $k + 1$.

With this, we define coherent items and coherent final items as for WG_1. Soundness is shown as for WG_1, changing the constraints on nodes so that any node can have gap degree up to k and the head of a properly formed tree can have gap degree $k + 1$.

Completeness is shown by induction on #($\lfloor h \rfloor$). The base case is the same as for WG_1, and for the induction step, we consider the direct children $d_1 \dots d_p$ of h. The case where p = 1 is proven by using LINK steps just as in WG_1. In the case for p $\geq$ 1, we also base our proof on the order annotation for h, but we have to take into account that the set of possible annotations is larger when we allow the gap degree to be greater than 1, so we must consider more cases in this part of the proof.

7.6 Computational Complexity of WG_k

The WG_k parser runs in time $O(n^{5+2k})$. As in the case of WG_1, the step with most free variables is COMBINE SHRINKING GAP CENTRE with $5 + 2k$ free indices. Again, this complexity result is in line with what could be expected from previous research in constituency parsing: Kuhlmann (2010) shows that the set of well-nested dependency structures with gap degree at most k is closely related to coupled CFG in which the maximal rank of a nonterminal is $k + 1$. The constituency parser defined by Hotz and Pitsch (1996) for these grammars also adds an n^2 factor for each unit increment of k. Note that a small value of k appears to be sufficient to account for the vast majority of the non-projective sentences found in natural language treebanks. For instance, the Prague Dependency Treebank (Hajič et al. 2006) contains no structures with gap degree greater than 4. Thus, a WG_4 parser would be able to analyze all the well-nested structures in this treebank, which represent 99.89% of the total (WG_1 would be able to parse 99.49%). Increasing k beyond 4 would not produce further improvements in coverage.

7.7 Parsing Ill-Nested Structures: MG_1 and MG_k

The WG_k parser analyzes dependency structures with bounded gap degree as long as they are well-nested. Although this covers the vast majority of the structures that occur in natural language treebanks (Kuhlmann and Nivre 2006), a significant number of sentences contain ill-nested structures. Maier and Lichte (2011) provide examples of some linguistic phenomena that cause ill-nestedness. Unfortunately, the general problem of parsing ill-nested structures is NP-complete, even when the gap degree is bounded. This set of structures is closely related to LCFRS with bounded fan-out and unbounded production length, and parsing in this formalism is known to be NP-complete (Satta 1992). The reason for this complexity is the problem of *unrestricted crossing configurations*, appearing when dependency subtrees are allowed to interleave in every possible way.

Ill-nested structures can be parsed in polynomial time with bounds on the gap degree and the number of dependents allowed per node: Kuhlmann (2010) presents a parser based on this idea, using a kind of grammar that resembles LCFRS, called *regular dependency grammar*. This parser is exponential in the gap degree, as well as in the maximum number of dependents allowed per node: Its complexity is $O(n^{k(m+1)})$, where k is the maximum gap degree and m is the maximum number of dependents per node. In contrast, the parsers presented here are data-driven and thus do not need an explicit grammar. Furthermore, they are able to parse dependency structures with any number of dependents per node, and their computational complexity is independent of this parameter m.

In line with the observation that most non-projective structures appearing in practice are only "slightly" non-projective (Nivre and Nilsson 2005), we characterize a sense in which the structures appearing in treebanks are only "slightly" ill-nested. We

generalize the algorithms WG_1 and WG_k to parse a proper superset of the set of well-nested structures in polynomial time, and give a characterization of this new set of structures, which includes all the structures in several dependency treebanks.

The WG_k parser for well-nested structures presented previously is based on a bottom–up process, where LINK steps are used to link completed subtrees to a head, and COMBINE steps are used to join subtrees governed by a common head to obtain a larger structure. As WG_k is a parser for well-nested structures of gap degree up to k, its COMBINER steps correspond to all the ways in which we can join two sets of sibling subtrees meeting these constraints, and having a common head, into another. Therefore, this parser does not use COMBINER steps that produce interleaved subtrees, because these would generate items corresponding to ill-nested structures.

We obtain a polynomial parser for a larger set of structures of gap degree at most k, including some ill-nested ones, by having COMBINER steps representing all ways in which two sets of sibling subtrees of gap degree at most k with a common head can be joined into another, including those producing interleaved subtrees. This does not mean that we build every possible ill-nested structure. Some structures with complex crossed configurations have gap degree k, but cannot be built by combining two structures of that gap degree. More specifically, this algorithm will parse a dependency structure (well-nested or not) if there exists a *binarization* of that structure that has gap degree at most k. The parser works by implicitly finding such a binarization, because COMBINE steps are always applied to two items and no intermediate item generated by them can exceed gap degree k (not counting the position of the head in the projection).

Definition 7
Let $w_1 \dots w_n$ be a string, and T a dependency tree headed at a node h. A **binarization** of T is a tree B in which each node has at most two children, such that:

1. Each node in B can be unlabelled, or labelled with a word position i. Several nodes may have the same label (in contrast to the dependency graphs, where a word occurrence cannot appear twice in the graph).
2. A node labelled i is a descendant of j in B if and only if $i \rightarrow^\star j$ in T.

The projection of a node in a binarization is the set of reflexive-transitive children of that node. With this, condition (2) of Definition 7 can be rewritten $i \in \lfloor j \rfloor_B \Leftrightarrow i \in \lfloor j \rfloor_T$, and the gap degree of a binarization can be defined as with a dependency structure, allowing us to define mildly ill-nested structures as follows.

Definition 8
A dependency structure is **mildly ill-nested** for gap degree k if it has at least one binarization of gap degree $\leq k$. Otherwise, it is **strongly ill-nested** for gap degree k.

The set of mildly ill-nested structures for gap degree k includes all well-nested structures with gap degree up to k. We define MG_1, a parser for mildly ill-nested structures for gap degree 1, as follows.

Item set and final item set: The item set and the final item set are the same as for WG_1, except that items can contain any mildly ill-nested structures for gap degree 1, instead of being restricted to well-nested structures.

Deduction steps: Deduction steps include those in WG_1, plus the following.

$$\textsc{Combine Interleaving:}\ \frac{[i,j,h,l,r]}{[l,k,h,r+1,j]}\quad [i,k,h,\diamond,\diamond]$$

$$\textsc{Combine Interleaving Gap C:}\ \frac{[i,j,h,l,r]}{[l,k,h,m,j]}\ m<r+1\quad [i,k,h,m,r]$$

$$\textsc{Combine Interleaving Gap L:}\ \frac{[i,j,h,l,r]}{[l,k,h,r+1,u]}\ u>j\quad [i,k,h,j+1,u]$$

$$\textsc{Combine Interleaving Gap R:}\ \frac{[i,j,h,l,r]}{[k,m,h,r+1,j]}\ k>l\quad [i,m,h,l,k-1]$$

These extra COMBINE steps allow the parser to combine interleaved subtrees with simple crossing configurations. The MG_1 parser still runs in $O(n^7)$, as these new steps do not use more than seven string positions. To generalize this algorithm to mildly ill-nested structures for gap degree k, we add a COMBINE step for every possible way of joining two structures of gap degree at most k into another. This is done in a systematic way by considering a set of strings over an alphabet of three symbols: a and b to represent intervals of words in the projection of each of the structures, and g to represent intervals that are not in the projection of either of the structures and will correspond to gaps in the joined structure. The legal combinations of structures for gap degree k will correspond to strings where symbols a and b each appear at most $k+1$ times, g appears at most k times and is not the first or last symbol, and there is no more than one consecutive appearance of any symbol. Given a string of this form, of length n, with a's located at positions $a_1 \ldots a_p (1 \leq a_1 < \ldots < a_p \leq n)$, b's at positions $b_1 \ldots b_q (1 \leq b_1 < \ldots < b_q \leq n)$, and g's at positions $g_1 \ldots g_r (2 \leq g_1 < \ldots < g_r \leq n-1)$, such that $p+q+r=n$, the corresponding COMBINE step is as follows.

$$\frac{\begin{array}{c}[i_{a_1}, i_{a_p+1}-1, h, \langle (i_{a_1+1}, i_{a_2}-1), \ldots, (i_{a_{p-1}+1}, i_{a_p}-1)\rangle] \\ [i_{b_1}, i_{b_q+1}-1, h, \langle (i_{b_1+1}, i_{b_2}-1), \ldots, (i_{b_{q-1}+1}, i_{b_q}-1)\rangle]\end{array}}{[i_{min(a_1,b_1)}, i_{max(a_p+1,b_q+1)}-1, h, \langle (i_{g_1}, i_{g_1+1}-1), \ldots, (i_{g_r}, i_{g_r+1}-1)\rangle]}$$

For example, the COMBINE INTERLEAVING GAP C step in MG_1 is obtained from the string *abgab*. Therefore, we define the parsing schema for MG_k, a parser for mildly ill-nested structures for gap degree k, as the schema where the item set is the same as that of WG_k, except that items now contain mildly ill-nested structures for gap degree k; and the set of deduction steps consists of the LINK step in WG_k, plus a set of COMBINE steps obtained as explained herein.

7.8 Computational Complexity of MG_k

Because the string used to generate a COMBINER step can have length at most $3k+2$, and the resulting step contains an index for each symbol of the string plus two extra indices, the MG_k parser has complexity $O(n^{3k+4})$ with respect to the length of the input. Note that this expression denotes the complexity with respect to n of the MG_k parser obtained for a given k: Taking k to be a variable would add an additional $O(3^{3k})$ complexity factor, because the number of different COMBINER steps that can be applied to a given item grows exponentially with k.

7.9 Proof of Correctness for MG_k

As for previous parsers, we only show here a sketch of the proof that MG_k is correct. The detailed proof has been published previously (Gómez-Rodríguez, Weir, and Carroll 2008; Gómez-Rodríguez 2009).

Theorem 8
MG_k is correct.

Proof 8
As with WG_k, we define the sets of properly formed trees and coherent items for this algorithm. Let T be a dependency tree headed at a node h. We call such a tree a **properly formed tree** for the algorithm MG_k if it satisfies the following.

1. $\lfloor h \rfloor$ is of the form $\{h\} \cup ([i..j] \setminus \bigcup_{p=1}^{g} [l_p..r_p])$, with $0 \leq g \leq k$.
2. There is a binarization of T such that all the nodes in it have gap degree at most k except for its root node, which can have gap degree up to $k+1$.

The sets of coherent and coherent final items are defined as in previous proofs. Soundness is shown as for previous algorithms, where we show that consequent trees are properly formed by building a binarization for them from the binarizations obtained from antecedent items. This part of the proof involves imposing additional constraints on binarizations, which are useful to provide a suitable way of combining binarizations obtained from antecedents of steps. Completeness is proven by showing the following, stronger claim.

Proposition 1
Let T be a dependency tree headed at node h, and properly formed for MG_k. Then, if $\lfloor h \rfloor = \{h\} \cup ([i..j] \setminus \bigcup_{p=1}^{g} [l_p..r_p])$, for $g \leq k$, the item $[i, j, h, \langle (l_1, r_1), \ldots, (l_g, r_g) \rangle]$ containing T is derivable under this parser.
To prove this, we say that a binarization of a properly formed tree is a **well-formed binarization** for MG_k if each of its nodes has gap degree $\leq k$ except possibly the head, which can have gap degree $k+1$. We then reduce the proof to establishing the following lemma.

Lemma 2
Let B be a well-formed binarization of a dependency tree T, headed at a node h and properly formed for MG_k. If the projection of h in T is $\lfloor h \rfloor_T = \lfloor h \rfloor_B = \{h\} \cup ([i..j] \setminus \bigcup_{p=1}^{g} [l_p..r_p])$, for $g \leq k$, the item $[i, j, h, \langle (l_1, r_1), \ldots, (l_g, r_g) \rangle]$ containing T is derivable under this parser.
The lemma is shown by strong induction on the number of nodes of B (denoted $\#B$). The base case where $\#B = 1$ is trivial. For the induction step, we consider different cases depending on the number and type of children of the head node h of B. When h has a single child, we obtain the item corresponding to T from a smaller item, shown to be derivable by the induction hypothesis, by using a LINK step. Where h has two children in B, the relevant item is obtained by using a COMBINER step. ■

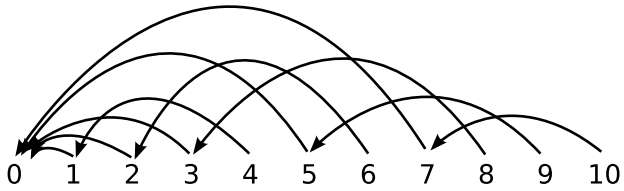

Figure 9
A structure which is strongly ill-nested for gap degree 1, but only mildly ill-nested for gap degree ≥ 2.

7.10 Mildly Ill-Nested Dependency Structures

The MG_k algorithm parses mildly ill-nested structures for gap degree k in polynomial time. The mildly ill-nested structures for gap degree k are those with a binarization of gap degree $\leq k$. Because a binarization of a dependency structure cannot have lower gap degree than the original structure, the mildly ill-nested structures for gap degree k all have gap degree $\leq k$. Given the relation between MG_k and WG_k, we know they contain all well-nested structures with gap degree $\leq k$. Figure 9 shows a structure with gap degree 1, but which is strongly ill-nested for gap degree 1. For all trees up to 10 nodes (excluding the dummy root node at position 0) all structures of gap degree k with length smaller than 10 are well-nested or only mildly ill-nested for that gap degree. Even if T is strongly ill-nested for a gap degree, there is always an $m \in \mathbb{N}$ such that T is mildly ill-nested for m (every structure can be binarized, and binarizations have finite gap degree). For example, the structure in Figure 9 is mildly ill-nested for gap degree 2. Therefore, MG_k parsers have the property of being able to parse any arbitrary dependency structure as long as we make k large enough. Structures like the one in Figure 9 do not arise in dependency treebanks. None of the treebanks for nine different languages[10] contain structures that are strongly ill-nested for their gap degree (Table 1). Therefore, in any of these treebanks, the MG_k parser can parse every sentence with gap degree at most k in time $O(n^{3k+4})$.

8. Link Grammar Schemata

Link Grammar (LG), introduced by Sleator and Temperley (1991, 1993), is a theory of syntax whose structural representation of sentences is closely related to projective dependency representations, but with some important differences.[11]

Undirected links: Like dependency formalisms, LG represents the structure of sentences as a set of links between words. However, whereas dependency links are directed, the links used in LG are undirected: There is no distinction made between heads and dependents.

10 Arabic (Hajič et al. 2004), Czech (Hajič et al. 2006), Danish (Kromann 2003), Dutch (van der Beek et al. 2002), Latin (Bamman and Crane 2006), Portuguese (Afonso et al. 2002), Slovene (Džeroski et al. 2006), Swedish (Nilsson, Hall, and Nivre 2005), and Turkish (Atalay, Oflazer, and Say 2003; Oflazer et al. 2003).

11 A complete treatment of LG is beyond the scope of this article: Schneider (1998) gives a detailed comparison of Link Grammar and dependency formalisms.

Table 1
Counts of dependency structures in treebanks for several languages, classified by projectivity, gap degree, and mild and strong ill-nestedness (for their gap degree).

Language	Structures								
	Total	Nonprojective							
		Total	By gap degree				By nestedness		
			Gap deg 1	Gap deg 2	Gap deg 3	Gap deg > 3	Well-nested	Mildly ill-nested	Strongly ill-nested
Arabic	2,995	205	189	13	2	1	204	1	0
Czech	87,889	20,353	19,989	359	4	1	20,257	96	0
Danish	5,430	864	854	10	0	0	856	8	0
Dutch	13,349	4,865	4,425	427	13	0	4,850	15	0
Latin	3,473	1,743	1,543	188	10	2	1,552	191	0
Portuguese	9,071	1,718	1,302	351	51	14	1,711	7	0
Slovene	1,998	555	443	81	21	10	550	5	0
Swedish	11,042	1,079	1,048	19	7	5	1,008	71	0
Turkish	5,583	685	656	29	0	0	665	20	0

Cycles: The sets of links representing the structure of sentences in LG may contain cycles, in contrast to dependency structures.

LG is a grammar-based formalism in which a grammar G consists of a set of words, each of which is associated with a set of linking requirements. Given a link grammar G, a set of labelled links between the words of a sentence $w_1 \dots w_n$ is said to be a **linkage** for that sentence if it satisfies the following conditions: planarity (the links do not cross when drawn above the words), connectivity (the undirected graph defined by links is connected), and satisfaction (the links satisfy the linking requirements of all the words in the input). An input sentence is considered grammatical with respect to a link grammar G if it is possible to build a linkage for the sentence with the grammar G.

The linking requirements of a word are expressed as a set of rules specifying the labels of the links that can be established between that word and other words located to its left or to its right. Linking requirements can include constraints on the order of the links, for example, a requirement can specify that a word w can be linked to two words located to its left in such a way that the link to the farthest (leftmost) word has a particular label L_2 and the link to the closest word has a label L_1.

We use the **disjunctive form** notation (Sleator and Temperley 1991) to denote linking requirements: The requirements of words are expressed as a set of **disjuncts**. Each disjunct corresponds to one way of satisfying the requirements of the word. We represent a disjunct for a word w as a pair of strings $\Delta = (R_1 R_2 \dots R_q, L_1 L_2 \dots L_p)$ where $L_1, L_2, \dots L_p$ are the labels of the links that must connect w to words located to the left of w, which must be monotonically increasing in distance from w (e.g., L_p links to the leftmost word that is directly linked to w), and $R_1, R_2, \dots R_p$ are the labels of the links that must connect w to words to its right, also monotonically increasing in distance from w (e.g., R_q links to the rightmost word that is directly connected to w).

Parsing schemata for LG parsers follow the same principles used for constituency and dependency formalisms. Item sets for LG parsing schemata are defined as sets of partial syntactic structures, which in this case are partial linkages:

Definition 9
Given a link grammar G and a string $w_1 \dots w_n$, a **partial linkage** is any edge-labeled undirected graph H such that the following conditions hold.

- The graph H has n vertices $\{v_1, \dots, v_n\}$, where each vertex v_i is a tuple (w_i, i, Δ_i) such that Δ_i is a disjunct for w_i in the grammar G.
- The graph H is connected and satisfies the planarity requirement with respect to the order $v_1, \dots, v_n$ of vertices (i.e., if we draw vertices in that order, with the given links, the links do not cross).
- Given a vertex $v_i = (w_i, i, \Delta_i)$ such that $\Delta_i = (R_1 R_2 \dots R_q, L_1 L_2 \dots L_p)$, the following conditions are satisfied:
  - Every edge $\{v_i, v_j\}$ with $j < i$ must be labelled L_s for some $1 \leq s \leq p$.
  - For every pair of edges $\{v_i, v_j\}, \{v_i, v_k\}$ such that $k < j < i$, we have that $\{v_i, v_j\}$ is labelled L_{s_1}, $\{v_i, v_k\}$ is labelled L_{s_2}, and $s_1 < s_2$.
  - Every edge $\{v_i, v_j\}$ with $j > i$ must be labelled R_t for some $1 \leq t \leq q$.
  - For every pair of edges $\{v_i, v_j\}, \{v_i, v_k\}$ such that $k > j > i$, we have that $\{v_i, v_j\}$ is labelled R_{t_1}, $\{v_i, v_k\}$ is labelled R_{t_2}, and $t_1 < t_2$.

Informally, a partial linkage is the result of choosing a particular disjunct from those available for each word in the input string, and then adding labelled links between words that are compatible with the requirements of the disjunct. **Compatibility** means that, for each word w_i associated with a disjunct $\Delta_i = (R_1 R_2 \dots R_q, L_1 L_2 \dots L_p)$, the list of labels of links connecting v_i to words to its right, ordered from the leftmost to the rightmost such word, is of the form $R_{i_1}, R_{i_2}, \dots R_{i_r}$, with $0 < i_1 < i_2 < \dots < i_r \leq q$ and, symmetrically, the list of labels of links connecting v_i to words to its left, ordered from the rightmost to the leftmost, is of the form $L_{j_1}, L_{j_2}, \dots L_{j_l}$, with $0 < j_1 < j_2 < \dots < j_l \leq p$. Given such a linkage, the right linking requirements $R_{i_1}, R_{i_2}, \dots R_{i_r}$ of the word w_i are **satisfied**, and the same for the left linking requirements $L_{j_1}, L_{j_2}, \dots L_{j_l}$ of w_i. Linking requirements that are not satisfied (e.g., the requirement of a link R_k in the disjunct associated with word w_i, with $0 < k \leq q$, such that $k \notin \{i_1, \dots, i_r\}$) are said to be **unsatisfied**.

The definition of item sets for LG resembles those for dependency parsers (Definition 4), where items come from a partition of the set of partial linkages for a given link grammar G. With these item sets, LG parsing schemata are analogous to the dependency and constituency cases. As an example of an LG parsing schema, we describe the original LG parser by Sleator and Temperley (1991), and show how projective parsing schemata, such as those seen in Section 3, can be adapted to obtain new LG parsers.

8.1 Sleator and Temperley's LG Parser

The LG parser of Sleator and Temperley (1991) is a dynamic programming algorithm that builds linkages top–down: A link between v_i and v_k is always added before links between v_i and v_j or between v_j and v_k, if $i < j < k$. This contrasts with many of the

dependency parsers seen in previous sections (Eisner 1996; Eisner and Satta 1999; Yamada and Matsumoto 2003), which build dependency graphs bottom–up.

Item set: The item set for Sleator and Temperley's parser is

$$\mathcal{I}_{SlT} = \{[i, j, \alpha \bullet \beta, \gamma \bullet \delta, B, C] \mid 0 \leq i \leq j \leq n+1 \\ \wedge B, C \in \{True, False\} \text{ and } \alpha, \beta, \gamma, \delta \text{ are strings of link labels}\}$$

where an item $[i, j, \alpha \bullet \beta, \gamma \bullet \delta, B, C]$ represents the set of partial linkages over the substring $w_i \ldots w_j$ of the input, w_i is linked to words in that substring by links labelled α and has right linking requirements β unsatisfied, w_j is linked to words in the substring by links labelled γ and has left linking requirements δ unsatisfied, B is *True* if and only if there is a direct link between w_i and w_j, and C is *True* if and only if all the inner words in the span are transitively reflexively linked to one of the end words w_i or w_j, and have all of their linking requirements satisfied.

String positions referenced by the items in $\mathcal{I}_{SlT}$ range from 0 to $n+1$. Position 0 corresponds to an artificial word w_0 (the **wall**) that the LG formalism inserts at the beginning of every input sentence (Sleator and Temperley 1991). Therefore, we assume that strings are extended with this symbol. On the other hand, position $n+1$ corresponds to a dummy word w_{n+1} that must not be linkable to any other, and is used by the parser for convenience, as in the schema for Yamada and Matsumoto's dependency parser (Section 3.4).

We use the notation $[i, \alpha, \beta]$ as shorthand for the item $[i, i, \bullet\alpha, \bullet\beta, False, True]$, which is an item used to select a particular disjunct for a word w_i.

Deduction steps: The set of deduction steps is the following:

$$\textsc{SelectDisjunct}: \frac{}{[i, R_q R_{q-1} \ldots R_1, L_p L_{p-1} \ldots L_1]}$$

such that w_i has a disjunct $\Delta = (R_1 R_2 \ldots R_q, L_1 L_2 \ldots L_p)$

$$\textsc{Initter}: \frac{[0, \alpha, \gamma] \qquad [n+1, \epsilon, \epsilon]}{[0, n+1, \bullet\alpha, \epsilon, False, False]}$$

$$\textsc{LeftPredict}: \frac{[i, j, \alpha \bullet \beta, \gamma \bullet \delta, B_1, False] \qquad [z, \sigma, \phi]}{[i, z, \bullet\beta, \bullet\phi, False, (z - i = 1)]}\ i < z < j$$

$$\textsc{LeftLinkPredict}\ (v_i \overset{b}{\longleftarrow} v_z): \frac{[i, j, \alpha \bullet b\beta, \gamma \bullet \delta, B_1, False] \qquad [z, \sigma, b\phi]}{[i, z, b \bullet \beta, b \bullet \phi, True, (z - i = 1)]}\ i < z < j$$

$$\textsc{RightPredict}: \frac{[i, j, \alpha \bullet \beta, \gamma \bullet \delta, B_1, False] \qquad [z, \sigma, \phi]}{[z, j, \bullet\sigma, \bullet\delta, False, (j - z = 1)]}\ i < z < j$$

$$\textsc{RightLinkPredict}\ (v_z \overset{b}{\longleftarrow} v_j): \frac{[i, j, \alpha \bullet \beta, \gamma \bullet b\delta, B_1, False] \qquad [z, b\sigma, \phi]}{[z, j, b \bullet \sigma, b \bullet \delta, True, (j - z = 1)]}\ i < z < j$$

$$\textsc{Completer}: \frac{\begin{array}{c}[i, j, \alpha \bullet \beta, \gamma \bullet \delta, B_1, False] \\ [i, z, \beta\bullet, \phi\bullet, B_2, True] \qquad [z, j, \sigma\bullet, \delta\bullet, B_3, True] \qquad [z, \sigma, \phi]\end{array}}{[i, j, \alpha\beta\bullet, \gamma\delta\bullet, B_1, True]}\ B_2 \vee B_3$$

An annotation of the form $(v_i \overset{b}{\longleftarrow} v_j)$ near the name of a step in this and subsequent LG schemata indicates that the corresponding step adds a link labelled b between nodes

v_i and v_j, and can be used to recover a set of complete linkages contained in a final item from each sequence of deduction steps that generates it. The SELECTDISJUNCT step chooses one of the available disjuncts for a given word w_i. The INITTER step starts the top–down process by constructing a linkage that spans the whole string $w_1 \dots w_n$, but where no links have been constructed yet. Then, the PREDICT and LINKPREDICT steps repeatedly divide the problem of finding a linkage for a substring $w_i \dots w_j$ into the smaller subproblems of finding linkages for $w_i \dots w_z$ and $w_z \dots w_j$, with $i < z < j$. In particular, the LEFTPREDICT step poses the subproblem of finding a linkage for $w_i \dots w_z$ in which w_i is not directly linked to w_z, and LEFTLINKPREDICT poses the same problem while building a direct link from w_i to w_z. RIGHTPREDICT and RIGHTLINKPREDICT proceed analogously for the substring $w_z \dots w_j$. After these two smaller linkages have been found, they are combined by a COMPLETER step into a larger linkage; the flags b and c in items are used by the COMPLETER step to ensure that its resulting item will contain a valid linkage satisfying the connectivity constraint. An example of this process, where a particular substring is parsed by using the LEFTLINKPREDICT and RIGHTPREDICT steps to divide it into smaller substrings, is shown in Figure 10. The algorithm runs in time $O(n^3)$ with respect to the length of the input, because none of its deduction steps uses more than three independent string position indices.

Final items: The set of final items is $\{[0, n+1, \alpha\bullet, \beta\bullet, B, \mathit{True}]\}$. Items of this form contain full valid linkages for the string $w_0 \dots w_n$, because having the second boolean flag set to *True* implies that their linkages for $w_0 \dots w_{n+1}$ have at most two connected components, and we have assumed that the word w_{n+1} cannot be linked to any other, so one of the components must link $w_0 \dots w_n$.

8.2 Adapting Projective Dependency Parsers to Link Grammar

We now exploit similarities between LG linkages and projective dependency structures to adapt projective dependency parsers to the LG formalism. As an example we present an LG version of the parser by Eisner (1996), but the same principles can be applied to other parsers: schemata for LG versions of the parsers by Eisner and Satta (1999) and Yamada and Matsumoto (2003) can be found in Gómez-Rodríguez (2009).

Item sets from dependency parsers are adapted to LG parsers by considering the forests contained in each dependency item. The corresponding LG items contain linkages with the same structure as these forests. For example, because each forest in an item of the form $[i, j, \mathit{False}, \mathit{False}]$ in Eisner's dependency parsing schema contains two trees

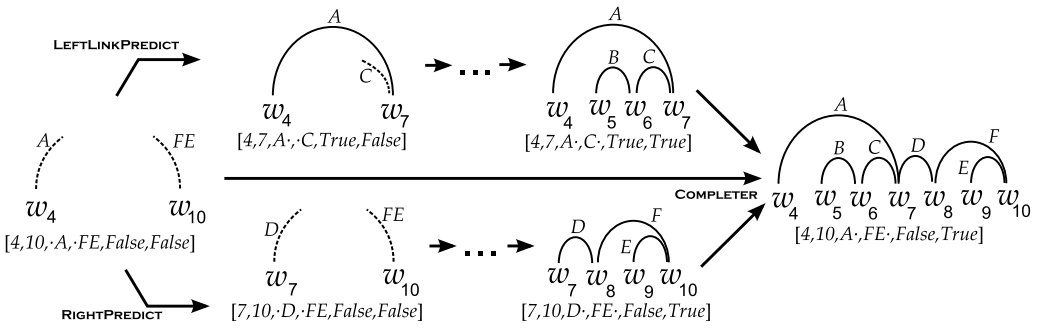

Figure 10
An example of LG parsing with the schema for Sleator and Temperley's parser.

headed at the words w_i and w_j, the analogous item in the corresponding LG parsing schema will contain linkages with two connected components, one containing the word w_i and the other containing w_j. The notion of a head is lost in the conversion because the undirected LG linkages do not make distinctions between heads and dependents. This simplifies the notation used to denote items in some cases: For instance, we do not need to make a distinction between Eisner items of the form $[i, j, True, False]$ and those of the form $[i, j, False, True]$, because their structure is the same other than the direction of the links. Therefore, items in the LG version of Eisner's parser will use a single flag, indicating whether linkages contained in them have one or two connected components.

The combining and linking steps of the dependency parsers are directly translated to LG. If the original dependency steps always produce items containing projective dependency forests, the resulting LG steps produce items with planar linkages. When the original dependency steps have constraints related to the position of the head in items (like combiner steps in Eisner's parser, where we can combine $[i, j, True, False]$ with $[j, k, True, False]$ but not with $[j, k, False, True]$), we ignore these constraints, allowing any word in a linkage to be its "head" for the purpose of linking it to other linkages.

Because projective dependency parsers do not allow cyclic structures, we add steps or remove constraints to allow cycles, so that the parsers are able to link two words that are already in the same connected component of a linkage. In the schema obtained from Eisner's parser, this is done by allowing LINK steps to be applied to items representing fully connected linkages; in the schema corresponding to Eisner and Satta's parser we allow COMBINER steps to create a link in addition to joining two linkages; and in the schema for Yamada and Matsumoto's parser we add a step that creates two links at the same time, combining the functionality of the L-LINK and R-LINK steps.

Finally, because LG is a grammar-based formalism where the set of valid linkages is constrained by disjuncts associated with words, we include disjunct information in items in order to ensure that only grammatical linkages are constructed. This is similar to the schema for Sleator and Temperley's parser, but in this case items need to specify both left and right linking requirements for each of their end words: These bottom–up parsers establish links from end words of an item to words outside the item's span (which can be to the left or to the right of the span) rather than to words inside the span (which are always to the right of the left end word, and to the left of the right end word).

Based on this, the following is an LG variant of the projective dependency parser of Eisner (1996).

Item set: The item set is

$$\mathcal{I}_{EisLG} = \{[i, j, \alpha_1 \bullet \beta_1, \alpha_2 \bullet \beta_2, \alpha_3, \alpha_4, B] \mid 0 \leq i \leq j \leq n$$
$$B \in \{True, False\} \text{ and } \alpha_1, \beta_1, \alpha_2, \beta_2, \alpha_3, \alpha_4 \text{ are strings of link labels}\}$$

where an item of the form $[i, j, \alpha_1 \bullet \beta_1, \alpha_2 \bullet \beta_2, \alpha_3, \alpha_4, B]$ represents the set of partial linkages over the substring $w_i \ldots w_j$ of the input, satisfying the following conditions.

- All words in positions k, such that $i < k < j$, have all linking requirements satisfied.
- The word in position i has left linking requirements α_3 not satisfied, and right linking requirements $\alpha_1\beta_1$, where the requirements α_1 are satisfied by links to words within the item's span, and the requirements β_1 are not satisfied. Requirements appear in α_3 and $\alpha_1\beta_1$ in increasing order of link distance.

- The word in position j has right linking requirements α_4 not satisfied, and left linking requirements $\alpha_2\beta_2$, where the requirements α_2 are satisfied by links to words within the item's span, and the requirements β_2 are not satisfied. Requirements appear in α_4 and $\alpha_2\beta_2$ in increasing order of link distance.
- The partial linkage is connected if B equals *True*, or has exactly two connected components (one containing the node v_i and the other containing v_j) if B equals *False*.

Deduction steps: The set of deduction steps for this parser is as follows:

$$\textsc{Initter:}\ \frac{}{[i, i+1, \alpha_R, \beta_L, \alpha_L, \beta_R, \mathit{False}]}\ 0 \leq i \leq n-1$$

such that w_i has a disjunct $\Delta_i = (\alpha_R, \alpha_L)$ and w_{i+1} has a disjunct $\Delta_{i+1} = (\beta_R, \beta_L)$.

$$\textsc{Link}\ (v_i \overset{b}{\longleftrightarrow} v_j)\text{:}\ \frac{[i, j, \alpha_1 \bullet b\beta_1, \alpha_2 \bullet b\beta_2, \alpha_3, \alpha_4, B]}{[i, j, \alpha_1 b \bullet \beta_1, \alpha_2 b \bullet \beta_2, \alpha_3, \alpha_4, \mathit{True}]}$$

$$\textsc{Combine:}\ \frac{[i, j, \alpha_1 \bullet \beta_1, \alpha_2 \bullet, \alpha_3, \alpha_4, B_1] \quad [j, k, \alpha_4 \bullet, \gamma_2 \bullet \delta_2, \alpha_2, \delta_4, B_2]}{[i, k, \alpha_1 \bullet \beta_1, \gamma_2 \bullet \delta_2, \alpha_3, \delta_4, B_1 \wedge B_2]}\ B_1 \vee B_2$$

These steps resemble those in the schema for Eisner's dependency parser, with the exception that the LINK step is able to build links on items that contain fully connected linkages (equivalent to the $[i, j, \mathit{True}, \mathit{False}]$ and $[i, j, \mathit{False}, \mathit{True}]$ items of the dependency parser). A version of the parser restricted to acyclic linkages can be obtained by adding the constraint that B must equal *False* in the LINK step.

Final items: The set of final items is $\{[0, n, \alpha\bullet, \beta\bullet, \epsilon, \epsilon, \mathit{True}]\}$, corresponding to the set of items containing fully connected linkages for the whole input string.

LG parsing schemata based on the parsers of Eisner and Satta (1999) and Yamada and Matsumoto (2003) are not shown here for space reasons, but are presented by Gómez-Rodríguez (2009). The relationships between these three LG parsing schemata are the same as the corresponding dependency parsing schemata, that is, the LG variants of Eisner and Satta's and Yamada and Matsumoto's dependency parsers are step contractions of the LG variant of Eisner's parser. As with the algorithm of Sleator and Temperley, these bottom–up LG parsers run in cubic time with respect to input length.

9. Conclusions and Future Work

The parsing schemata formalism of Sikkel (1997) has previously been used to define, analyze, and compare algorithms for constituency-based parsing. We have shown how to extend the formalism to dependency parsers, as well as the related Link Grammar formalism.

Deductive approaches have been used in the past to describe individual dependency parsers: In Kuhlmann (2007, 2010) a grammatical deduction system was used to define a parser for regular dependency grammars.

McDonald and Nivre (2007) give an alternative framework for dependency parsers, viewing them as transition systems. That model is based on parser configurations and transitions, and has no clear relationship to the approach described here.

To demonstrate the theoretical uses of dependency parsing schemata, we have used them to describe a wide range of existing projective and non-projective dependency parsers. We have also clarified various relations between parsers which were originally formulated very differently—for example, establishing the relation between the dynamic programming algorithm of Eisner (1996) and the transition-based parser of Yamada and Matsumoto (2003). We have also used the parsing schemata framework as a formal tool to verify the correctness of parsing algorithms.

Not only are dependency parsing schemata useful when describing and extending existing parsing algorithms, they can be used to define new parsers. We have presented an algorithm that can parse any well-nested dependency structure with gap degree bounded by a constant k with time complexity $O(n^{2k+5})$, and additionally, have defined a wider set of structures that we call mildly ill-nested for a given gap degree k, and presented an algorithm that can parse these in time $O(n^{3k+4})$. The practical relevance of this set of structures can be seen in the data obtained from several dependency treebanks, showing that all the sentences contained in them are mildly ill-nested for their gap degree, and thus they are parsable with this algorithm. The strategy used by this algorithm for parsing mildly ill-nested structures has been adapted to solve the problem of finding minimal fan-out binarizations of LCFRS to improve parsing efficiency (see Gómez-Rodríguez et al. 2009).

An interesting line of future work would be to provide implementations of the mildly non-projective dependency parsers presented here, using probabilistic models to guide their linking decisions, and compare their practical performance and accuracy to those of other non-projective dependency parsers. Additionally, our definition of mildly ill-nested structures is closely related to the way the corresponding parser works. It would be interesting to find a more grammar-oriented definition that would provide linguistic insight into this set of structures.

An alternative generalization of the concept of well-nestedness has recently been introduced by Maier and Lichte (2011). The definition of this property of structures, called *k-ill-nestedness*, is more declarative than that of mildly ill-nestedness. However, it is based on properties that are not local to projections or subtrees, and there is no evidence that k-ill-nested structures are parsable in polynomial time.

Finally, we observe that that some well-known parsing algorithms discussed here (Nivre 2003; McDonald et al. 2005) rely on statistically-driven control mechanisms that fall below the abstraction level of parsing schemata. Therefore, it would be useful to have an extension of parsing schemata allowing the description and comparison of these control structures in a general way.

Acknowledgments

This work was partially supported by MEC and FEDER (HUM2007-66607-C04) and Xunta de Galicia (PGIDIT07SIN005206PR, INCITE08E1R104022ES, INCITE08ENA305025ES, INCITE08PXIB302179PR, Rede Galega de Proc. da Linguaxe e Recup. de Información, Rede Galega de Lingüística de Corpus, Bolsas Estadías INCITE/FSE cofinanced).

References

Afonso, Susana, Eckhard Bick, Renato Haber, and Diana Santos. 2002. "Floresta sintá(c)tica": A treebank for Portuguese. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC 2002)*, pages 1968–1703, Las Palmas.

Alonso, Miguel A., David Cabrero, Éric Villemonte de la Clergerie, and Manuel Vilares. 1999. Tabular algorithms for TAG

parsing. In *Proceedings of the Ninth Conference of the European Chapter of the Association for Computational Linguistics (EACL-99)*, pages 150–157, Bergen.

Atalay, Nart B., Kemal Oflazer, and Bilge Say. 2003. The annotation process in the Turkish treebank. In *Proceedings of EACL Workshop on Linguistically Interpreted Corpora (LINC-03)*, pages 243–246, Budapest.

Attardi, Giuseppe. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL-X)*, pages 166–170, New York, NY.

Bamman, David and Gregory Crane. 2006. The design and use of a Latin dependency treebank. In *Proceedings of the Fifth Workshop on Treebanks and Linguistic Theories (TLT 2006)*, pages 67–78, Prague.

Barbero, Cristina, Leonardo Lesmo, Vincenzo Lombardo, and Paola Merlo. 1998. Integration of syntactic and lexical information in a hierarchical dependency grammar. In *Proceedings of COLING-ACL '98 Workshop on Processing of Dependency-Based Grammars*, pages 58–67, Montreal.

van der Beek, Leonoor, Gosse Bouma, Robert Malouf, and Gertjan van Noord. 2002. The Alpino dependency treebank. In *Language and Computers, Computational Linguistics in the Netherlands 2001. Selected Papers from the Twelfth CLIN Meeting*, pages 8–22, Amsterdam.

Billot, Sylvie and Bernard Lang. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics (ACL'89)*, pages 143–151, Montreal.

Bodirsky, Manuel, Marco Kuhlmann, and Mathias Möhl. 2005. Well-nested drawings as models of syntactic structure (extended version). Technical report, Saarland University.

Che, Wanxiang, Zhenghua Li, Yuxuan Hu, Yongqiang Li, Bing Qin, Ting Liu, and Sheng Li. 2008. A cascaded syntactic and semantic dependency parsing system. In *Proceedings of the 12th Conference on Computational Natural Language Learning (CoNLL 2008)*, pages 238–242, Manchester.

Collins, Michael John. 1996. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL'96)*, pages 184–191, Santa Cruz, CA.

Corston-Oliver, Simon, Anthony Aue, Kevin Duh, and Eric Ringger. 2006. Multilingual dependency parsing using Bayes Point Machines. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (NAACL HLT 2006)*, pages 160–167, New York, NY.

Courtin, Jacques and Damien Genthial. 1998. Parsing with dependency relations and robust parsing. In *Proceedings of COLING-ACL '98 Workshop on Processing of Dependency-Based Grammars*, pages 88–94, Montreal.

Covington, Michael A. 1990. A dependency parser for variable-word-order languages. Technical Report AI-1990-01, University of Georgia, Athens, GA.

Covington, Michael A. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, Athens, GA.

Cui, Hang, Renxu Sun, Keya Li, Min-Yen Kan, and Tat-Seng Chua. 2005. Question answering passage retrieval using dependency relations. In *SIGIR '05: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 400–407, Salvador.

Culotta, Aron and Jeffrey Sorensen. 2004. Dependency tree kernels for relation extraction. In *ACL '04: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics*, pages 423–429, Barcelona.

Ding, Yuan and Martha Palmer. 2005. Machine translation using probabilistic synchronous dependency insertion grammars. In *ACL '05: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 541–548, Ann Arbor, MI.

Džeroski, Sašo, Tomaž Erjavec, Nina Ledinek, Petr Pajas, Zdeněk Žabokrtský, and Andreja Žele. 2006. Towards a Slovene dependency treebank. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC 2006)*, pages 1388–1391, Genoa.

Earley, Jay. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Eisner, Jason. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th International Conference on Computational*

Linguistics (COLING-96), pages 340–345, Copenhagen.
Eisner, Jason, Eric Goldlust, and Noah A. Smith. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP 2005)*, pages 281–290, Vancouver.
Eisner, Jason and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL'99)*, pages 457–464, College Park, MD.
Eisner, Jason and Giorgio Satta. 2000. A faster parsing algorithm for lexicalized tree-adjoining grammars. In *Proceedings of the 5th Workshop on Tree-Adjoining Grammars and Related Formalisms (TAG+5)*, pages 14–19, Paris.
Fundel, Katrin, Robert Küffner, and Ralf Zimmer. 2006. RelEx—Relation extraction using dependency parse trees. *Bioinformatics*, 23(3):365–371.
Gaifman, Haim. 1965. Dependency systems and phrase-structure systems. *Information and Control*, 8:304–337.
Gómez-Rodríguez, Carlos. 2009. *Parsing Schemata for Practical Text Analysis*. Ph.D. thesis, Universidade da Coruña, Spain.
Gómez-Rodríguez, Carlos, John Carroll, and David Weir. 2008. A deductive approach to dependency parsing. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL'08:HLT)*, pages 968–976, Columbus, OH.
Gómez-Rodríguez, Carlos, Marco Kuhlmann, Giorgio Satta, and David Weir. 2009. Optimal reduction of rule length in linear context-free rewriting systems. In *Proceedings of NAACL HLT 2009: the Conference of the North American Chapter of the Association for Computational Linguistics*, pages 539–547, Boulder, CO.
Gómez-Rodríguez, Carlos, Jesús Vilares, and Miguel A. Alonso. 2009. A compiler for parsing schemata. *Software: Practice and Experience*, 39(5):441–470.
Gómez-Rodríguez, Carlos, David Weir, and John Carroll. 2008. Parsing mildly non-projective dependency structures (extended version). Technical Report CSRP 600, Department of Informatics, University of Sussex.
Gómez-Rodríguez, Carlos, David Weir, and John Carroll. 2009. Parsing mildly non-projective dependency structures. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL-09)*, pages 291–299, Athens.
Hajič, Jan, Jarmila Panevová, Eva Hajičová, Jarmila Panevová, Petr Sgall, Petr Pajas, Jan Štěpánek, Jiří Havelka, and Marie Mikulová. 2006. Prague Dependency Treebank 2.0. CDROM CAT: LDC2006T01, ISBN 1-58563-370-4. Linguistic Data Consortium, University of Pennsylvania.
Hajič, Jan, Otakar Smrž, Petr Zemánek, Jan Šnaidauf, and Emanuel Beška. 2004. Prague Arabic dependency treebank: Development in data and tools. In *Proceedings of the NEMLAR International Conference on Arabic Language Resources and Tools*, pages 110–117, Cairo.
Havelka, Jiří. 2007. Beyond projectivity: Multilingual evaluation of constraints and measures on non-projective structures. In *ACL 2007: Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics*, pages 608–615, Prague.
Hays, David. 1964. Dependency theory: a formalism and some observations. *Language*, 40:511–525.
Herrera, Jesús, Anselmo Peñas, and Felisa Verdejo. 2005. Textual entailment recognition based on dependency analysis and WordNet. In J. Quiñonero-Camdela, I. Dagan, B. Magnini, and F. d'Alché-Buc, editors, *Machine Learning Challenges.* Lecture Notes in Computer Science, vol. 3944. Springer-Verlag, Berlin-Heidelberg-New York, pages 231–239.
Hotz, Günter and Gisela Pitsch. 1996. On parsing coupled-context-free languages. *Theoretical Computer Science*, 161(1-2):205–233.
Joshi, Aravind K. and Yves Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, vol. 3: Beyond Words*, Springer-Verlag, New York, NY, pages 69–123.
Kahane, Sylvain, Alexis Nasr, and Owen Rambow. 1998. Pseudo-projectivity: A polynomially parsable non-projective dependency grammar. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and the 17th International Conference on Computational Linguistics (COLING-ACL'98)*, pages 646–652, San Francisco, CA.

Kasami, Tadao. 1965. An efficient recognition and syntax algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA.

Kromann, Matthias T. 2003. The Danish dependency treebank and the underlying linguistic theory. In *Proceedings of the 2nd Workshop on Treebanks and Linguistic Theories (TLT)*, pages 217–220, Växjö.

Kuhlmann, Marco. 2007. *Dependency Structures and Lexicalized Grammars*. D. Phil dissertation, Saarland University, Saarbrücken, Germany.

Kuhlmann, Marco. 2010. *Dependency Structures and Lexicalized Grammars: An Algebraic Approach*. Lecture Notes in Computer Science, vol. 6270. Springer, New York, NY.

Kuhlmann, Marco and Mathias Möhl. 2007. Mildly context-sensitive dependency languages. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL 2007)*, pages 160–167, Prague.

Kuhlmann, Marco and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 507–514, Montreal.

Lombardo, Vincenzo and Leonardo Lesmo. 1996. An Earley-type recognizer for dependency grammar. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING 96)*, pages 723–728, San Francisco, CA.

Maier, Wolfgang and Timm Lichte. 2011. Characterizing discontinuity in constituent treebanks. In P. de Grook, M. Egg, and L. Kallmeyer, editors, *Formal Grammar,* volume 5591 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin-Heidelberg-New York, pages 164–179.

McDonald, Ryan, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *ACL '05: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 91–98, Ann Arbor, MI.

McDonald, Ryan and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007)*, pages 122–131, Prague.

McDonald, Ryan, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *HLT/EMNLP 2005: Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 523–530, Vancouver.

McDonald, Ryan and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In *IWPT 2007: Proceedings of the 10th International Conference on Parsing Technologies*, pages 121–132, Prague.

Nilsson, Jens, Johan Hall, and Joakim Nivre. 2005. MAMBA meets TIGER: Reconstructing a Swedish treebank from antiquity. In *Proceedings of NODALIDA 2005 Special Session on Treebanks*, pages 119–132, Joensuu.

Nivre, Joakim. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 149–160, Nancy.

Nivre, Joakim. 2007. Incremental non-projective dependency parsing. In *Proceedings of NAACL HLT 2007: The Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 396–403, Rochester, NY.

Nivre, Joakim, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932, Prague.

Nivre, Joakim, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL-2004)*, pages 49–56, Boston, MA.

Nivre, Joakim, Johan Hall, Jens Nilsson, Atanas Chanev, Gülşen Eryiğit, Sandra Kübler, Stetoslav Marinov, and Erwin Marsi. 2007. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):99–135.

Nivre, Joakim, Johan Hall, Jens Nilsson, Gülşen Eryiğit, and Stetoslav Marinov. 2006. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL-X)*, pages 221–225, Sydney.

Nivre, Joakim and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-08: HLT)*, pages 950–958, Columbus, OH.

Nivre, Joakim and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *ACL '05: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 99–106, Ann Arbor, MI.

Oflazer, Kemal, Bilge Say, Dilek Zeynep Hakkani-Tür, and Gökhan Tür. 2003. Building a Turkish treebank. In A. Abeille, editor, *Building and Exploiting Syntactically-annotated Corpora*. Kluwer, Dordrecht, pages 261–277.

Satta, Giorgio. 1992. Recognition of linear context-free rewriting systems. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL'92)*, pages 89–95, Newark, DE.

Schneider, Gerold. 1998. A linguistic comparison of constituency, dependency, and link grammar. M.Sc. thesis, University of Zurich, Switzerland.

Shen, Libin, Jinxi Xu, and Ralph Weischedel. 2008. A new string-to-dependency machine translation algorithm with a target dependency language model. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-08: HLT)*, pages 577–585, Columbus, OH.

Shieber, Stuart M., Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24:3–36.

Sikkel, Klaas. 1994. How to compare the structure of parsing algorithms. In *Proceedings of ASMICS Workshop on Parsing Theory*, pages 21–39, Milano.

Sikkel, Klaas. 1997. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science — An EATCS Series. Springer-Verlag, Berlin-Heidelberg-New York.

Sleator, Daniel and Davy Temperley. 1991. Parsing English with a Link Grammar. Technical report CMU-CS-91-196, Carnegie Mellon University, Pittsburgh, PA.

Sleator, Daniel and Davy Temperley. 1993. Parsing English with a Link Grammar. In *Proceedings of the Third International Workshop on Parsing Technologies (IWPT'93)*, pages 277–292, Tilburg.

Surdeanu, Mihai, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. 2008. The CoNLL-2008 shared task on joint parsing of syntactic and semantic dependencies. In *Proceedings of the 12th Conference on Computational Natural Language Learning (CoNLL-2008)*, pages 159–177, Manchester.

Vijay-Shanker, K., David J. Weir, and Aravind K. Joshi. 1987. Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics (ACL'87)*, pages 104–111, Stanford, CA.

de Vreught, J. P. M. and H. J. Honig. 1989. A tabular bottom–up recognizer. Report 89-78, Delft University of Technology, Delft, the Netherlands.

Yamada, Hiroyasu and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of 8th International Workshop on Parsing Technologies (IWPT 2003)*, pages 195–206, Nancy.

Younger, Daniel H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.