Mildly Non-Projective Dependency Grammar

Marco Kuhlmann*
Uppsala University

Syntactic representations based on word-to-word dependencies have a long-standing tradition in descriptive linguistics, and receive considerable interest in many applications. Nevertheless, dependency syntax has remained something of an island from a formal point of view. Moreover, most formalisms available for dependency grammar are restricted to projective analyses, and thus not able to support natural accounts of phenomena such as wh-movement and cross–serial dependencies. In this article we present a formalism for non-projective dependency grammar in the framework of linear context-free rewriting systems. A characteristic property of our formalism is a close correspondence between the non-projectivity of the dependency trees admitted by a grammar on the one hand, and the parsing complexity of the grammar on the other. We show that parsing with unrestricted grammars is intractable. We therefore study two constraints on non-projectivity, block-degree and well-nestedness. Jointly, these two constraints define a class of "mildly" non-projective dependency grammars that can be parsed in polynomial time. An evaluation on five dependency treebanks shows that these grammars have a good coverage of empirical data.

1. Introduction

Syntactic representations based on word-to-word dependencies have a long-standing tradition in descriptive linguistics. Since the seminal work of Tesnière (1959), they have become the basis for several linguistic theories, such as Functional Generative Description (Sgall, Hajičová, and Panevová 1986), Meaning–Text Theory (Mel'čuk 1988), and Word Grammar (Hudson 2007). In recent years they have also been used for a wide range of practical applications, such as information extraction, machine translation, and question answering. We ascribe the widespread interest in dependency structures to their intuitive appeal, their conceptual simplicity, and in particular to the availability of accurate and efficient dependency parsers for a wide range of languages (Buchholz and Marsi 2006; Nivre et al. 2007).

Although there exist both a considerable practical interest and an extensive linguistic literature, dependency syntax has remained something of an island from a formal point of view. In particular, there are relatively few results that bridge between dependency syntax and other traditions, such as phrase structure or categorial syntax.

* Department of Linguistics and Philology, Box 635, 751 26 Uppsala, Sweden.
E-mail: marco.kuhlmann@lingfil.uu.se.

Submission received: 17 December 2009; revised submission received: 3 April 2012; accepted for publication: 24 May 2012.

doi:10.1162/COLI_a_00125

© 2013 Association for Computational Linguistics

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

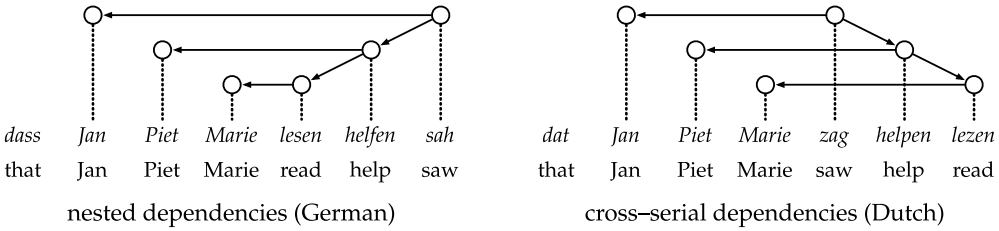


Figure 1
Nested dependencies and cross–serial dependencies.

This makes it hard to gauge the similarities and differences between the paradigms, and hampers the exchange of linguistic resources and computational methods. An overarching goal of this article is to bring dependency grammar closer to the mainland of formal study.

One of the few bridging results for dependency grammar is thanks to Gaifman (1965), who studied a formalism that we will refer to as Hays–Gaifman grammar, and proved it to be weakly equivalent to context-free phrase structure grammar. Although this result is of fundamental importance from a theoretical point of view, its practical usefulness is limited. In particular, Hays–Gaifman grammar is restricted to projective dependency structures, which is similar to the familiar restriction to contiguous constituents. Yet, *non*-projective dependencies naturally arise in the analysis of natural language. One classic example of this is the phenomenon of cross–serial dependencies in Dutch. In this language, the nominal arguments of verbs that also select an infinitival complement occur in the same order as the verbs themselves:

(i) dat Jan_1 Piet_2 Marie_3 zag_1 helpen_2 lezen_3 (Dutch)
that Jan Piet Marie saw help read

'that Jan saw Piet help Marie read'

In German, the order of the nominal arguments instead inverts the verb order:

(ii) dass Jan_1 Piet_2 Marie_3 lesen_3 helfen_2 sah_1 (German)
that Jan Piet Marie read help saw

Figure 1 shows dependency trees for the two examples.[1] The German linearization gives rise to a projective structure, where the verb–argument dependencies are nested within each other, whereas the Dutch linearization induces a non-projective structure with crossing edges. To account for such structures we need to turn to formalisms more expressive than Hays–Gaifman grammars.

In this article we present a formalism for non-projective dependency grammar based on linear context-free rewriting systems (LCFRSs) (Vijay-Shanker, Weir, and Joshi 1987; Weir 1988). This framework was introduced to facilitate the comparison of various

1 We draw the nodes of a dependency tree as circles, and the edges as arrows pointing towards the dependent (away from the root node). Following Hays (1964), we use dotted lines to help us keep track of the positions of the nodes in the linear order, and to associate nodes with lexical items.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

grammar formalisms, including standard context-free grammar, tree-adjoining grammar (Joshi and Schabes 1997), and combinatory categorial grammar (Steedman and Baldridge 2011). It also comprises, among others, multiple context-free grammars (Seki et al. 1991), minimalist grammars (Michaelis 1998), and simple range concatenation grammars (Boullier 2004).

The article is structured as follows. In Section 2 we provide the technical background to our work; in particular, we introduce our terminology and notation for linear context-free rewriting systems. An LCFRS generates a set of terms (formal expressions) which are interpreted as derivation trees of objects from some domain. Each term also has a secondary interpretation under which it denotes a tuple of strings, representing the string yield of the derived object. In Section 3 we introduce the central notion of a lexicalized linear context-free rewriting system, which is an LCFRS in which each rule of the grammar is associated with an overt lexical item, representing a syntactic head (cf. Schabes, Abeillé, and Joshi 1988 and Schabes 1990). We show that this property gives rise to an additional interpretation under which each term denotes a dependency tree on its yield. With this interpretation, lexicalized LCFRSs can be used as dependency grammars.

In Section 4 we show how to acquire lexicalized LCFRSs from dependency treebanks. This works in much the same way as the extraction of context-free grammars from phrase structure treebanks (cf. Charniak 1996), except that the derivation trees of dependency trees are not immediately accessible in the treebank. We therefore present an efficient algorithm for computing a canonical derivation tree for an input dependency tree; from this derivation tree, the rules of the grammar can be extracted in a straightforward way. The algorithm was originally published by Kuhlmann and Satta (2009). It produces a restricted type of lexicalized LCFRS that we call "canonical." In Section 5 we provide a declarative characterization of this class of grammars, and show that every lexicalized LCFRS is (strongly) equivalent to a canonical one, in the sense that it induces the same set of dependency trees.

In Section 6 we present a simple parsing algorithm for LCFRSs. Although the runtime of this algorithm is polynomial in the length of the sentence, the degree of the polynomial depends on two grammar-specific measures called fan-out and rank. We show that even in the restricted case of canonical grammars, parsing is an NP-hard problem. It is important therefore to keep the fan-out and the rank of a grammar as low as possible, and much of the recent work on LCFRSs has been devoted to the development of techniques that optimize parsing complexity in various scenarios Gómez-Rodríguez and Satta 2009; Gómez-Rodríguez et al. 2009; Kuhlmann and Satta 2009; Gildea 2010; Gómez-Rodríguez, Kuhlmann, and Satta 2010; Sagot and Satta 2010; and Crescenzi et al. 2011).

In this article we explore the impact of non-projectivity on parsing complexity. In Section 7 we present the structural correspondent of the fan-out of a lexicalized LCFRS, a measure called **block-degree** (or gap-degree) (Holan et al. 1998). Although there is no theoretical upper bound on the block-degree of the dependency trees needed for linguistic analysis, we provide evidence from several dependency treebanks showing that, from a practical point of view, this upper bound can be put at a value of as low as 2. In Section 8 we study a second constraint on non-projectivity called **well-nestedness** (Bodirsky, Kuhlmann, and Möhl 2005), and show that its presence facilitates tractable parsing. This comes at the cost of a small loss in coverage on treebank data. Bounded block-degree and well-nestedness jointly define a class of "mildly" non-projective dependency grammars that can be parsed in polynomial time.

Section 9 summarizes our main contributions and concludes the article.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

2. Technical Background

We assume basic familiarity with linear context-free rewriting systems (see, e.g., Vijay-Shanker, Weir, and Joshi 1987 and Weir 1988) and only review the terminology and notation that we use in this article.

A **linear context-free rewriting system** (LCFRS) is a structure $G = (N, \Sigma, P, S)$ where N is a set of nonterminals, Σ is a set of function symbols, P is a finite set of production rules, and $S \in N$ is a distinguished start symbol. Rules take the form

$$A_0 \to f(A_1, \ldots, A_m) \tag{1}$$

where f is a function symbol and the A_i are nonterminals. Rules are used for rewriting in the same way as in a context-free grammar, with the function symbols acting as terminals. The outcome of the rewriting process is a set $T(G)$ of terms, tree-formed expressions built from function symbols. Each term is then associated with a string yield, more specifically a *tuple* of strings. For this, every function symbol f comes with a **yield function** that specifies how to compute the yield of a term $f(t_1, \ldots, t_m)$ from the yields of its subterms t_i. Yield functions are defined by equations

$$f(\langle x_{1,1}, \ldots, x_{1,k_1}\rangle, \ldots, \langle x_{m,1}, \ldots, x_{m,k_m}\rangle) = \langle \alpha_1, \ldots, \alpha_{k_0}\rangle \tag{2}$$

where the tuple on the right-hand side consists of strings over the variables on the left-hand side and some given alphabet of yield symbols, and contains exactly one occurrence of each variable. For a yield function f defined by an equation of this form, we say that f is of **type** $k_1 \cdots k_m \to k_0$, denoted by $f : k_1 \cdots k_m \to k_0$. To guarantee that the string yield of a term is well-defined, each nonterminal A is associated with a **fan-out** $\varphi(A) \geq 1$, and it is required that for every rule (1),

$$f : \varphi(A_1) \cdots \varphi(A_m) \to \varphi(A_0)$$

In Equation (2), the values m and k_0 are called the **rank** and the **fan-out** of f, respectively. The rank and the fan-out of an LCFRS are the maximal rank and fan-out of its yield functions.

Example 1
Figure 2 shows an example of an LCFRS for the language $\{ \langle a^n b^n c^n d^n\rangle \mid n \geq 0 \}$.

Equation (2) is uniquely determined by the tuple on the right-hand side of the equation. We call this tuple the **template** of the yield function f, and use it as the canonical function symbol for f. This gives rise to a compact notation for LCFRSs,

Rules	*Equations*	*Compact Notation*
$\mathrm{S} \to f_1(\mathrm{R})$	$f_1(\langle x_{1,1}, x_{1,2}\rangle) = \langle x_{1,1}\, x_{1,2}\rangle$	$\mathrm{S} \to \langle x_1\, x_2\rangle(\mathrm{R})$
$\mathrm{R} \to f_2()$	$f_2() = \langle \varepsilon, \varepsilon\rangle$	$\mathrm{R} \to \langle \varepsilon, \varepsilon\rangle$
$\mathrm{R} \to f_3(\mathrm{R})$	$f_3(\langle x_{1,1}, x_{1,2}\rangle) = \langle a\, x_{1,1}\, b, c\, x_{1,2}\, d\rangle$	$\mathrm{R} \to \langle a\, x_1\, b, c\, x_2\, d\rangle(\mathrm{R})$

Figure 2
An LCFRS that generates the yield language $\{ \langle a^n b^n c^n d^n\rangle \mid n \geq 0 \}$.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

illustrated in the right column of Figure 2. In this notation, to save some subscripts, we use the following shorthands for variables: x and x_1 for $x_{1,1}$; x_2 for $x_{1,2}$; x_3 for $x_{1,3}$; y and y_1 for $x_{2,1}$; y_2 for $x_{2,2}$; y_3 for $x_{2,3}$.

3. Lexicalized LCFRSs as Dependency Grammars

Recall the following examples for verb–argument dependencies in German and Dutch from Section 1:

(iii)	dass	Jan_1	Piet_2	Marie_3	lesen_3	helfen_2	sah_1	(German)
	that	Jan	Piet	Marie	read	help	saw	
(iv)	dat	Jan_1	Piet_2	Marie_3	zag_1	helpen_2	lezen_3	(Dutch)
	that	Jan	Piet	Marie	saw	help	read	

'that Jan saw Piet help Marie read'

Figure 3 shows the production rules of two linear context-free rewriting systems (one for German, one for Dutch) that generate these examples. The grammars are **lexicalized** in the sense that each of their yield functions is associated with a lexical item, such as *sah* or *zag* (cf. Schabes, Abeillé, and Joshi 1988 and Schabes 1990). Productions with lexicalized yield functions can be read as dependency rules. For example, the rules

$$\text{V} \to \langle x\ y\ \mathit{sah} \rangle(\text{N}, \text{V}) \quad \text{(German)} \qquad \text{V} \to \langle x\, y_1\ \mathit{zag}\, y_2 \rangle(\text{N}, \text{V}) \quad \text{(Dutch)}$$

can be read as stating that the verb *to see* requires two dependents, one noun (N) and one verb (V). Based on this reading, every term generated by a lexicalized LCFRS does not only yield a tuple of strings, but also induces a dependency tree on these strings: Each parent–child relation in the term represents a dependency between the associated lexical items (cf. Rambow and Joshi 1997). Thus every lexicalized LCFRS can be reinterpreted as a dependency grammar. To illustrate the idea, Figure 4 shows (the tree representations of) two terms generated by the grammars G_1 and G_2, together with the dependency trees induced by them. Note that these are the same trees that we gave for (iii) and (iv) in Figure 1.

Our goal for the remainder of this section is to make the notion of induction formally precise. To this end we will reinterpret the yield functions of lexicalized LCFRSs as operations on dependency trees.

$\text{S} \to \langle x\ y\ \mathit{sah} \rangle(\text{N}, \text{V})$	$\text{S} \to \langle x\, y_1\ \mathit{zag}\, y_2 \rangle(\text{N}, \text{V})$
$\text{V} \to \langle x\ y\ \mathit{helfen} \rangle(\text{N}, \text{V})$	$\text{V} \to \langle x\, y_1, \mathit{helpen}\, y_2 \rangle(\text{N}, \text{V})$
$\text{V} \to \langle x\ \mathit{lesen} \rangle(\text{N})$	$\text{V} \to \langle x, \mathit{lezen} \rangle(\text{N})$
$\text{N} \to \langle \mathit{Jan} \rangle$	$\text{N} \to \langle \mathit{Jan} \rangle$
$\text{N} \to \langle \mathit{Piet} \rangle$	$\text{N} \to \langle \mathit{Piet} \rangle$
$\text{N} \to \langle \mathit{Marie} \rangle$	$\text{N} \to \langle \mathit{Marie} \rangle$
G_1 (German)	G_2 (Dutch)

Figure 3
Lexicalized linear context-free rewriting systems.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

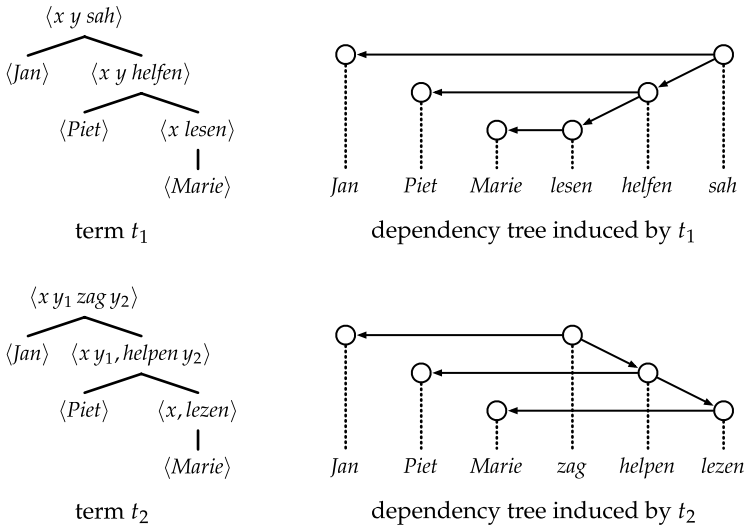


Figure 4
Lexicalized linear context-free rewriting systems induce dependency trees.

3.1 Dependency Trees

By a dependency tree, we mean a pair $(\vec{w}, D)$, where $\vec{w}$ is a tuple of strings, and D is a tree-shaped graph whose nodes correspond to the occurrences of symbols in $\vec{w}$, and whose edges represent dependency relations between these occurrences. We identify occurrences in $\vec{w}$ by pairs (i, j) of integers, where i indexes the component of $\vec{w}$ that contains the occurrence, and j specifies the linear position of the occurrence within that component. We can then formally define a **dependency graph** for a tuple of strings

$$\vec{w} = \langle a_{1,1} \cdots a_{1,n_1}, \ldots, a_{k,1} \cdots a_{k,n_k} \rangle$$

as a directed graph $G = (V, E)$ where

$$V = \{ (i, j) \mid 1 \leq i \leq k,\ 1 \leq j \leq n_i \} \qquad \text{and} \qquad E \subseteq V \times V$$

We use u and v as variables for nodes, and denote edges (u, v) as $u \to v$. A **dependency tree** D for $\vec{w}$ is a dependency graph for $\vec{w}$ in which there exists a root node r such that for any node u, there is exactly one directed path from r to u. A dependency tree is called **simple** if $\vec{w}$ consists of a single string w. In this case, we write the dependency tree as (w, D), and identify occurrences by their linear positions j in w, with $1 \leq j \leq |w|$.

Example 2
Figure 5 shows examples of dependency trees. In pictures of such structures we use dashed boxes to group nodes that correspond to occurrences from the same tuple

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

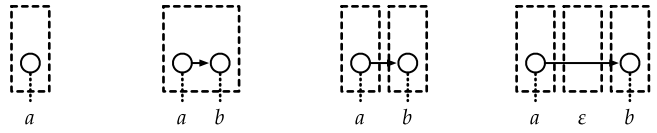


Figure 5
Dependency trees.

component; however, we usually omit the box when there is only one component. Writing D_i as $D_i = (V_i, E_i)$ we have:

$$\begin{aligned} V_1 &= \{(1,1)\} & E_1 &= \{\} \\ V_2 &= \{(1,1),(1,2)\} & E_2 &= \{(1,1) \to (1,2)\} \\ V_3 &= \{(1,1),(2,1)\} & E_3 &= \{(1,1) \to (2,1)\} \\ V_4 &= \{(1,1),(3,1)\} & E_4 &= \{(1,1) \to (3,1)\} \end{aligned}$$

We use standard terminology from graph theory for dependency trees and the relations between their nodes. In particular, for a node u, the set of **descendants** of u, which we denote by $\lfloor u \rfloor$, is the set of nodes that can be reached from u by following a directed path consisting of zero or more edges. We write $u < v$ to express that the node u precedes the node v when reading the yield from left to right. Formally, precedence is the lexicographical order on occurrences:

$$(i_1, j_1) < (i_2, j_2) \qquad \text{if and only if} \qquad \text{either } i_1 < i_2 \text{ or } (i_1 = i_2 \text{ and } j_1 < j_2)$$

3.2 Operations on Dependency Trees

A yield function f is called **lexicalized** if its template contains exactly one yield symbol, representing a lexical item; this symbol is then called the **anchor** of f. With every lexicalized yield function f we associate an operation f' on dependency trees as follows. Let $\vec{w}_1, \ldots, \vec{w}_m, \vec{w}$ be tuples of strings such that

$$f(\vec{w}_1, \ldots, \vec{w}_m) = \vec{w}$$

and let D_i be a dependency tree for $\vec{w}_i$. By the definition of yield functions, every occurrence u in an input tuple $\vec{w}_i$ corresponds to exactly one occurrence in the output tuple $\vec{w}$; we denote this occurrence by $\bar{u}$. Let G be the dependency graph for $\vec{w}$ that has an edge $\bar{u} \to \bar{v}$ whenever there is an edge $u \to v$ in some D_i, and no other edges. Because f is lexicalized, there is exactly one occurrence r in the output tuple $\vec{w}$ that does not correspond to any occurrence in some $\vec{w}_i$; this is the occurrence of the anchor of f. Let D be the dependency tree for $\vec{w}$ that is obtained by adding to the graph G all edges of the form $r \to \bar{r}_i$, where r_i is the root node of D_i. By this construction, the occurrence r of the anchor becomes the root node of D, and the root nodes of the input dependency trees D_i become its dependents. We then define

$$f'((\vec{w}_1, D_1), \ldots, (\vec{w}_m, D_m)) = (\vec{w}, D)$$

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

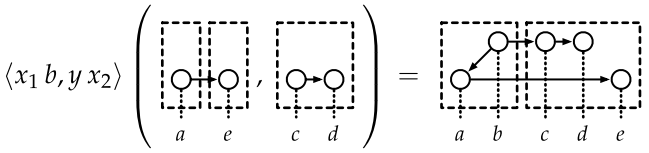


Figure 6
Operations on dependency trees.

Example 3
We consider a concrete application of an operation on dependency trees, illustrated in Figure 6. In this example we have

$$f = \langle x_1\, b, y\, x_2\rangle \qquad \vec{w}_1 = \langle a, e\rangle \qquad \vec{w}_2 = \langle c\, d\rangle \qquad \vec{w} = f(\vec{w}_1, \vec{w}_2) = \langle a\, b, c\, d\, e\rangle$$

and the dependency trees D_1, D_2 are defined as

$$D_1 = (\{(1,1),(2,1)\}, \{(1,1) \to (2,1)\}) \quad D_2 = (\{(1,1),(1,2)\}, \{(1,1) \to (1,2)\})$$

We show that $f'((\vec{w}_1, D_1), (\vec{w}_2, D_2)) = (\vec{w}, D)$, where $D = (V, E)$ with

$$V = \{(1,1),(1,2),(2,1),(2,2),(2,3)\}$$
$$E = \{(1,1) \to (2,3),(1,2) \to (1,1),(1,2) \to (2,1),(2,1) \to (2,2)\}$$

The correspondences between the occurrences u in the input tuples and the occurrences $\bar{u}$ in the output tuple are as follows:

$$\text{for } \vec{w}_1\text{:}\quad \overline{(1,1)} = (1,1)\,,\ \overline{(2,1)} = (2,3) \qquad \text{for } \vec{w}_2\text{:}\quad \overline{(1,1)} = (2,1)\,,\ \overline{(1,2)} = (2,2)$$

By copying the edges from the input dependency trees, we obtain the intermediate dependency graph $G = (V, E')$ for $\vec{w}$, where

$$E' = \{(1,1) \to (2,3),(2,1) \to (2,2)\}$$

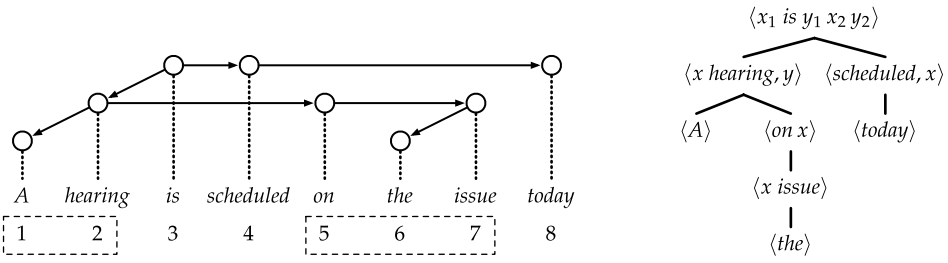
The occurrence r of the anchor b of f in $\vec{w}$ is $(1,2)$; the nodes of G that correspond to the root nodes of D_1 and D_2 are $\bar{r}_1 = (1,1)$ and $\bar{r}_2 = (2,1)$. The dependency tree D is obtained by adding the edges $r \to \bar{r}_1$ and $r \to \bar{r}_2$ to G.

4. Extraction of Dependency Grammars

We now show how to extract lexicalized linear context-free rewriting systems from dependency treebanks. To this end, we adapt the standard technique for extracting context-free grammars from phrase structure treebanks (Charniak 1996).

Our technique was originally published by Kuhlmann and Satta (2009). In recent work, Maier and Lichte (2011) have shown how to unify it with a similar technique for the extraction of range concatenation grammars from discontinuous constituent structures, due to Maier and Søgaard (2008). To simplify our presentation we restrict our attention to treebanks containing simple dependency trees.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024



dependency tree (w, D) construction tree t

Figure 7
A dependency tree and one of its construction trees.

To extract a lexicalized LCFRS from a dependency treebank we proceed as follows. First, for each dependency tree (w, D) in the treebank, we compute a **construction tree**, a term t over yield functions that induces (w, D). Then we collect a set of production rules, one rule for each node of the construction trees. As an example, consider Figure 7, which shows a dependency tree with one of its construction trees. (The analysis is taken from Kübler, McDonald, and Nivre [2009].) From this construction tree we extract the following rules. The nonterminals (in bold) represent linear positions of nodes.

$$\mathbf{1} \to \langle A \rangle \qquad \mathbf{5} \to \langle \mathit{on}\ x \rangle(\mathbf{7})$$
$$\mathbf{2} \to \langle x\ \mathit{hearing}, y \rangle(\mathbf{1}, \mathbf{5}) \qquad \mathbf{6} \to \langle \mathit{the} \rangle$$
$$\mathbf{3} \to \langle x_1\ \mathit{is}\ y_1\ x_2\ y_2 \rangle(\mathbf{2}, \mathbf{4}) \qquad \mathbf{7} \to \langle x\ \mathit{issue} \rangle(\mathbf{6})$$
$$\mathbf{4} \to \langle \mathit{scheduled}, x \rangle(\mathbf{8}) \qquad \mathbf{8} \to \langle \mathit{today} \rangle$$

Rules like these can serve as the starting point for practical systems for data-driven, non-projective dependency parsing (Maier and Kallmeyer 2010).

Because the extraction of rules from construction trees is straightforward, the problem that we focus on in this section is how to obtain these trees in the first place. Our procedure for computing construction trees is based on the concept of "blocks."

4.1 Blocks

Let D be a dependency tree. A **segment** of D is a contiguous, non-empty sequence of nodes of D, all of which belong to the same component of the string yield. Thus a segment contains its endpoints, as well as all nodes between the endpoints in the precedence order. For a node u of D, a **block** of u is a longest segment consisting of descendants of u. This means that the left endpoint of a block of u either is the first node in its component, or is preceded by a node that is not a descendant of u. A symmetric property holds for the right endpoint.

Example 4
Consider the node 2 of the dependency tree in Figure 7. The descendants of 2 fall into two blocks, marked by the dashed boxes: 1 2 and 5 6 7.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

We use $\vec{u}$ and $\vec{v}$ as variables for blocks. Extending the precedence order on nodes, we say that a block $\vec{u}$ precedes a block $\vec{v}$, denoted by $\vec{u} < \vec{v}$, if the right endpoint of $\vec{u}$ precedes the left endpoint of $\vec{v}$.

4.2 Computing Canonical Construction Trees

To obtain a canonical construction tree t for a dependency tree (w, D) we label each node u of D with a yield function f as follows. Let $\vec{w}$ be the tuple consisting of the blocks of u, in the order of their precedence, and let $\vec{w}_1, \ldots, \vec{w}_m$ be the corresponding tuples for the children of u. We may view blocks as strings of nodes. Taking this view, we compute the (unique) yield function g with the property that

$$g(\vec{w}_1, \ldots, \vec{w}_m) = \vec{w}$$

The anchor of g is the node u, the rank of g corresponds to the number of children of u, the variables in the template of g represent the blocks of these children, and the components of the template represent the blocks of u. To obtain f, we take the template of g and replace the occurrence of u with the corresponding lexical item.

Example 5
Node 2 of the dependency tree shown in Figure 7 has two children, 1 and 5. We have

$$\vec{w} = \langle 1\,2, 5\,6\,7 \rangle \qquad \vec{w}_1 = \langle 1 \rangle \qquad \vec{w}_2 = \langle 5\,6\,7 \rangle \qquad g = \langle x\,2, y \rangle \qquad f = \langle x\,\mathit{hearing}, y \rangle$$

Note that in order to properly define f we need to assume some order on the children of u. The function g (and hence the construction tree t) is unique up to the specific choice of this order. In the following we assume that children are ordered from left to right based on the position of their leftmost descendants.

4.3 Computing the Blocks of a Dependency Tree

The algorithmically most interesting part of our extraction procedure is the computation of the yield function g. The template of g is uniquely determined by the left-to-right sequence of the endpoints of the blocks of u and its children. An efficient algorithm that can be used to compute these sequences is given in Table 1.

4.3.1 Description. We start at a virtual root node $\perp$ (line 1) which serves as the parent of the real root node. For each node *next* in the precedence order of D, we follow the shortest path from the current node *current* to *next*. To determine this path, we compute the lowest common ancestor *lca* of the two nodes (lines 4–5), using a set of markings on the nodes. At the beginning of each iteration of the *for* loop in line 2, all ancestors of *current* (including the virtual root node $\perp$) are marked; therefore, we find *lca* by going upwards from *next* to the first node that is marked. To restore the loop invariant, we then unmark all nodes on the path from *current* to *lca* (lines 6–9). Each time we move down from a node to one of its children (line 12), we record the information that *next* is the left endpoint of a block of *current*. Symmetrically, each time we move up from a node to its parent (lines 8 and 17), we record the information that *next* $-$ 1 is the right endpoint of a block of *current*. The *while* loop in lines 15–18 takes us from the last node of the dependency tree back to the node $\perp$.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Table 1
Computing the blocks of a simple dependency tree.

Input: a string w and a simple dependency tree D for w

```
 1:  current ← ⊥; mark current
 2:  for each node next of D from 1 to |w| do
 3:      lca ← next; stack ← []
 4:      while lca is not marked do                          loop 1
 5:          push lca to stack; lca ← the parent of lca
 6:      while current ≠ lca do                              loop 2
 7:          ▷ next − 1 is the right endpoint of a block of current
 8:          ▷ move up from current to the parent of current
 9:          unmark current; current ← the parent of current
10:      while stack is not empty do                         loop 3
11:          current ← pop stack; mark current
12:          ▷ move down from the parent of current to current
13:          ▷ next is the left endpoint of a block of current
14:      ▷ arrive at next; at this point, current = next
15:  while current ≠ ⊥ do                                    loop 4
16:      ▷ |w| is the right endpoint of a block of current
17:      ▷ move up from current to the parent of current
18:      unmark current; current ← the parent of current
```

4.3.2 Runtime Analysis. We analyze the runtime of our algorithm. Let m be the total number of blocks of D. Let us write n_i for the total number of iterations of the ith *while* loop, and let $n = n_1 + n_2 + n_3 + n_4$. Under the reasonable assumption that every line in Table 1 can be executed in constant time, the runtime of the algorithm clearly is in $O(n)$. Because each iteration of loop 2 and loop 4 determines the right endpoint of a block, we have $n_2 + n_4 = m$. Similarly, as each iteration of loop 3 fixes the left endpoint of a block, we have $n_3 = m$. To determine n_1, we note that every node that is pushed to the auxiliary stack in loop 1 is popped again in loop 3; therefore, $n_1 = n_3 = m$. Putting everything together, we have $n = 3m$, and we conclude that the runtime of the algorithm is in $O(m)$. Note that this runtime is asymptotically optimal for the task we are considering.

5. Canonical Grammars

Our extraction technique produces a restricted type of lexicalized linear context-free rewriting system that we will refer to as "canonical." In this section we provide a declarative characterization of these grammars, and show that every lexicalized LCFRS is equivalent to a canonical one.

5.1 Definition of Canonical Grammars

We are interested in a syntactic characterization of the yield functions that can occur in extracted grammars. We give such a characterization in terms of four properties, stated in the following. We use the following terminology and notation. Consider a yield function

$$f : k_1 \cdots k_m \to k\,, \qquad f = \langle \alpha_1, \ldots, \alpha_k \rangle$$

For variables x, y we write $x <_f y$ to state that x precedes y in the template of f, that is, in the string $\alpha_1 \cdots \alpha_k$. Recall that, in the context of our extraction procedure, the

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

components in the template of f represent the blocks of a node u, and the variables in the template represent the blocks of the children of u. For a variable $x_{i,j}$ we call i the **argument index** and j the **component index** of the variable.

Property 1
For all $1 \leq i_1, i_2 \leq m$, if $i_1 < i_2$ then $x_{i_1,1} <_f x_{i_2,1}$.

This property is an artifact of our decision to order the children of a node from left to right based on the position of their leftmost descendants. A variable with argument index i represents a block of the ith child of u in that order. An example of a yield function that does not have Property 1 is $\langle x_{2,1}\, x_{1,1} \rangle$, which defines a kind of "reverse concatenation operation."

Property 2
For all $1 \leq i \leq m$ and $1 \leq j_1, j_2 \leq k_i$, if $j_1 < j_2$ then $x_{i,j_1} <_f x_{i,j_2}$.

This property reflects that, in our extraction procedure, the variable $x_{i,j}$ represents the jth block of the ith child of u, where the blocks of a node are ordered from left to right based on their precedence. An example of a yield function that violates the property is $\langle x_{1,2}\, x_{1,1} \rangle$, which defines a kind of **swapping operation**. In the literature on LCFRSs and related formalisms, yield functions with Property 2 have been called **monotone** (Michaelis 2001; Kracht 2003), **ordered** (Villemonte de la Clergerie 2002; Kallmeyer 2010), and **non-permuting** (Kanazawa 2009).

Property 3
No component α_h is the empty string.

This property, which is similar to ε-freeness as known from context-free grammars, has been discussed for multiple context-free grammars (Seki et al. 1991, Property N3 in Lemma 2.2) and range concatenation grammars (Boullier 1998, Section 5.1). For our extracted grammars it holds because each component α_h represents a block, and blocks are always non-empty.

Property 4
No component α_h contains a substring of the form $x_{i,j_1} x_{i,j_2}$.

This property, which does not seem to have been discussed in the literature before, is a reflection of the facts that variables with the same argument index represent blocks of the same child node, and that these blocks are *longest* segments of descendants.

A yield function with Properties 1–4 is called **canonical**. An LCFRS is canonical if all of its yield functions are canonical.

Lemma 1
A lexicalized LCFRS is canonical if and only if it can be extracted from a dependency treebank using the technique presented in Section 4.

Proof
We have already argued for the "only if" part of the claim. To prove the "if" part, it suffices to show that for every canonical, lexicalized yield function f, one can construct

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

a dependency tree such that the construction tree extracted for this dependency tree contains f. This is an easy exercise. ■

We conclude by noting that Properties 2–4 are also shared by the treebank grammars extracted from constituency treebanks using the technique by Maier and Søgaard (2008).

5.2 Equivalence Between General and Canonical Grammars

Two lexicalized LCFRSs are called **strongly equivalent** if they induce the same set of dependency trees. We show the following equivalence result:

Lemma 2
For every lexicalized LCFRS G one can construct a strongly equivalent lexicalized LCFRS G' such that G' is canonical.

Proof
Our proof of this lemma uses two normal-form results about multiple context-free grammars: Michaelis (2001, Section 2.4) provides a construction that transforms a multiple context-free grammar into a weakly equivalent multiple context-free grammar in which all rules satisfy Property 2, and Seki et al. (1991, Lemma 2.2) present a corresponding construction for Property 3. Whereas both constructions are only quoted to preserve weak equivalence, we can verify that, in the special case where the input grammar is a lexicalized LCFRS, they also preserve the set of induced dependency trees. To complete the proof of Lemma 2, we show that every lexicalized LCFRS can be cast into normal forms that satisfy Property 1 and Property 4. It is not hard then to combine the four constructions into a single one that simultaneously establishes all properties of canonical yield functions. ■

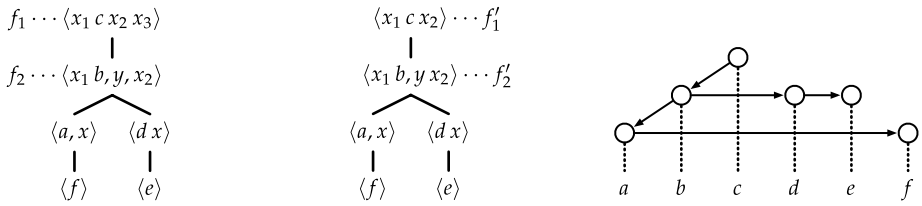
Lemma 3
For every lexicalized LCFRS G one can construct a strongly equivalent lexicalized LCFRS G' such that G' only contains yield functions which satisfy Property 1.

Proof
The proof is very simple. Intuitively, Property 1 enforces a canonical naming of the arguments of yield functions. To establish it, we determine, for every yield function f, a permutation π that renames the argument indices of the variables occurring in the template of f in such a way that the template meets Property 1. This renaming gives rise to a modified yield function f_π. We then replace every rule $A \to f(A_1, \ldots, A_m)$ with the modified rule $A \to f_\pi(A_{\pi(1)}, \ldots, A_{\pi(m)})$. ■

Lemma 4
For every lexicalized LCFRS G one can construct a strongly equivalent lexicalized LCFRS G' such that G' only contains yield functions which satisfy Property 4.

Proof
The idea behind our construction of the grammar G' is perhaps best illustrated by an example. Imagine that the grammar G generates the term t shown in Figure 8a. The yield function $f_1 = \langle x_1\, c\, x_2\, x_3 \rangle$ at the root node of that term violates Property 4, as its template contains the offending substring $x_2\, x_3$. We set up G' in such a way that instead of t it generates the term t' shown in Figure 8b in which f_1 is replaced with the yield function

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024



(a) original term t (b) transformed term t' (c) dependency tree

Figure 8
The transformation implemented by the construction of the grammar G' in Lemma 4.

$f_1' = \langle x_1\, c\, x_2 \rangle$. To obtain f_1' from f_1 we *reduce* the offending substring $x_2\, x_3$ to the single variable x_2. In order to ensure that t and t' induce the same dependency tree (shown in Figure 8c), we then *adapt* the function $f_2 = \langle x_1\, b, y, x_2 \rangle$ at the first child of the root node: Dual to the reduction, we replace the two-component sequence y, x_2 in the template of f_2 with the single component $y\, x_2$; in this way we get $f_2' = \langle x_1\, b, y\, x_2 \rangle$.

Because adaptation operations may introduce new offending substrings, we need a recursive algorithm to compute the rules of the grammar G'. Such an algorithm is given in Table 2. For every rule $A \to f(A_1, \ldots, A_m)$ of G we construct new rules

$$(A, g) \to f'((A_1, g_1), \ldots, (A_m, g_m))$$

where g and the g_i are yield functions encoding adaptation operations. As an example, the adaptation of the function f_2 in the term t may be encoded into the adaptor function $\langle x_1, x_2\, x_3 \rangle$. The function f_2' can then be written as the composition of this function and f_2:

$$f_2' \;=\; \langle x_1, x_2\, x_3 \rangle \circ f_2 \;=\; \langle x_1, x_2\, x_3 \rangle(\langle x_1\, b, y, x_2 \rangle) \;=\; \langle x_1\, b, y\, x_2 \rangle$$

The yield function f' and the adaptor functions g_i are computed based on the template of the g-adapted yield function f, that is, the composed function $g \circ f$. In Table 2 we write this as $f' = \mathit{reduce}(f, g)$ and $g_i = \mathit{adapt}(f, g, i)$, respectively. Let us denote the template of the adapted function $g \circ f$ by τ. An *i-block* of τ is a maximal, non-empty substring of some component of τ that consists of variables with argument index i. To compute the template of g_i we read the i-blocks of τ from left to right and rename the variables by changing their argument indices from i to 1. To compute the template of f' we take the

Table 2
Computing the production rules of an LCFRS in which all yield functions satisfy Property 4.

Input: a linear context-free rewriting system $G = (N, \Sigma, P, S)$

```
1:  P' ← ∅; agenda ← {(S, ⟨x⟩)}; chart ← ∅
2:  while agenda is not empty
3:      remove some (A, g) from agenda
4:      if (A, g) ∉ chart then
5:          add (A, g) to chart
6:          for each rule A → f(A_1, ..., A_m) ∈ P do
7:              f ← reduce(f, g); g_i ← adapt(f, g, i) (1 ≤ i ≤ m)
8:              for each i from 1 to m do
9:                  add (A_i, g_i) to agenda
10:             add (A, g) → f'((A_1, g_1), ..., (A_m, g_m)) to P'
```


Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

template τ and replace the jth i-block with the variable $x_{i,j}$, for all argument indices i and component indices j.

Our algorithm is controlled by an agenda and a chart, both containing pairs of the form (A, g), where A is a nonterminal of G and g is an adaptor function. These pairs also constitute the nonterminals of the new grammar G'. The fan-out of a nonterminal is the fan-out of g. The agenda is initialized with the pair $(S, \langle x \rangle)$ where $\langle x \rangle$ is the identity function; this pair also represents the start symbol of G'. To see that the algorithm terminates, one may observe that the fan-out of every nonterminal (A, g) added to the agenda is upper-bounded by the fan-out of A. Hence, there are only finitely many pairs (A, g) that may occur in the chart, and a finite number of iterations of the *while*-loop. ■

We conclude by noting that when constructing a canonical grammar, one needs to be careful about the order in which the individual constructions (for Properties 1–4) are combined. One order that works is

$$\text{Property } 3 < \text{Property } 4 < \text{Property } 2 < \text{Property } 1$$

6. Parsing and Recognition

Lexicalized linear context-free rewriting systems are able to account for arbitrarily non-projective dependency trees. This expressiveness comes with a price: In this section we show that parsing with lexicalized LCFRSs is intractable, unless we are willing to restrict the class of grammars.

6.1 Parsing Algorithm

To ground our discussion of parsing complexity, we present a simple bottom–up parsing algorithm for LCFRSs, specified as a grammatical deduction system (Shieber, Schabes, and Pereira 1995). Several similar algorithms have been described in the literature (Seki et al. 1991; Bertsch and Nederhof 2001; Kallmeyer 2010). We assume that we are given a grammar $G = (N, \Sigma, P, S)$ and a string $w = a_1 \cdots a_n \in V^*$ to be parsed.

Item form. The items of the deduction system take the form

$$[A, l_1, r_1, \ldots, l_k, r_k]$$

where $A \in N$ with $\varphi(A) = k$, and the remaining components are indices identifying the left and right endpoints of pairwise non-overlapping substrings of w. More formally, $0 \leq l_h \leq r_h \leq n$, and for all h, h' with $h \neq h'$, either $r_h \leq l_{h'}$ or $r_{h'} \leq l_h$. The intended interpretation of an item of this form is that A derives a term $t \in T(G)$ that yields the specified substrings of w, that is,

$$A \Rightarrow_G^* t \qquad \text{and} \qquad \mathit{yield}(t) = \langle a_{l_1+1} \cdots a_{r_1}, \ldots, a_{l_k+1} \cdots a_{r_k} \rangle$$

Goal item. The goal item is $[S, 0, n]$. By this item, there exists a term that can be derived from the start symbol S and yields the full string $\langle w \rangle$.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Inference rules. The inference rules of the deduction system are defined based on the rules in P. Each production rule

$$A \to f(A_1, \ldots, A_m) \qquad \text{with} \qquad f : k_1 \cdots k_m \to k\,, \quad f = \langle \alpha_1, \ldots, \alpha_k \rangle$$

is converted into a set of inference rules of the form

$$\frac{\left[A_1, l_{1,1}, r_{1,1}, \ldots, l_{1,k_1}, r_{1,k_1}\right] \quad \cdots \quad \left[A_m, l_{m,1}, r_{m,1}, \ldots, l_{m,k_m}, r_{m,k_m}\right]}{\left[A, l_1, r_1, \ldots, l_k, r_k\right]} \tag{3}$$

Each such rule is subject to the following constraints. Let $1 \leq h \leq k$, $v \in V^*$, $1 \leq i \leq m$, and $1 \leq j \leq k_i$. We write $\delta(l, v) = r$ to assert that $r = l + |v|$ and that v is the substring of w between indices l and r.

If	$\alpha_h = v$	then	$\delta(l_h, v) = r_h$	(c1)
If	$v\, x_{i,j}$ is a prefix of α_h	then	$\delta(l_h, v) = l_{i,j}$	(c2)
If	$x_{i,j}\, v$ is a suffix of α_h	then	$\delta(r_{i,j}, v) = r_h$	(c3)
If	$x_{i,j}\, v\, x_{i',j'}$ is an infix of α_h	then	$\delta(r_{i,j}, v) = l_{i',j'}$	(c4)

These constraints ensure that the substrings corresponding to the premises of the inference rule can be combined into the substrings corresponding to the conclusion by means of the yield function f.

Based on the deduction system, a tabular parser for LCFRSs can be implemented using standard dynamic programming techniques. This parser will compute a packed representation of the set of all derivation trees that the grammar G assigns to the string w. Such a packed representation is often called a **shared forest** (Lang 1994). In combination with appropriate semirings, the shared forest is useful for many tasks in syntactic analysis and machine learning (Goodman 1999; Li and Eisner 2009).

6.2 Parsing Complexity

We are interested in an upper bound on the runtime of the tabular parser that we have just presented. We can see that the parser runs in time $O(|G||w|^c)$, where $|G|$ denotes the size of some suitable representation of the grammar G, and c denotes the maximal number of instantiations of an inference rule (cf. McAllester 2002). Let us write $c(f)$ for the specialization of c to inference rules for productions with yield function f. We refer to this value as the **parsing complexity** of f (cf. Gildea 2010). Then to show an upper bound on c it suffices to show an upper bound on the parsing complexities of the yield functions that the parser has to handle. An obvious such upper bound is

$$c(f) \;\leq\; 2k + \sum_{i=1}^{m} 2k_i$$

Here we imagine that we could choose each endpoint in Equation (3) independently of all the others. By virtue of the constraints, however, some of the endpoints cannot be chosen freely; in particular, some of the substrings may be adjacent. In general, to show

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

an upper bound $c(f) \leq b$ we specify a strategy for choosing b endpoints, and then argue that, given the constraints, these choices determine the remaining endpoints.

Lemma 5
For a yield function $f : k_1 \cdots k_m \rightarrow k$ we have

$$c(f) \leq k + \sum_{i=1}^{m} k_i$$

Proof
We adopt the following strategy for choosing endpoints: For $1 \leq i \leq k$, choose the value of l_h. Then, for $1 \leq i \leq m$ and $1 \leq j \leq k_i$, choose the value of $r_{i,j}$. It is not hard to see that these choices suffice to determine all other endpoints. In particular, each left endpoint $l_{i',j'}$ will be shared either with the left endpoint l_h of some component (by constraint c2), or with some right endpoint $r_{i,j}$ (by constraint c4). ■

6.3 Universal Recognition

The runtime of our parsing algorithm for LCFRSs is exponential in both the rank and the fan-out of the input grammar. One may wonder whether there are parsing algorithms that can be substantially faster. We now show that the answer to this question is likely to be negative even if we restrict ourselves to canonical lexicalized LCFRSs. To this end we study the universal recognition problem for this class of grammars.

The **universal recognition problem** for a class of linear context-free rewriting systems is to decide, given a grammar G from the class in question and a string w, whether G yields $\langle w \rangle$. A straightforward algorithm for solving this problem is to first compute the shared forest for G and w, and to return "yes" if and only if the shared forest is non-empty. Choosing appropriate data structures, the emptiness of shared forests can be decided in linear time and space with respect to the size of the forest. Therefore, the computational complexity of universal recognition is upper-bounded by the complexity of constructing the shared forest. Conversely, parsing cannot be faster than universal recognition.

In the next three lemmas we prove that the universal recognition problem for canonical lexicalized LCFRSs is NP-complete unless we restrict ourselves to a class of grammars where both the fan-out and the rank of the yield functions are bounded by constants. Lemma 6, which shows that the universal recognition problem of lexicalized LCFRSs is in NP, distinguishes lexicalized LCFRSs from general LCFRSs, for which the universal recognition problem is known to be PSPACE-complete (Kaji et al. 1992). The crucial difference between general and lexicalized LCFRSs is the fact that in the latter, the size of the generated terms is bounded by the length of the input string. Lemma 7 and Lemma 8, which establish two NP-hardness results for lexicalized LCFRSs, are stronger versions of the corresponding results for general LCFRSs presented by Satta (1992), and are proved using similar reductions. They show that the hardness results hold under significant restrictions of the formalism: to lexicalized form and to canonical yield functions. Note that, whereas in Section 5.2 we have shown that every lexicalized LCFRS is equivalent to a canonical one, the normal form transformation increases the size of the original grammar by a factor that is at least exponential in the fan-out.

Lemma 6
The universal recognition problem of lexicalized LCFRSs is in NP.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Proof
Let G be a lexicalized LCFRS, and let w be a string. To test whether G yields $\langle w \rangle$, we guess a term $t \in T(G)$ and check whether t yields $\langle w \rangle$. Let $|t|$ denote the length of some string representation of t. Since the yield functions of G are lexicalized, $|t| \leq |w||G|$. Note that we have

$$|t| \leq |w||G| \leq |w|^2 + 2|w||G| + |G|^2 = (|w| + |G|)^2$$

Using a simple tabular algorithm, we can verify in time $O(|w||G|)$ whether a candidate term t belongs to $T(G)$. It is then straightforward to compute the string yield of t in time $O(|w||G|)$. Thus we have a nondeterministic polynomial-time decider for the universal recognition problem. ■

For the following two lemmas, recall the decision problem 3SAT, which is known to be NP-complete. An instance of 3SAT is a Boolean formula ϕ in conjunctive normal form where each clause contains exactly three literals, which may be either variables or negated variables. We write m for the number of distinct variables that occur in ϕ, and n for the number of clauses. In the proofs the index i will always range over values from 1 to m, and the index j will range over values from 1 to n.

In order to make the grammars in the following reductions more readable, we use yield functions with more than one lexical anchor. Our use of these yield functions is severely restricted, however, and each of our grammars can be transformed into a proper lexicalized LCFRS without affecting the correctness or polynomial size of the reductions.

Lemma 7
The universal recognition problem for canonical lexicalized LCFRSs with unbounded fan-out and rank 1 is NP-hard.

Proof
To prove this claim, we provide a polynomial-time reduction of 3SAT. The basic idea is to use the derivations of the grammar to guess truth assignments for the variables, and to use the feature of unbounded fan-out to ensure that the truth assignment satisfies all clauses.

Let ϕ be an instance of 3SAT. We construct a canonical lexicalized LCFRS G and a string w as follows. Let M denote the $m \times n$ matrix with entries $M_{i,j} = (v_i, c_j)$, that is, entries in the same row share the same variable, and entries in the same column share the same clause. We set up G in such a way that each of its derivations simulates a row-wise iteration over M. Before visiting a new row, the derivation chooses a truth value for the corresponding variable, and sticks to that choice until the end of the row. The string w takes the form

$$w = w_1 \,\$ \cdots \$\, w_n \qquad \text{where} \qquad w_j = c_{j,1} \cdots c_{j,m}\, c_{j,1} \cdots c_{j,m}$$

This string is built up during the iteration over M in a column-wise fashion, where each column corresponds to one component of a tuple with fan-out n. More specifically, for each entry (v_i, c_j), the derivation generates one of two strings, denoted by $\gamma_{i,j}$ and $\bar{\gamma}_{i,j}$:

$$\gamma_{i,j} = c_{j,i} \cdots c_{j,m}\, c_{j,1} \cdots c_{j,i} \qquad \bar{\gamma}_{i,j} = c_{j,i}$$

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

The string $\gamma_{i,j}$ is generated only if v_i can be used to satisfy c_j under the hypothesized truth assignment. By this construction, every successful derivation of G represents a truth assignment that satisfies ϕ. Conversely, using a satisfying truth assignment for ϕ, we will be able to construct a derivation of G that yields w.

To see how the traversal of the matrix M can be implemented by the grammar G, consider the grammar fragment in Figure 9. Each of the rules specifies one possible step of the iteration for the pair (v_i, c_j) under the truth assignment $v_i = \mathit{true}$; rules with left-hand side $F_{i,j}$ (not shown here) specify possible steps under the assignment $v_i = \mathit{false}$. ■

Lemma 8
The universal recognition problem for canonical lexicalized LCFRSs with unbounded rank and fan-out 2 is NP-hard.

Proof
We provide another polynomial-time reduction of 3SAT to a grammar G and a string w, again based on the matrix M mentioned in the previous proof. Also as in the previous reduction, we set up the grammar G to simulate a row-wise iteration over M. The major difference this time is that the entries of M are not visited during one long rank 1 derivation, but during mn rather short fan-out 2 subderivations. The string w is

$$w = w_{\lhd,1} \cdots w_{\lhd,m} \,\$\, w_{\rhd,1} \cdots w_{\rhd,n}$$

$$\text{where} \qquad w_{\lhd,i} = a_{i,1} \cdots a_{i,n}\, b_{i,1} \cdots b_{i,n} \quad \text{and} \quad w_{\rhd,j} = c_{1,j} \cdots c_{m,j}\, c_{1,j} \cdots c_{m,j}$$

During the traversal of M, for each entry (v_i, c_j), we generate a tuple consisting of two substrings of w. The right component of the tuple consists of one the two strings $\gamma_{i,j}$ and $\bar{\gamma}_{i,j}$ mentioned previously. As before, the string $\gamma_{i,j}$ is generated only if v_i can be used to satisfy c_j under the hypothesized truth assignment. The left component consists of one of two strings, denoted by $\sigma_{i,j}$ and $\bar{\sigma}_{i,j}$:

$$\sigma_{i,1} = a_{i,1} \cdots a_{i,n}\, b_{i,1} \qquad \sigma_{i,j} = b_{i,j} \;\; (1 < j) \qquad \bar{\sigma}_{i,n} = a_{i,n}\, b_{i,1} \cdots b_{i,n} \qquad \bar{\sigma}_{i,j} = a_{i,j} \;\; (j < n)$$

These strings are generated to represent the truth assignments $v_i = \mathit{true}$ and $v_i = \mathit{false}$, respectively. By this construction, each substring $w_{\lhd,i}$ can be derived in exactly one of two ways, ensuring a consistent truth assignment for all subderivations that are linked to the same variable v_i.

$$T_{i,j} \rightarrow \langle x_{1,1}, \ldots, x_{1,j-1}, \gamma_{i,j}\, x_{1,j}, x_{1,j+1}, \ldots, x_{1,n} \rangle (T_{i,j+1}) \qquad j < n,\ v_i \text{ occurs in } c_j$$

$$T_{i,j} \rightarrow \langle x_{1,1}, \ldots, x_{1,j-1}, \bar{\gamma}_{i,j}\, x_{1,j}, x_{1,j+1}, \ldots, x_{1,n} \rangle (T_{i,j+1}) \qquad j < n$$

$$T_{i,n} \rightarrow \langle x_{1,1}, \ldots, x_{1,n-1}, \gamma_{i,n}\, x_{1,n} \rangle (T_{i+1,1}) \qquad v_i \text{ occurs in } c_n$$

$$T_{i,n} \rightarrow \langle x_{1,1}, \ldots, x_{1,n-1}, \gamma_{i,n}\, x_{1,n} \rangle (F_{i+1,1}) \qquad v_i \text{ occurs in } c_n$$

$$T_{i,n} \rightarrow \langle x_{1,1}, \ldots, x_{1,n-1}, \bar{\gamma}_{i,n}\, x_{1,n} \rangle (T_{i+1,1})$$

$$T_{i,n} \rightarrow \langle x_{1,1}, \ldots, x_{1,n-1}, \bar{\gamma}_{i,n}\, x_{1,n} \rangle (F_{i+1,1})$$

Figure 9
A fragment of the grammar used in the proof of Lemma 7.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

The grammar G is defined as follows. There is one rather complex rule to rewrite the start symbol S; this rule sets up the general topology of w. Let I be the $m \times n$ matrix with entries $I_{i,j} = (j-1)m + i$. Define $\vec{x}_1$ to be the sequence of variables of the form $x_{h,1}$, where the argument index i is taken from a row-wise reading of the matrix I; in this case, the argument indices in $\vec{x}$ will simply go up from 1 to mn. Now define $\vec{x}_2$ to be the sequence of variables of the form $x_{h,2}$, where h is taken from a column-wise reading of the matrix I. Then S can be expanded with the rule

$$S \rightarrow \langle \vec{x}_1 \, \$ \, \vec{x}_2 \rangle (V_{1,1}, \ldots, V_{1,n}, \ldots, V_{m,1}, \ldots, V_{m,n})$$

Note that there is one nonterminal $V_{i,j}$ for each variable–clause pair (v_i, c_j). These nonterminals can be rewritten using the following rules:

$$V_{i,1} \rightarrow \langle \sigma_{i,1}, x \rangle (T_{i,1}) \qquad V_{i,j} \rightarrow \langle \sigma_{i,j}, x \rangle (T_{i,j})$$
$$V_{i,n} \rightarrow \langle \bar{\sigma}_{i,n}, x \rangle (F_{i,n}) \qquad V_{i,j} \rightarrow \langle \bar{\sigma}_{i,j}, x \rangle (F_{i,j})$$

The remaining rules rewrite the nonterminals $T_{i,j}$ and $F_{i,j}$:

$$T_{i,j} \rightarrow \langle \gamma_{i,j} \rangle \quad (\text{if } v_i \text{ occurs in } c_j) \qquad T_{i,j} \rightarrow \langle \bar{\gamma}_{i,j} \rangle$$
$$F_{i,j} \rightarrow \langle \gamma_{i,j} \rangle \quad (\text{if } \bar{v}_i \text{ occurs in } c_j) \qquad F_{i,j} \rightarrow \langle \bar{\gamma}_{i,j} \rangle$$

It is not hard to see that both G and w can be constructed in polynomial time. ■

7. Block-Degree

To obtain efficient parsing, we would like to have grammars with as low a fan-out as possible. Therefore it is interesting to know how low we can go without losing too much coverage. In lexicalized LCFRSs extracted from dependency treebanks, the fan-out of a grammar has a structural correspondence in the maximal number of blocks per subtree, a measure known as "block-degree." In this section we formally define block-degree, and evaluate grammar coverage under different bounds on this measure.

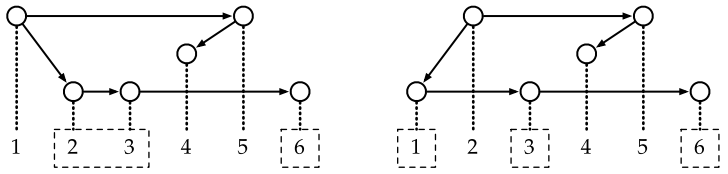
7.1 Definition of Block-Degree

Recall the concept of "blocks" that was defined in Section 4.2. The **block-degree** of a node u of a dependency tree D is the number of distinct blocks of u. The block-degree of D is the maximal block-degree of its nodes.[2]

Example 6
Figure 10 shows two non-projective dependency trees. For D_1, consider the node 2. The descendants of 2 fall into two blocks, marked by the dashed boxes. Because this is the maximal number of blocks per node in D_1, the block-degree of D_1 is 2. Similarly, we can verify that the block-degree of the dependency tree D_2 is 3.

2 We note that, instead of counting the blocks of each node, one may also count the gaps between these blocks and define the "gap-degree" of a dependency tree (Holan et al. 1998).

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024



D_1, block-degree 2 D_2, block-degree 3

Figure 10
Block-degree.

A dependency tree is **projective** if its block-degree is 1. In a projective dependency tree, each subtree corresponds to a substring of the underlying tuple of strings. In a non-projective dependency tree, a subtree may span over several, discontinuous substrings.

7.2 Computing the Block-Degrees

Using a straightforward extension of the algorithm in Table 1, the block-degrees of all nodes of a dependency tree D can be computed in time $O(m)$, where m is the total number of blocks. To compute the block-degree of D, we simply take the maximum over the degrees of each node. We can also adapt this procedure to test whether D is projective, by aborting the computation as soon as we discover that some node has more than one block. The runtime of this test is linear in the number of nodes of D.

7.3 Block-Degree in Extracted Grammars

In a lexicalized LCFRS extracted from a dependency treebank, there is a one-to-one correspondence between the blocks of a node u and the components of the template of the yield function f extracted for u. In particular, the fan-out of f is exactly the block-degree of u. As a consequence, any bound on the block-degree of the trees in the treebank translates into a bound on the fan-out of the extracted grammar. This has consequences for the generative capacity of the grammars: As Seki et al. (1991) show, the class of LCFRSs with fan-out $k > 1$ can generate string languages that cannot be generated by the class of LCFRSs with fan-out $k - 1$.

It may be worth emphasizing that the one-to-one correspondence between blocks and tuple components is a consequence of two characteristic properties of extracted grammars (Properties 3 and 4), and does not hold for non-canonical lexicalized LCFRSs.

Example 7
The following term induces a two-node dependency tree with block-degree 1, but contains yield functions with fan-out 2: $\langle a\, x_1\, x_2\rangle(\langle b, \varepsilon\rangle)$. Note that the yield functions in this term violate both Property 3 and Property 4.

7.4 Coverage on Dependency Treebanks

In order to assess the consequences of different bounds on the fan-out, we now evaluate the block-degree of dependency trees in real-world data. Specifically, we look into five

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

dependency treebanks used in the 2006 CoNLL shared task on dependency parsing (Buchholz and Marsi 2006): the Prague Arabic Dependency Treebank (Hajič et al. 2004), the Prague Dependency Treebank of Czech (Böhmová et al. 2003), the Danish Dependency Treebank (Kromann 2003), the Slovene Dependency Treebank (Džeroski et al. 2006), and the Metu-Sabancı treebank of Turkish (Oflazer et al. 2003). The full data used in the CoNLL shared task also included treebanks that were produced by conversion of corpora originally annotated with structures other than dependencies, which is a potential source of "noise" that one has to take into account when interpreting any findings. Here, we consider only genuine dependency treebanks. More specifically, our statistics concern the training sections of the treebanks that were set off for the task. For similar results on other data sets, see Kuhlmann and Nivre (2006), Havelka (2007), and Maier and Lichte (2011).

Our results are given in Table 3. For each treebank, we list the number of rules extracted from that treebank, as well as the number of corresponding dependency trees. We then list the number of rules that we lose if we restrict ourselves to rules with fan-out $= 1$, or rules with fan-out ≤ 2, as well as the number of dependency trees that we lose because their construction trees contain at least one such rule. We count rule *tokens*, meaning that two otherwise identical rules are counted twice if they were extracted from different trees, or from different nodes in the same tree.

By putting the bound at fan-out 1, we lose between 0.74% (Arabic) and 1.75% (Slovene) of the rules, and between 11.16% (Arabic) and 23.15% (Czech) of the trees in the treebanks. This loss is quite substantial. If we instead put the bound at fan-out ≤ 2, then rule loss is reduced by between 94.16% (Turkish) and 99.76% (Arabic), and tree loss is reduced by between 94.31% (Turkish) and 99.39% (Arabic). This outcome is surprising. For example, Holan et al. (1998) argue that it is impossible to give a theoretical upper bound for the block-degree of reasonable dependency analyses of Czech. Here we find that, if we are ready to accept a loss of as little as 0.02% of the rules extracted from the Prague Dependency Treebank, and up to 0.5% of the trees, then such an upper bound can be set at a block-degree as low as 2.

8. Well-Nestedness

The parsing of LCFRSs is exponential both in the fan-out and in the rank of the grammars. In this section we study "well-nestedness," another restriction on the non-projectivity of dependency trees, and show how enforcing this constraint allows us to restrict our attention to the class of LCFRSs with rank 2.

Table 3
Loss in coverage under the restriction to yield functions with fan-out $= 1$ and fan-out ≤ 2.

			fan-out $= 1$		fan-out ≤ 2	
	rules	trees	rules	trees	rules	trees
Arabic	5,839	1,460	411	163	1	1
Czech	1,322,111	72,703	22,283	16,831	328	312
Danish	99,576	5,190	1,229	811	11	9
Slovene	30,284	1,534	530	340	14	11
Turkish	62,507	4,997	924	580	54	33

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

8.1 Definition of Well-Nestedness

Let D be a dependency tree, and let u and v be nodes of D. The descendants of u and v **overlap**, denoted by $\lfloor u \rfloor \between \lfloor v \rfloor$, if there exist nodes $u_l, u_r \in \lfloor u \rfloor$ and $v_l, v_r \in \lfloor v \rfloor$ such that

$$u_l < v_l < u_r < v_r \qquad \text{or} \qquad v_l < u_l < v_r < u_r$$

A dependency tree D is called **well-nested** if for all pairs of nodes u, v of D

$$\lfloor u \rfloor \between \lfloor v \rfloor \qquad \text{implies that} \qquad \lfloor u \rfloor \cap \lfloor v \rfloor \neq \emptyset$$

In other words, $\lfloor u \rfloor$ and $\lfloor v \rfloor$ may overlap only if u is an ancestor of v, or v is an ancestor of u. If this implication does not hold, then D is called **ill-nested**.

Example 8
Figure 11 shows three non-projective dependency trees. Both D_1 and D_2 are well-nested: D_1 does not contain any overlapping sets of descendants at all. In D_2, although $\lfloor 1 \rfloor$ and $\lfloor 2 \rfloor$ overlap, it is also the case that $\lfloor 1 \rfloor \supseteq \lfloor 2 \rfloor$. In contrast, D_3 is ill-nested, as

$$\lfloor 2 \rfloor \between \lfloor 3 \rfloor \qquad \text{but} \qquad \lfloor 2 \rfloor \cap \lfloor 3 \rfloor = \emptyset$$

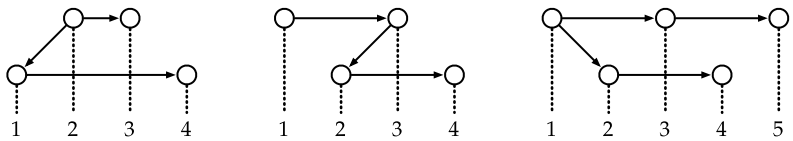
The following lemma characterizes well-nestedness in terms of blocks.

Lemma 9
A dependency tree is ill-nested if and only if it contains two sibling nodes u, v and blocks $\vec{u}_1, \vec{u}_2$ of u and $\vec{v}_1, \vec{v}_2$ of v such that

$$\vec{u}_1 < \vec{v}_1 < \vec{u}_2 < \vec{v}_2 \tag{4}$$

Proof
Let D be a dependency tree. Suppose that D contains a configuration of the form (4). This configuration witnesses that the sets $\lfloor u \rfloor$ and $\lfloor v \rfloor$ overlap. Because u, v are siblings, $\lfloor u \rfloor \cap \lfloor v \rfloor = \emptyset$. Therefore we conclude that D is ill-nested. Conversely now, suppose that D is ill-nested. In this case, there exist two nodes u and v such that

$$\lfloor u \rfloor \between \lfloor v \rfloor \qquad \text{and} \qquad \lfloor u \rfloor \cap \lfloor v \rfloor = \emptyset \tag{$*$}$$



D_1, well-nested D_2, well-nested D_3, ill-nested

Figure 11
Well-nestedness and ill-nestedness.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Here, we may assume u and v to be siblings: otherwise, we may replace either u or v with its parent node, and property $(*)$ will continue to hold. Because $\lfloor u \rfloor \between \lfloor v \rfloor$, there exist descendants $u_l, u_r \in \lfloor u \rfloor$ and $v_l, v_r \in \lfloor v \rfloor$ such that

$$u_l < v_l < u_r < v_r \qquad \text{or} \qquad v_l < u_l < v_r < u_r$$

Without loss of generality, assume that we have the first case. The nodes u_l and u_r belong to different blocks of u, say $\vec{u}_1$ and $\vec{u}_2$; and the nodes v_l and v_r belong to different blocks of v, say $\vec{v}_1$ and $\vec{v}_2$. Then it is not hard to verify Equation (4). ■

Note that projective dependency trees are always well-nested; in these structures, every node has exactly one block, so configuration (4) is impossible. For every $k > 1$, there are both well-nested and ill-nested dependency trees with block-degree k.

8.2 Testing for Well-Nestedness

Based on Lemma 9, testing whether a dependency tree D is well-nested can be done in time linear in the number of blocks in D using a simple subsequence test as follows. We run the algorithm given in Table 1, maintaining a stack $s[u]$ for every node u. The first time we make a down step to u, we push u to the stack for the parent of u; every other time, we pop the stack for the parent until we either find u as the topmost element, or the stack becomes empty. In the latter case, we terminate the computation and report that D is ill-nested; if the computation can be completed without any stack ever becoming empty, we report that D is well-nested.

To show that the algorithm is sound, suppose that some stack $s[p]$ becomes empty when making a down step to some child v of p. In this case, the node v must have been popped from $s[p]$ when making a down step to some other child u of p, and that child must have already been on the stack before the first down step to v. This witnesses the existence of a configuration of the form in Equation (4).

8.3 Well-Nestedness in Extracted Grammars

Just like block-degree, well-nestedness can be characterized in terms of yield functions. Recall the notation $x <_f y$ from Section 5.1. A yield function

$$f : k_1 \cdots k_m \to k\,, \qquad f = \langle \alpha_1, \ldots, \alpha_k \rangle$$

is **ill-nested** if there are argument indices $1 \leq i_1, i_2 \leq m$ with $i_1 \neq i_2$ and component indices $1 \leq j_1, j_1' \leq k_{i_1}$, $1 \leq j_2, j_2' \leq k_{i_2}$ such that

$$x_{i_1,j_1} <_f x_{i_2,j_2} <_f x_{i_1,j_1'} <_f x_{i_2,j_2'} \tag{5}$$

Otherwise, we say that f is **well-nested**. As an immediate consequence of Lemma 9, a restriction to well-nested dependency trees translates into a restriction to well-nested yield functions in the extracted grammars. This puts them into the class of what Kanazawa (2009) calls "well-nested multiple context-free grammars."[3] These grammars

3 Kanazawa (2009) calls a multiple context-free grammar well-nested if each of its rules is non-deleting, non-permuting (our Property 2), and well-nested according to (5).

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

have a number of interesting properties that set them apart from general LCFRSs; in particular, they have a standard pumping lemma (Kanazawa 2009). The yield languages generated by well-nested multiple context-free grammars form a proper subhierarchy within the languages generated by general LCFRSs (Kanazawa and Salvati 2010). Perhaps the most prominent subclass of well-nested LCFRSs is the class of tree-adjoining grammars (Joshi and Schabes 1997).

Similar to the situation with block-degree, the correspondence between structural well-nestedness and syntactic well-nestedness is tight only for canonical grammars. For non-canonical grammars, syntactic well-nestedness alone does not imply structural well-nestedness, nor the other way around.

8.4 Coverage on Dependency Treebanks

To estimate the coverage of well-nested grammars, we extend the evaluation presented in Section 7.4. Table 4 shows how many rules and trees in the five dependency treebanks we lose if we restrict ourselves to well-nested yield functions with fan-out ≤ 2. The losses reported in Table 3 are repeated here for comparison. Although the coverage of well-nested rules is significantly smaller than the coverage of rules without this requirement, rule loss is still reduced by between 92.65% (Turkish) and 99.51% (Arabic) when compared to the fan-out $= 1$ baseline.

8.5 Binarization of Well-Nested Grammars

Our main interest in well-nestedness comes from the following:

Lemma 10
The universal recognition problem for well-nested lexicalized LCFRS with fan-out k and unbounded rank can be decided in time

$$O\left(|G| \cdot |w|^{2k+2}\right)$$

To prove this lemma, we will provide an algorithm for the **binarization** of well-nested lexicalized LCFRSs. In the context of LCFRSs, a binarization is a procedure for transforming a grammar into an equivalent one with rank at most 2. Binarization, either explicit at the level or the grammar or implicit at the level of some parsing algorithm, is essential for achieving efficient recognition algorithms, in particular the usual cubic-time algorithms for context-free grammars. Note that our binarization only

Table 4
Loss in coverage under the restriction to yield functions with fan-out $= 1$, fan-out ≤ 2, and to well-nested yield functions with fan-out ≤ 2 (last column).

	fan-out $= 1$		fan-out ≤ 2		**+ well-nested**			
	rules	trees	rules	trees	rules	trees	**rules**	**trees**
Arabic	5,839	1,460	411	163	1	1	**2**	**2**
Czech	1,322,111	72,703	22,283	16,831	328	312	**407**	**382**
Danish	99,576	5,190	1,229	811	11	9	**17**	**15**
Slovene	30,284	1,534	530	340	14	11	**17**	**13**
Turkish	62,507	4,997	924	580	54	33	**68**	**43**

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

preserves *weak* equivalence; in effect, it reduces the universal recognition problem for well-nested lexicalized LCFRSs to the corresponding problem for well-nested LCFRSs with rank 2. Many interesting semiring computations on the original grammar can be simulated on the binarized grammar, however. A direct parsing algorithm for well-nested dependency trees has been presented by Gómez-Rodríguez, Carroll, and Weir (2011).

The binarization that we present here is a special case of the binarization proposed by Gómez-Rodríguez, Kuhlmann, and Satta (2010). They show that every well-nested LCFRS can be transformed (at the cost of a linear size increase) into a weakly equivalent one in which all yield functions are either constants (that is, have rank 0) or binary functions of one of two types:

$$\langle x_1, \ldots, x_{k_1}\, y_1, \ldots, y_{k_2} \rangle : k_1\, k_2 \to (k_1 + k_2 - 1) \qquad \text{(concatenation)} \quad (6)$$

$$\langle x_1, \ldots, x_j\, y_1, \ldots, y_{k_2}\, x_{j+1}, \ldots, x_{k_1} \rangle : k_1\, k_2 \to (k_1 + k_2 - 2) \qquad \text{(wrapping)} \quad (7)$$

A **concatenation function** takes a k_1-tuple and a k_2-tuple and returns the $(k_1 + k_2 - 1)$-tuple that is obtained by concatenating the two arguments. The simplest concatenation function is the standard concatenation operation $\langle x\, y \rangle$. We will write $conc : k_1\, k_2$ to refer to a concatenation function of the type given in Equation (6). By counting endpoints, we see that the parsing complexity of concatenation functions is

$$c(conc : k_1\, k_2) \;\leq\; 2k_1 + 2k_2 - 1$$

A **wrapping function** takes a k_1-tuple (for some $k_1 \geq 2$) and a k_2-tuple and returns the $(k_1 + k_2 - 2)$-tuple that is obtained by "wrapping" the first argument around the second argument, filling some gap in the former. The simplest function of this type is $\langle x_1\, y\, x_2 \rangle$, which wraps a 2-tuple around a 1-tuple. We write $wrap : k_1\, k_2\, j$ to refer to a wrapping function of the type given in Equation (7). The parsing complexity is

$$c(wrap : k_1\, k_2\, j) \;\leq\; 2k_1 + 2k_2 - 2 \qquad \text{(for all choices of } j\text{)}$$

The constants of the binarized grammar have the form $\langle \varepsilon \rangle$, $\langle \varepsilon, \varepsilon \rangle$, and $\langle a \rangle$, where a is the anchor of some yield function of the original grammar.

8.5.1 Parsing Complexity. Before presenting the actual binarization, we determine the parsing complexity of the binarized grammar. Because the binarization preserves the fan-out of the original grammar, and because in a grammar with fan-out k, for concatenation functions $conc : k_1\, k_2$ we have $k_1 + k_2 - 1 \leq k$ and for wrapping functions $wrap : k_1\, k_2\, j$ we have $k_1 + k_2 - 2 \leq k$, we can rewrite the general parsing complexities as

$$c(conc : k_1\, k_2) \;\leq\; 2k_1 + 2k_2 - 1 \;=\; 2(k_1 + k_2 - 1) + 1 \;\leq\; 2k + 1$$

$$c(wrap : k_1\, k_2\, j) \;\leq\; 2k_1 + 2k_2 - 2 \;=\; 2(k_1 + k_2 - 2) + 2 \;\leq\; 2k + 2$$

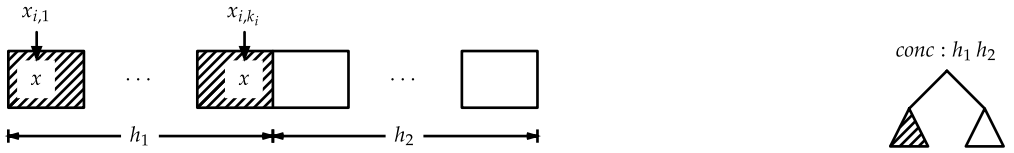
Thus the maximal parsing complexity in the binarized grammar is $2k + 2$; this is achieved by wrapping operations. This gives the bound stated in Lemma 10.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

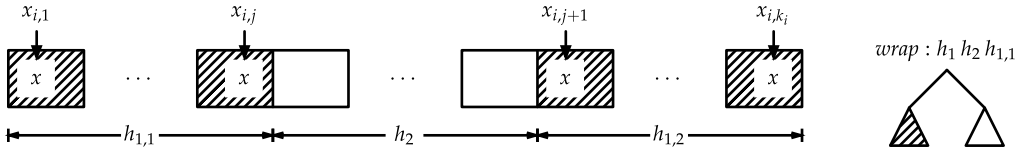
Case 1: The first component of the template starts with the anchor.



Case 2: The first component of the template starts with a variable $x_{i,1}$, but the last component of the template does not end with the variable x_{i,k_i}.



Case 3: The first component of the template starts with a variable $x_{i,1}$, and the last component of the template ends with the variable x_{i,k_i}.



Here, $h_1 = h_{1,1} + h_{1,2}$, and the index j is chosen as the smallest index that maximizes the number of component boundaries between the variables $x_{i,j}$ and $x_{i,j+1}$.

Figure 12
Binarization of well-nested LCFRSs (complex cases).

8.5.2 Binarization. We now turn to the actual binarization. Consider a rule

$$A \to f(A_1, \ldots, A_m)$$

where f is not already a concatenation function, wrapping function, or constant. We decompose this rule into up to three rules

$$A \to f'(B, C) \qquad B \to f_1(B_1, \ldots, B_{m_1}) \qquad C \to f_2(C_1, \ldots, C_{m_2})$$

as follows. We match the template of f against one of three cases, shown schematically in Figure 12. In each case we select a concatenation or wrapping function f' (shown in the right half of the figure), and split up the template of f into two parts defining yield functions f_1 and f_2, respectively. In Figure 12, f_1 is drawn shaded, and f_2 is drawn non-shaded.[4] The split of f partitions the variables that occur in the template, in the sense

4 In order for these parts to make well-defined templates, we will in general need to rename the variables. We leave this renaming implicit here.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

that if for some argument index $1 \leq i \leq m$, either f_1 or f_2 contains *any* variable with argument index i, then it contains *all* such variables. The two sequences

$$B_1, \ldots, B_{m_1} \quad \text{and} \quad C_1, \ldots, C_{m_2} \qquad \text{are obtained from} \qquad A_1, \ldots, A_m$$

by collecting the nonterminal A_i if the variables with argument index i belong to the template of f_1 and f_2, respectively. The nonterminals B and C are fresh nonterminals. We do not create rules for f_1 and f_2 if they are identity functions.

Example 9
We illustrate the binarization by showing how to transform the rule

$$A \to \langle x_1\, a\, x_2\, y_1, y_2, y_3\, x_3 \rangle (A_1, A_2)$$

The template $\langle x_1\, a\, x_2\, y_1, y_2, y_3\, x_3 \rangle$ is complex and matches Case 3 in Figure 12, because its first component starts with the variable x_1 and its last component ends with the variable x_3. We therefore split the template into two smaller parts $\langle x_1\, a\, x_2, x_3 \rangle$ and $\langle y_1, y_2, y_3 \rangle$. The function $\langle y_1, y_2, y_3 \rangle$ is an identity. We therefore create two rules:

$$A \to f_1'(X, A_2)\,, \;\; f_1' = \mathit{wrap} : 2\,3\,1 = \langle x_1\, y_1, y_2, y_3\, x_2 \rangle \qquad X \to \langle x_1\, a\, x_2, x_3 \rangle (A_1)$$

Note that the index j for the wrapping function was chosen to be $j = 2$ because there were more component boundaries between x_2 and x_3 than between x_1 and x_2. The template $\langle x_1\, a\, x_2, x_3 \rangle$ requires further decomposition according to Case 3. This time, the two smaller parts are the identity function $\langle x_1, x_2, x_3 \rangle$ and the constant $\langle a \rangle$. We therefore create the following rules:

$$X \to f_2'(A_1, Y)\,, \;\; f_2' = \mathit{wrap} : 3\,1\,1 = \langle x_1\, y\, x_2, x_3 \rangle \qquad Y \to \langle a \rangle$$

At this point, the transformation ends.

8.5.3 Correctness. We need to show that the fan-out of the binarized grammar does not exceed the fan-out of the original grammar. We reason as follows. Starting from some initial yield function $f_0 : k_1 \cdots k_m \to k$, each step of the binarization decomposes some yield function f into two new yield functions f_1, f_2. Let us denote the fan-outs of the three functions by h, h_1, h_2, respectively. We have

$$h = h_1 + h_2 - 1 \qquad \text{in Case 1 and Case 2} \tag{8}$$

$$h = h_1 + h_2 - 2 \qquad \text{in Case 3} \tag{9}$$

From Equation (8) it is clear that in Case 1 and Case 2, both h_1 and h_2 are upper-bounded by h. In Case 3 we have $h_1 \geq 2$, which together with Equation (9) implies that $h_2 \leq h$. However, h_1 is upper-bounded by h only if $h_2 \geq 2$; if $h_2 = 1$, then h_1 may be greater than h. As an example, consider the decomposition of $\langle x_1\, a\, x_2 \rangle$ (fan-out 1) into the wrapping function $\langle x_1, x_2 \rangle$ (fan-out 2) and the constant $\langle a \rangle$ (fan-out 1). But because in Case 3 the index j is chosen to maximize the number of component boundaries between the variables $x_{i,j}$ and $x_{i,j+1}$, the assumption $h_2 = 1$ implies that each of the h_1 components of f_1 contains at least one variable with argument index i—if there were

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

a component without such a variable, then the two variables that surrounded that component would have given rise to a different choice of j. Hence we deduce that $h_1 \leq k_i$.

9. Conclusion

In this article, we have presented a formalism for non-projective dependency grammar based on linear context-free rewriting systems, along with a technique for extracting grammars from dependency treebanks. We have shown that parsing with the full class of these grammars is intractable. Therefore, we have investigated two constraints on the non-projectivity of dependency trees, block-degree and well-nestedness. Jointly, these two constraints define a class of "mildly" non-projective dependency grammars that can be parsed in polynomial time.

Our results in Sections 7 and 8 allow us to relate the formal power of an LCFRS to the structural properties of the dependency structures that it induces. Although we have used this relation to identify a class of dependency grammars that can be parsed in polynomial time, it also provides us with a new perspective on the question about the descriptive adequacy of a grammar formalism. This question has traditionally been discussed on the basis of strong and weak generative capacity (Bresnan et al. 1982; Huybregts 1984; Shieber 1985). A notion of generative capacity based on dependency trees makes a useful addition to this discussion, in particular when comparing formalisms for which no common concept of strong generative capacity exists. As an example for a result in this direction, see Koller and Kuhlmann (2009).

We have defined the dependency trees that an LCFRS induces by means of a compositional mapping on the derivations. While we would claim that compositionality is a generally desirable property, the particular notion of induction is up for discussion. In particular, our interpretation of derivations may not always be in line with how the grammar producing these derivations is actually *used*. One formalism for which such a mismatch between derivation trees and dependency trees has been pointed out is tree-adjoining grammar (Rambow, Vijay-Shanker, and Weir 1995; Candito and Kahane 1998). Resolving this mismatch provides an interesting line of future work.

One aspect that we have not discussed here is the linguistic adequacy of block-degree and well-nestedness. Each of our dependency grammars is restricted to a finite block-degree. As a consequence of this restriction, our dependency grammars are not expressive enough to capture linguistic phenomena that require unlimited degrees of non-projectivity, such as the "scrambling" in German subordinate clauses (Becker, Rambow, and Niv 1992). The question whether it is reasonable to assume a bound on the block-degree of dependency trees, perhaps for some performance-based reason, is open. Likewise, it is not clear whether well-nestedness is a "natural" constraint on dependency analyses (Chen-Main and Joshi 2010; Maier and Lichte 2011).

Although most of the results that we have presented in this article are of a theoretical nature, some of them have found their way into practical systems. In particular, the extraction technique from Section 4 is used by the data-driven dependency parser of Maier and Kallmeyer (2010).

Acknowledgments

The author gratefully acknowledges financial support from The German Research Foundation (Sonderforschungsbereich 378, project MI 2) and The Swedish Research Council (diary no. 2008-296).

References

Becker, Tilman, Owen Rambow, and Michael Niv. 1992. The derivational generative power of formal systems, or: Scrambling is beyond LCFRS. IRCS Report 92-38, University of Pennsylvania, Philadelphia, PA.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Bertsch, Eberhard and Mark-Jan Nederhof. 2001. On the complexity of some extensions of RCG parsing. In *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT)*, pages 66–77, Beijing.

Bodirsky, Manuel, Marco Kuhlmann, and Mathias Möhl. 2005. Well-nested drawings as models of syntactic structure. In *Proceedings of the 10th Conference on Formal Grammar (FG) and Ninth Meeting on Mathematics of Language (MOL)*, pages 195–203, Edinburgh.

Böhmová, Alena, Jan Hajič, Eva Hajičová, and Barbora Hladká. 2003. The Prague Dependency Treebank: A three-level annotation scenario. In Abeillé, Anne, editor. *Treebanks: Building and Using Parsed Corpora*. Kluwer Academic Publishers, Dordrecht, chapter 7, pages 103–127.

Boullier, Pierre. 1998. Proposal for a natural language processing syntactic backbone. Rapport de recherche 3342, INRIA Rocquencourt, Paris, France.

Boullier, Pierre. 2004. Range Concatenation Grammars. In Harry C. Bunt, John Carroll, and Giorgio Satta, editors, *New Developments in Parsing Technology*, volume 23 of *Text, Speech and Language Technology*. Kluwer Academic Publishers, Dordrecht, pages 269–289.

Bresnan, Joan, Ronald M. Kaplan, Stanley Peters, and Annie Zaenen. 1982. Cross-serial dependencies in Dutch. *Linguistic Inquiry*, 13(4):613–635.

Buchholz, Sabine and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL)*, pages 149–164, New York, NY.

Candito, Marie-Hélène and Sylvain Kahane. 1998. Can the TAG derivation tree represent a semantic graph? An answer in the light of Meaning-Text Theory. In *Proceedings of the Fourth Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+)*, pages 21–24, Philadelphia, PA.

Charniak, Eugene. 1996. Tree-bank grammars. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI) and Eighth Innovative Applications of Artificial Intelligence Conference (IAAI)*, volume 2, pages 1031–1036, Portland, OR.

Chen-Main, Joan and Aravind K. Joshi. 2010. Unavoidable ill-nestedness in natural language and the adequacy of tree local-MCTAG induced dependency structures. In *Proceedings of the Tenth International Conference on Tree Adjoining Grammars and Related Formalisms (TAG+)*, New Haven, CT. Available at `http://dx.doi.org/10.1093/logcom/exs012`.

Crescenzi, Pierluigi, Daniel Gildea, Andrea Marino, Gianluca Rossi, and Giorgio Satta. 2011. Optimal head-driven parsing complexity for linear context-free rewriting systems. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 450–459, Portland, OR.

Džeroski, Sašo, Tomaž Erjavec, Nina Ledinek, Petr Pajas, Zdenek Žabokrtsky, and Andreja Žele. 2006. Towards a Slovene dependency treebank. In *Fifth International Conference on Language Resources and Evaluations (LREC)*, pages 1388–1391, Genoa.

Gaifman, Haim. 1965. Dependency systems and phrase-structure systems. *Information and Control*, 8(3):304–337.

Gildea, Daniel. 2010. Optimal parsing strategies for linear context-free rewriting systems. In *Proceedings of Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 769–776, Los Angeles, CA.

Gómez-Rodríguez, Carlos, John Carroll, and David J. Weir. 2011. Dependency parsing schemata and mildly non-projective dependency parsing. *Computational Linguistics*, 37(3):541–586.

Gómez-Rodríguez, Carlos, Marco Kuhlmann, and Giorgio Satta. 2010. Efficient parsing of well-nested linear context-free rewriting systems. In *Proceedings of Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 276–284, New Haven, CT.

Gómez-Rodríguez, Carlos, Marco Kuhlmann, Giorgio Satta, and David J. Weir. 2009. Optimal reduction of rule length in linear context-free rewriting systems. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 539–547, Boulder, CO.

Gómez-Rodríguez, Carlos and Giorgio Satta. 2009. An optimal-time binarization algorithm for linear context-free rewriting systems with fan-out two. In *Proceedings of the Joint Conference of the 47th Annual*

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Meeting of the Association for Computational Linguistics (ACL) and the Fourth International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (IJCNLP), pages 985–993, Singapore.

Goodman, Joshua. 1999. Semiring parsing. *Computational Linguistics*, 25(4):573–605.

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 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 *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 608–615, Prague.

Hays, David G. 1964. Dependency theory: A formalism and some observations. *Language*, 40(4):511–525.

Holan, Tomáš, Vladislav Kuboň, Karel Oliva, and Martin Plátek. 1998. Two useful measures of word order complexity. In *Proceedings of the Workshop on Processing of Dependency-Based Grammars*, pages 21–29, Montréal.

Hudson, Richard. 2007. *Language Networks. The New Word Grammar*. Oxford University Press, Oxford.

Huybregts, Riny. 1984. The weak inadequacy of context-free phrase structure grammars. In Ger de Haan, Mieke Trommelen, and Wim Zonneveld, editors, *Van periferie naar kern*. Foris, Dordrecht, pages 81–99.

Joshi, Aravind K. and Yves Schabes. 1997. Tree-Adjoining Grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer, Berlin, pages 69–123.

Kaji, Yuichi, Ryuichi Nakanishi, Hiroyuki Seki, and Tadao Kasami. 1992. The universal recognition problems for multiple context-free grammars and for linear context-free rewriting systems. *IEICE Transactions on Information and Systems*, E75-D(1):78–88.

Kallmeyer, Laura. 2010. *Parsing Beyond Context-Free Grammars*. Springer, Berlin.

Kanazawa, Makoto. 2009. The pumping lemma for well-nested multiple context-free languages. In *Developments in Language Theory. Proceedings of the 13th International Conference, DLT 2009*, volume 5583 of *Lecture Notes in Computer Science*, pages 312–325, Stuttgart.

Kanazawa, Makoto and Sylvain Salvati. 2010. The copying power of well-nested multiple context-free grammars. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications. Proceedings of the 4th International Conference, LATA 2010*, volume 6031 of *Lecture Notes in Computer Science*, pages 344–355, Trier.

Koller, Alexander and Marco Kuhlmann. 2009. Dependency trees and the strong generative capacity of CCG. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 460–468, Athens.

Kracht, Marcus. 2003. *The Mathematics of Language*, volume 63 of *Studies in Generative Grammar*. Mouton de Gruyter, Paris.

Kromann, Matthias Trautner. 2003. The Danish Dependency Treebank and the underlying linguistic theory. In *Proceedings of the Second Workshop on Treebanks and Linguistic Theories (TLT)*, pages 217–220, Växjö.

Kübler, Sandra, Ryan McDonald, and Joakim Nivre. 2009. *Dependency Parsing*. Synthesis Lectures on Human Language Technologies. Morgan and Claypool.

Kuhlmann, Marco and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *Proceedings of the 21st International Conference on Computational Linguistics (COLING) and 44th Annual Meeting of the Association for Computational Linguistics (ACL) Main Conference Poster Sessions*, pages 507–514, Sydney.

Kuhlmann, Marco and Giorgio Satta. 2009. Treebank grammar techniques for non-projective dependency parsing. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 478–486, Athens.

Lang, Bernard. 1994. Recognition can be harder than parsing. *Computational Intelligence*, 10(4):486–494.

Li, Zhifei and Jason Eisner. 2009. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 40–51, Singapore.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Maier, Wolfgang and Laura Kallmeyer. 2010. Discontinuity and non-projectivity: Using mildly context-sensitive formalisms for data-driven parsing. In *Proceedings of the Tenth International Conference on Tree Adjoining Grammars and Related Formalisms (TAG+)*, New Haven, CT.

Maier, Wolfgang and Timm Lichte. 2011. Characterizing discontinuity in constituent treebanks. In Philippe de Groote, Markus Egg, and Laura Kallmeyer, editors, *Formal Grammar. Proceedings of the 14th International Conference, FG 2009, Revised Selected Papers*, volume 5591 of *Lecture Notes in Computer Science*, pages 167–182, Bordeaux.

Maier, Wolfgang and Anders Søgaard. 2008. Treebanks and mild context-sensitivity. In *Proceedings of the 13th Conference on Formal Grammar (FG)*, pages 61–76, Hamburg.

McAllester, David. 2002. On the complexity analysis of static analyses. *Journal of the Association for Computing Machinery*, 49(4):512–537.

Mel'čuk, Igor. 1988. *Dependency Syntax: Theory and Practice*. State University of New York Press, Albany, NY.

Michaelis, Jens. 1998. Derivational minimalism is mildly context-sensitive. In *Logical Aspects of Computational Linguistics, Third International Conference, LACL 1998, Selected Papers*, volume 2014 of *Lecture Notes in Computer Science*, pages 179–198, Grenoble.

Michaelis, Jens. 2001. *On Formal Properties of Minimalist Grammars*. Ph.D. thesis, Universität Potsdam, Potsdam, Germany.

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 Joint Conference on Empirical Methods in Natural Language Processing (EMNLP) and Computational Natural Language Learning (CoNLL)*, pages 915–932, Prague.

Oflazer, Kemal, Bilge Say, Dilek Zeynep Hakkani-Tür, and Gökhan Tür. 2003. Building a Turkish treebank. In Abeillé, Anne, editor. *Treebanks: Building and Using Parsed Corpora*. Kluwer Academic Publishers, Dordrecht, chapter 15, pages 261–277.

Rambow, Owen and Aravind K. Joshi. 1997. A formal look at dependency grammars and phrase-structure grammars, with special consideration of word-order phenomena. In Leo Wanner, editor, *Recent Trends in Meaning-Text Theory*, volume 39 of *Studies in Language, Companion Series*. John Benjamins, Amsterdam, pages 167–190.

Rambow, Owen, K. Vijay-Shanker, and David J. Weir. 1995. D-Tree grammars. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 151–158, Cambridge, MA.

Sagot, Benoît and Giorgio Satta. 2010. Optimal rank reduction for linear context-free rewriting systems with fan-out two. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 525–533, Uppsala.

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

Schabes, Yves. 1990. *Mathematical and Computational Aspects of Lexicalized Grammars*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA.

Schabes, Yves, Anne Abeillé, and Aravind K. Joshi. 1988. Parsing strategies with 'lexicalized' grammars: Application to tree adjoining grammars. In *Proceedings of the Twelfth International Conference on Computational Linguistics (COLING)*, pages 578–583, Budapest.

Seki, Hiroyuki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On Multiple Context-Free Grammars. *Theoretical Computer Science*, 88(2):191–229.

Sgall, Petr, Eva Hajičová, and Jarmila Panevová. 1986. *The Meaning of the Sentence in Its Semantic and Pragmatic Aspects*. Springer, Berlin.

Shieber, Stuart M. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343.

Shieber, Stuart M., Yves Schabes, and Fernando Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36.

Steedman, Mark and Jason Baldridge. 2011. Combinatory categorial grammar. In Robert D. Borsley and Kersti Börjars, editors, *Non-Transformational Syntax: Formal and Explicit Models of Grammar*. Wiley-Oxford, Blackwell, chapter 5, pages 181–224.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Tesnière, Lucien. 1959. *Éléments de syntaxe structurale*. Klinksieck, Paris.

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)*, pages 104–111, Stanford, CA.

Villemonte de la Clergerie, Éric. 2002. Parsing mildly context-sensitive languages with thread automata. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING)*, pages 1–7, Taipei.

Weir, David J. 1988. *Characterizing Mildly Context-Sensitive Grammar Formalisms*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/COLI_a_00125 by guest on 28 March 2024