Lecture Notes in Artificial Intelligence 6270

Edited by R. Goebel, J. Siekmann, and W. Wahlster

Subseries of Lecture Notes in Computer Science

FoLLI Publications on Logic, Language and Information

Marco Kuhlmann

Dependency Structures and Lexicalized Grammars

An Algebraic Approach

Springer

Volume Editor

Marco Kuhlmann
Uppsala University, Department of Linguistics and Philology
Box 635, 75126 Uppsala, Sweden
E-mail: marco.kuhlmann@lingfil.uu.se

Library of Congress Control Number: 2010930607

CR Subject Classification (1998): I.2, I.2.7, I.7, F.4.1, F.4.3, F.3

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN 0302-9743
ISBN-10 3-642-14567-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-14567-4 Springer Berlin Heidelberg New York

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 06/3180 5 4 3 2 1 0

Foreword

Since 2002, FoLLI, the Association for Logic, Language, and Information (www.folli.org), has awarded an annual prize for an outstanding dissertation in the fields of logic, language, and information. The prize is named after the well-known Dutch logician Evert Willem Beth, whose interdisciplinary interests are in many ways exemplary of the aims of FoLLI. It is sponsored by the E.W. Beth Foundation. Dissertations submitted for the prize are judged on technical depth and strength, originality, and impact made in at least two of the three fields of logic, language, and computation. Every year the competition is strong and the interdisciplinary character of the award stimulates lively debate in the Beth Prize Committee.

Recipients of the award are offered the opportunity to prepare a book version of their thesis for publication in the *FoLLI Publications on Logic, Language and Information.*

This volume is based on the PhD thesis of Marco Kuhlmann, who was a joint winner of the E.W. Beth dissertation award in 2008. We wish to quote here the Committee's motivation for co-awarding the Beth Prize to him:

> Marco Kuhlmann's thesis on 'Dependency Structures and Lexicalized Grammars', in the area of Language and Computation, lays new theoretical foundations for the study of non-projective dependency grammars. Such grammars have recently become increasingly important for approaches to statistical parsing in computational linguistics that deal with free word order and long-distance dependencies. Dr. Kuhlmann provides new formal tools to define and understand dependency grammars, presents two new dependency language hierarchies with polynomial parsing algorithms, establishes the practical significance of these hierarchies through corpus studies, and links his work to the phrase-structure grammar tradition through an equivalence result with tree-adjoining grammars. Dr. Kuhlmann's thesis bridges gaps between linguistics and theoretical computer science,

between theoretical and empirical approaches in computational linguistics, and between previously disconnected strands of formal language research. It is highly original and beautifully presented.

Valentin Goranko
(Chair of the Beth Prize Committee in 2008)
Michael Moortgat
(President of the Association for Logic, Language, and Information)

Preface

This book reports the major results of roughly four years of doctoral research. Among the many people who have contributed to it, there are some to whom I owe a particularly large debt of gratitude.

My first thanks goes to my supervisor, Gert Smolka. He granted me the freedom to develop my own ideas, encouraged me to strive for simplicity in their presentation, and provided guidance and advice. I also thank Manfred Pinkal and Aravind Joshi for accepting to give their expert opinion on this dissertation, and to Raimund Seidel and Tilman Becker for agreeing to join my examination committee.

Guido Tack, as my office-mate, had to suffer from my meanderings on a large number of sometimes errant ideas related to this dissertation. I thank him for the patience he had with me, for his valuable feedback, and most of all, for his friendship. Mathias Möhl bore with me in endless discussions, and made essential contributions to this work. Joachim Niehren introduced me to tree automata and the algebraic perspective on formal languages. Without him, the machinery used in this dissertation would be far less canonical. Thanks also go to my coauthors: Mathias, Manuel Bodirsky, Robert Grabowski, and Joakim Nivre. Joachim, Bernhard Fisseni, Sebastian Maneth, and Martin Plátek provided me with the opportunity to present preliminary results of my work to their research groups.

The research reported in this dissertation was carried out in the context of the CHORUS project, which was part of the Collaborative Research Centre 378, 'Resource-adaptive cognitive processes,' and funded by the Deutsche Forschungsgemeinschaft, DFG. Throughout my doctoral studies, I have also been associated to the International Research Training Group 715, 'Language technology and cognitive systems,' likewise funded by the DFG. In both contexts, I have been privileged to work and study together with Ralph Debusmann, Alexander Koller, and Stefan Thater. Alexander deserves a special thanks for his constant enthusiasm about our research, which has been a source of motivation for me ever since I first met him as a freshman.

Between 2002 and 2008, my desk was in the Programming Systems Lab at Saarland University. I thank the other members of the Lab for creating a pleasant working environment: Gert, Guido, Mathias, Ralph, Mark Kaminski, Sandra Neumann, and Jan Schwinghammer; Joachim, Ondřej Bojar, Thorsten Brunklaus, Denys Duchier, Leif Kornstaedt, Didier Le Botlan, Tim Priesnitz, Andreas Rossberg, Gábor Szokoli, Lutz Straßburger, and Ann van de Veire. Thanks also go to our student assistants Robert Künnemann and Christophe Boutter.

Beyond the people I have already mentioned, there were many more who have listened patiently, answered thoroughly, and been helpful during the composition of this dissertation in all kinds of ways. I specifically want to mention Carlos Areces, Jason Baldridge, Achim Blumensath, Cem Bozsahin, Paul Braffort, Matthias Buch-Kromann, Balder ten Cate, Lukas Champollion, Joan Chen-Main, David Chiang, Michael Daniels, Arno Eigenwillig, Jason Eisner, Phillippe Flajolet, Mary Harper, Günter Hotz, Laura Kallmeyer, Anna Kasprzik, Stephan Kepser, Geert-Jan Kruijff, Ivana Kruijff-Korbayová, Markéta Lopatková, Detmar Meurers, Jens Michaelis, Uwe Mönnich, Timo von Oertzen, Gerald Penn, Fernando Pereira, Giorgio Satta, Yves Schabes, Tatjana Scheffler, Ingo Schröder, Sabine Schulte im Walde, Stuart Shieber, Oliver Suhre, Martin Volk, David Weir, Anssi Yli-Jyrä, and Zdeněk Žabokrtský. Gert, Mathias, Joachim, Ralph, Stefan, and Jiří have all read drafts of parts of my dissertation, and given valuable feedback. Alexander provided me with comments on the present book version of the dissertation.

The final phase of my doctoral studies made me appreciate how extremely fortunate I am to be surrounded by a large number of supportive friends and colleagues, such as Colin Bannard, Chris Callison-Burch, Danielle Matthews, Ulrike and Sebastian Padó, Monika Schwarz, and Markus Zacharski. A very special thanks goes to Lars Charbonnier, for his friendship and support during the last 13 years.

Finally, I want to express my love and gratitude to my family, and most of all to my wife, Antje, and my son, Matthias.

May 2010 Marco Kuhlmann

Contents

1

Introduction

In this book, we develop the formal theory of *dependency structures*, show how combining these structures with a regular means of composition yields infinite hierarchies of ever more powerful *dependency languages*, and classify several grammar formalisms with respect to the languages in these hierarchies that they are able to characterize. Our results show that the generative capacity and the parsing complexity of lexicalized grammar formalisms can be systematically related to structural properties of the dependency graphs that these formalisms can induce.

1.1 Motivation

Syntactic representations based on word-to-word dependencies, *dependency structures*, have a long tradition in descriptive linguistics. Since the seminal work of Tesnière [113], they have become the basis for several linguistic theories, such as Functional Generative Description [106], Meaning-Text Theory [77], and Word Grammar [51]. In recent years, they have also been used for a wide range of computational applications, such as information extraction [14], textual inference [39], and machine translation [94]. 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 [6, 89].

1.1.1 Dependency Structures

The basic assumptions behind the notion of dependency are summarized in the following sentences from the seminal work of Tesnière [113], ch. 1, §§ 2–4, and ch. 2, §§ 1–2:[1]

[1] The English translation (by the author) is based on a German translation [114].

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 1–10, 2010.

> The sentence is an *organized whole*; its constituent parts are the *words*. Every word that functions as part of a sentence is no longer isolated as in the dictionary: the mind perceives *connections* between the word and its neighbours; the totality of these connections forms the scaffolding of the sentence. The structural connections establish relations of *dependency* among the words. Each such connection in principle links a *superior* term and an *inferior* term. The superior term receives the name *governor* (*régissant*); the inferior term receives the name *dependent* (*subordonné*).

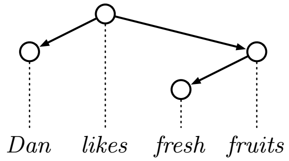


Fig. 1.1. A dependency structure

We can represent the dependency relations among the words of a sentence as a graph. More specifically, the *dependency structure* for a sentence $\vec{w} = w_1 \cdots w_n$ is the directed graph on the set of positions of $\vec{w}$ that contains an edge $i \rightarrow j$ if and only if w_j depends on w_i. In this way, just like strings and parse trees, dependency structures can capture information about certain aspects of the linguistic structure of a sentence. As an example, consider Figure 1.1. In this graph, the edge between the word *likes* and the word *Dan* may be understood to encode the syntactic information that *Dan* is the subject of *likes*; similarly, the edge between *likes* and *fruits* may be interpreted as saying that *fruits* is the direct object. When visualizing dependency structures, we represent (occurrences of) words by circles, and dependencies among them by arrows: the source of an arrow marks the governor of the corresponding dependency, the target marks the dependent. Furthermore, following Hays [47], we use dotted lines (and call them *projection lines*) to indicate the left-to-right ordering of the words in the sentence. Note that these lines do not belong to the graph structure as such.

While there exist both a considerable practical interest in dependency structures and an extensive linguistic literature, dependency syntax has remained somewhat of an island from a formal point of view. In particular, there are few results that bridge between dependency syntax and other traditions, such as phrase-structure or categorial syntax. This makes it hard to gauge the similarities and differences in how the different paradigms can be used to model specific phenomena, and hampers the exchange of linguistic resources and computational methods.

1.1.2 Generative Capacity and Non-projectivity

One of the few bridging results for dependency grammar is due to Gaifman [27], 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. While this result is of fundamental importance from a theoretical point of view, its practical usefulness is limited for at least two reasons.

For one thing, it is generally accepted today that context-free grammars are not adequate for the description of natural language. Independently of each other, Huybregts [52] and Shieber [107] showed that certain constructions in Swiss German require grammar formalisms that adequately model these constructions to generate the so-called copy language, which is beyond the string-generative capacity (and, a forteriori, the tree-generative capacity) of context-free grammars. If we accept this analysis, then we must conclude that context-free grammars are too weak, and that we should look out for more powerful formalisms. Unfortunately, the first class of formal languages in the Chomsky hierarchy that *does* contain the copy language, the class of *context-sensitive languages*, is too big a leap: it contains many languages that are considered to be beyond human capacity (such as the set of all prime numbers), and while context-free grammars can be parsed in polynomial time, the parsing problem of context-sensitive grammars is PSPACE-complete. For such problems, it is widely suspected that they cannot be solved in (deterministic or non-deterministic) polynomial time.

In search of a class of grammars that extends context-free grammar by the minimal amount of generative power that is needed to account for natural language, several so-called *mildly context-sensitive grammar formalisms* have been developed; perhaps best-known among these is Tree Adjoining Grammar (TAG) [54]. The class of string languages generated by TAGs contains the copy language, but unlike general context-sensitive grammars, TAGs can be parsed in polynomial time.

A second reason why the usefulness of Hays-Gaifman grammar is limited is that it is restricted to structures that meet a constraint called *projectivity*, which is similar to the ban on discontinuous constituents in phrase-structure syntax. Specifically, it requires each dependency subtree to cover a contiguous region of the sentence. Projectivity is interesting because the close relation between dependency and word order that it enforces can be exploited in parsing algorithms [21]. However, in recent literature, there is a growing interest in *non-projective dependency structures*, in which a subtree may be spread out over a discontinuous region of the sentence. Such representations naturally arise in the syntactic analysis of linguistic phenomena such as topicalization, and are particularly frequent in syntactic corpora for languages with flexible word order [44]. Unfortunately, most formal and computational results on non-projectivity are rather discouraging. In particular, non-projective dependency parsing is NP-complete [75, 83].

In search of a balance between the benefit of more expressiveness and the penalty of increased processing complexity, several authors have proposed structural constraints that relax the projectivity restriction, but at the same time ensure that the resulting classes of graphs are computationally well-behaved [44, 85, 121]. Such constraints identify classes of what we may call *mildly non-projective dependency structures*.

Given that Hays-Gaifman grammar is too weak both with respect to its string-generative capacity and with respect to the class of dependency structures that it is able to describe, an important problem is to extend Gaifman's equivalence result to grammars that can generate non-context-free string languages, and non-projective dependency trees. In this book, we provide a solution to this problem. In the next section, we outline the framework in which we will develop it.

1.2 Lexicalized Grammars Induce Dependency Trees

Grammar formalisms are mathematical devices that are developed to give explicit descriptions of linguistic theories. One of the fundamental questions that we can ask about a grammar formalism is, whether it adequately models natural language. One way to answer this question is by studying the *generative capacity* of the formalism at hand.

To focus the discussion, let us consider the case of context-free grammars. For these grammars, there are two standard measures of generative capacity: we can interpret them as generators of strings (*string-generative capacity*), or as generators of parse trees (*tree-generative capacity*). Figure 1.2 shows a toy context-free grammar together with a parse tree for a simple English sentence. Strings and parse trees are closely related. In particular, for each string generated by a context-free grammar, there is at least one parse tree from which this string can be recovered by reading the leaves of the tree from left to right. Formally, this reading-off can be described as a *homomorphism* from parse trees to strings—a structure-preserving map.

An interesting property of the context-free grammar in Figure 1.2 is that it is *lexicalized*: every production of the grammar contains exactly one terminal

S → SUBJ *likes* OBJ
SUBJ → *Dan*
OBJ → MOD *fruits*
MOD → *fresh*

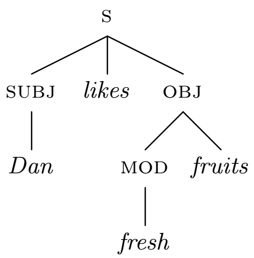


Fig. 1.2. A context-free grammar and a parse tree generated by this grammar

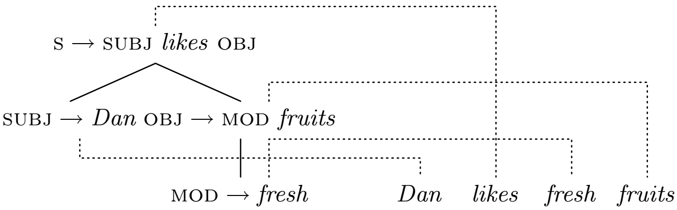


Fig. 1.3. Lexicalized derivations induce dependency structures

symbol, the *anchor* of that production (cf. [104]). Lexicalized grammars play a significant role in contemporary linguistic theories and practical applications. Crucially, they allow us to give context-free grammars a third interpretation: as generators of dependency trees.

Consider a derivation d of a terminal string $\vec{a}$ by means of a context-free grammar. A *derivation tree* for d is a tree in which the nodes are labelled with (occurrences of) the productions used in d, and the edges indicate how these productions were combined. To give an example, the left half of Figure 1.3 shows the unique derivation tree of our example grammar. If the underlying grammar is lexicalized, then there is a one-to-one correspondence between the nodes in the derivation tree and the positions in the derived string $\vec{a}$: every production that participates in the derivation contributes exactly one terminal symbol to this string: its anchor. If we now order the nodes of the derivation tree according to the string positions of their corresponding anchors, then what we get is a dependency structure. For our example, this procedure results in the structure depicted in Figure 1.1. We say that this dependency structure is *induced* by the derivation d.

Not all practically relevant dependency structures can be induced by derivations in lexicalized context-free grammars. A classic example is provided by the structural difference between the verb-argument dependencies in German and Dutch subordinate clauses, as shown in Figure 1.4: context-free grammar can only characterize the 'nested', projective dependencies of German (top), but not the 'cross-serial', non-projective assignments of Dutch (bottom). However, as we will show in this book, these structures can be induced by the derivation trees of lexicalized Tree Adjoining Grammar (TAG) [53]. More generally, based on the concept of induction that we have outlined here we will show that there is a systematic relation between structural properties of dependency trees (such as projectivity) and language-theoretic properties of grammar formalisms inducing them (such as context-freeness).

The main question that we ask in this book is,

> Which grammars induce which sets of dependency structures?

An answer to this question is interesting for at least two reasons. First, it allows us to use dependency structures as the basis of an alternative measure for

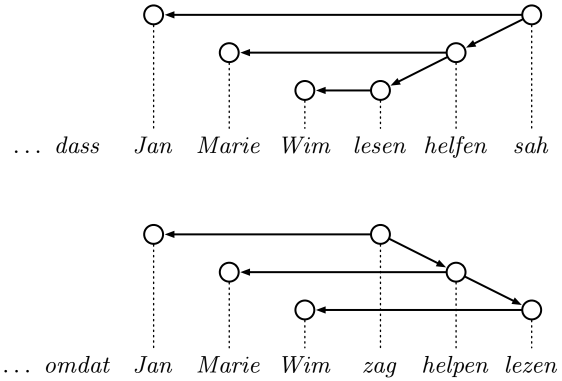


Fig. 1.4. Nested and cross-serial dependencies

the generative capacity of a grammar formalism. This is attractive, as dependency structures are more informative than strings, but less formalism-specific and more intuitively accessible than parse trees (cf. [57]). Second, an answer to the question allows us to tap the rich resource of formal results about grammar formalisms and transfer them to work on dependency representations. In particular, it allows us to import the expertise in developing parsing algorithms for lexicalized grammar formalisms into the field of dependency parsing (cf. [75]).

While the connection between the generative capacity of a grammar formalism and the structural properties of the dependency graphs that this formalism can induce is intuitive, there have been only few results that link the two dimensions. We believe that a fundamental reason for the lack of such bridging results is that, while structural constraints on dependency graphs are *internal* properties in the sense that they concern the nodes of the graph and their connections, grammars take an *external* perspective on the objects that they manipulate—the internal structure of an object is determined by the internal structure of its constituent parts and the operations that are used to combine them. An example for the difference between the two views is given by the different perspectives on trees that we find in graph theory and universal algebra. In graph theory, a tree is a special graph with an internal structure that meets certain constraints; in algebra, trees are abstract objects that can be composed and decomposed using certain operations. The development of such an algebraic view on dependency trees will provide the technical key to the results that we present in this book. Once we have it, we will be able to generalize Gaifman's [1965] equivalence result from projective dependency structures and context-free grammars to mildly non-projective structures and mildly context-sensitive grammar formalisms.

1.3 Overview of the Book

This book consists of two parts. In the first part, we develop an algebraic framework within which lexicalized grammars can be compared based on the structural properties of the dependency graphs that they induce. In the second part, we derive a natural notion of regular sets of dependency structures, and use it to study the connection between structural properties such as projectivity on the one hand, and language-theoretic properties such as string-generative capacity and parsing complexity on the other.

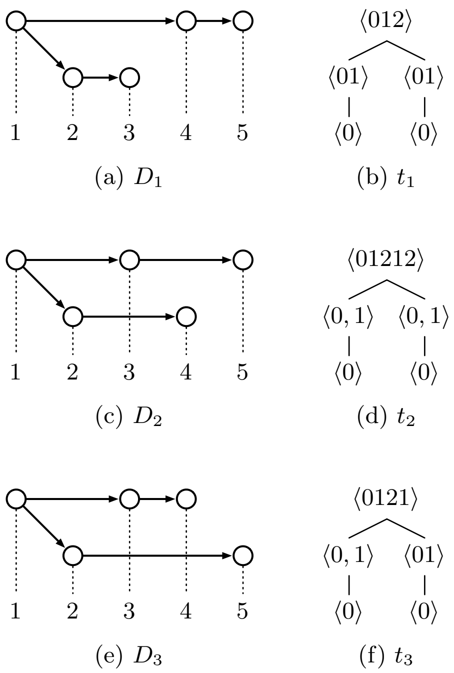
1.3.1 Dependency Structures

In the first part of the book, we study dependency structures. These structures clearly separate two relations: the dependency relation, which we call *governance*, and the total order on the nodes of the graph, which we call *precedence*. We discuss three interesting classes of mildly non-projective dependency structures, compare them to other classes in the literature, and evaluate their practical relevance using data from *dependency treebanks*.

Structural Constraints

The first two classes of dependency structures that we consider in this book have been studied before. *Projective dependency structures* (Chapter 3), as already mentioned, are characterized by the structural constraint that each subtree must form an interval with respect to the total order on the nodes. As an example, consider the dependency structure depicted in Figure 1.5a: each of the subtrees forms an interval with respect to the precedence relation. In *dependency structures of bounded degree* (Chapter 4), the projectivity constraint is relaxed in such a way that dependency subtrees can be distributed over more than one, but still a finite number of intervals. For example, in the structure depicted in Figure 1.5c, both the subtree rooted at the node 2 and the subtree rooted at the node 3 span two intervals. We call the maximal number of intervals per subtree the *block-degree* of the structure, and use it to quantify the non-projectivity of dependency graphs.

The third class of dependency structures that we investigate, the class of *well-nested dependency structures* (Chapter 5), is original to this work. Well-nestedness is the structural constraint that pairs of disjoint dependency subtrees must not cross, meaning that there must not be nodes i_1, i_2 in the first subtree and nodes j_1, j_2 in the second such that $i_1 < j_1 < i_2 < j_2$. The dependency structure depicted in Figure 1.5e is well-nested, while the structure depicted in Figure 1.5c is not. Well-nested dependency structures are closely related to several other combinatorial structures, such as non-crossing partitions and Dyck languages. We discuss an empirical evaluation that shows that they are also practically relevant: virtually all dependency analyses in two large and widely-used dependency treebanks obey the well-nestedness constraint.



(a) D_1

(b) t_1

(c) D_2

(d) t_2

(e) D_3

(f) t_3

Fig. 1.5. A zoo of dependency structures, and their corresponding terms

Algebraic Framework

As we already mentioned, one of the major contributions of this book is an algebraic framework in which projective, block-restricted and well-nested dependency structures can be understood as the outcome of compositional processes. Under this view, structural constraints do not apply to a fully specified dependency graph, but are inherent in the composition operations by which the graph is constructed. This provides a bridge between dependency structures and grammar formalisms. We formalize the algebraic framework in two steps. In the first step, we show that dependency structures can be encoded into terms over a certain signature of *order annotations* in such a way that the three different classes of dependency structures discussed above stand in one-to-one correspondence with terms over specific subsets of this signature. In the second step, we define the concept of a *dependency algebra*. In these algebras, order annotations are interpreted as composition operations on dependency structures. We prove that each dependency algebra is isomorphic to the corresponding term algebra, which means that the composition of dependency structures can be freely simulated by the usual composition operations on terms, such as substitution.

To give an intuition for the algebraic framework, the right half of Figure 1.5 shows the terms corresponding to the dependency structures in the left half. Each order annotation in these terms encodes node-specific information about the precedence relation. As an example, the symbol $\langle 0, 1 \rangle$ in Figure 1.5d represents the information that the corresponding subtree in Figure 1.5c consists of two intervals (the two components of the tuple $\langle 0, 1 \rangle$), with the root node (represented by the symbol 0) situated in the left interval, and the subtree rooted at the first child (represented by the symbol 1) in the right interval. Under this encoding, the block-degree measure corresponds to the maximal number of components per tuple, and the well-nestedness condition corresponds to the absence of certain 'forbidden substrings' in the individual order annotations, such as the substring 1212 in the term in Figure 1.5d.

Structures and Grammars

In Chapter 6, we apply the algebraic framework to classify the dependency structures induced by various lexicalized grammar formalisms. The key to this classification is the insight that the notion of induction can be formalized as the interpretation of the derivations of a grammar in a suitable dependency algebra. Based on this formalization, we can generalize Gaifman's [1965] result that projective dependency structures correspond to lexicalized context-free grammars into the realm of the mildly context-sensitive: the classes of block-restricted dependency structures correspond to Linear Context-Free Rewriting Systems [118, 119], the classes of well-nested block-restricted structures correspond to Coupled Context-Free Grammar [49]. As a special case, the class of well-nested dependency structures with a block-degree of at most 2 is characteristic for derivations in Lexicalized Tree Adjoining Grammar [54].

1.3.2 Dependency Languages

In the second part of the book, we lift our results from individual dependency structures to sets of such structures, or *dependency languages*. The key to this transfer is the formal concept of regular sets of dependency structures (Chapter 7), which we define as the recognizable subsets of dependency algebras [78]. From this definition, we obtain natural notions of automata and grammars on the basis of which we can reason about the language-theoretic properties of regular dependency languages.

Automata and Grammars

Given the isomorphism between dependency algebras and term algebras, we can derive a natural automaton model for dependency structures from the concept of a *tree automaton* [115]. This method in fact applies to all kinds of data structures that are constructible using a finite set of operations; for example, successful applications of the approach have previously led to stepwise tree automata for the data model of XML [8] and feature automata for unranked unordered trees [84]. From the notion of an automaton, we are led to the concept of a *regular dependency grammar*. By and large, grammars and automata are two sides of the same coin: we get a grammar from an automaton by interpreting the transition rules of the automaton as a directed rewriting system. Using regular dependency grammars, we show a powerful pumping lemma for regular dependency languages, and prove that these languages are semilinear [91], which is also characteristic for languages generated by mildly context-sensitive grammar formalisms.

String-Generative Capacity and Parsing Complexity

In the last technical chapter of the book (Chapter 8), we investigate the connections between structural constraints, string-generative capacity, and parsing complexity. We show how the block-degree measure gives rise to an infinite hierarchy of ever more powerful string languages, and how enforcing the well-nestedness of the underlying dependency structures leads to a proper decrease of string-generative power on nearly all levels of this hierarchy. In proving these results, we see how string languages can 'enforce' the presence of structures with certain properties in the corresponding dependency languages. As an example, for every natural number k, we identify a string language $L(k)$ that requires every regular set of dependency structures with block-degree at most k that projects $L(k)$ to contain structures that are not well-nested. Finally, we show that both the block-degree measure and the well-nestedness condition have direct implications for the parsing complexity of regular dependency languages. We prove that, while the parsing problem of regular dependency languages is polynomial in the length of the input string, the problem in which we take the grammar to be part of the input is NP-complete. Interestingly, for well-nested dependency languages, parsing is polynomial even with the size of the grammar taken into account.

1.3.3 Contributions

In summary, this book makes two main contributions:

1. an algebraic theory of mildly non-projective dependency structures and regular sets of such structures (dependency languages), and
2. a classification of mildly context-sensitive, lexicalized grammar formalisms in terms of the dependency structures that these formalisms induce.

The algebraic theory complements previous work on dependency representations in that it enables us to link structural constraints such as projectivity, block-degree and well-nestedness to language-theoretic properties such as string-generative capacity and parsing complexity. The classification of grammar formalisms in terms of their ability to induce dependency structures yields a new, practically useful measure of generative capacity. Both results provide fundamental insights into the relation between dependency structures and lexicalized grammars.

2

Preliminaries

This chapter provides a compact review of the basic terminology and notation that we will use in this book. It is thought more as a reference than as a tutorial presentation. The reader is invited to browse through these preliminaries and return to them whenever some notation or terminology is unclear.

Our formal toolbox is composed from four main sources: The terminology for mathematical structures and the relations between these structures come from universal algebra [16]. The specific formalization of dependency structures takes an order-theoretic perspective [15], but also alludes to graph theory [17]. To describe and manipulate structures and sets of structures, we make use of terms and term languages, and of the usual operations defined on them [28]. Note that we use the word 'term' for the syntactic object, and the word 'tree' when referring to the order-theoretic and graph-theoretic structures.

Basic Notations

We write $\mathbb{N}$ for the set of non-negative integers. For $n \in \mathbb{N}$, we write $[n]$ to refer to the set $\{\, m \in \mathbb{N} \mid 1 \leq m \leq n \,\}$. Note that by this definition, $[0] = \emptyset$. For a set A, we write $|A|$ for the cardinality of A.

We use the notations A^* and A^+ to refer to the sets of all and all non-empty strings over the set A, respectively. We treat strings as vectors: the notation a_i refers to the ith element of the string $\vec{a}$. The length of a string $\vec{a}$ is denoted by $|\vec{a}|$; the empty string is denoted by ε. The concatenation of two strings $\vec{x}$ and $\vec{y}$ is written as $\vec{x}\vec{y}$; only where this could create confusion, we use the alternative notation $\vec{x} \cdot \vec{y}$. An *alphabet* is a finite, non-empty set of *symbols*.

Indexed Sets and Sorted Sets

Let I be a non-empty set. An *I-indexed set* is a total function with domain I. We use the notation $\langle\, x_i \mid i \in I \,\rangle$ to refer to an I-indexed set, where x_i denotes

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 11–15, 2010.

the image of i. We freely identify the set of all indexed sets with index set $[n]$, $n \in \mathbb{N}$, with the set of all n-tuples, and with the set of all strings of length n. An *I-indexed family* is an I-indexed set with a set-valued codomain. For indexed families, the usual set-theoretic operation are defined index-wise. In particular, if A and B both are I-indexed families, then $A \times B = \langle\, A_i \times B_i \mid i \in I \,\rangle$.

Let $\mathcal{S}$ be a non-empty collection of *sorts*. An *$\mathcal{S}$-sorted set* consists of a non-empty set A and a *type assignment* $\mathrm{type}_A\colon A \to \mathcal{S}^+$. We write $\mathcal{S}_A$ for the collection of sorts that underlies A. When the sorted set under consideration is irrelevant or clear from the context, we write $a\colon \vec{s}$ instead of $\mathrm{type}_A(a) = \vec{s}$; this is to be read as 'a has type $\vec{s}$ in A'. The set of all elements of A with type $\vec{s}$ is denoted by $A_{\vec{s}}$. In the following, let A and B be sorted sets. We write $\langle A, B\rangle$ for the sorted set $A \times B$ in which $\langle a, b\rangle\colon \mathrm{type}_A(a)$, for all $a \in A$, $b \in B$. For an element $a\colon \vec{s}s$ with $\vec{s} \in \mathcal{S}^*$ and $s \in \mathcal{S}$, the length of $\vec{s}$ is called the *rank* of a, and is denoted by $\mathrm{rank}_A(a)$. If $|\mathcal{S}| = 1$, the type of an element $a \in A$ is uniquely determined by its rank; in this case, the set A is called a *ranked set*, and the set of all elements with rank k is denoted by A_k. We freely treat $\mathcal{S}$-indexed sets A as $\mathcal{S}$-sorted sets by stipulating that $\mathrm{type}_A(a) = s$ if and only if $a \in A_s$.

Structures

We now define the notion of a mathematical structure. Let Σ be a sorted set, now called a *signature*. A (concrete) *Σ-structure* is a pair

$$\mathfrak{A} \;=\; (\langle\, A_s^{\mathfrak{A}} \mid s \in \mathcal{S}_\Sigma \,\rangle, \langle\, R_\sigma^{\mathfrak{A}} \mid \sigma \in \Sigma \,\rangle)\,,$$

where the first component is an $\mathcal{S}_\Sigma$-indexed family of non-empty sets, called the *domains* of $\mathfrak{A}$, and the second component is a Σ-indexed family of relations over the domains such that $R_\sigma^{\mathfrak{A}} \subseteq A_{s_1}^{\mathfrak{A}} \times \cdots \times A_{s_n}^{\mathfrak{A}}$, for every symbol $\sigma\colon s_1 \cdots s_n$. We use the notation $\mathrm{dom}(\mathfrak{A})$ to refer to the domains of $\mathfrak{A}$. For structures with small signatures over a single sort, we use the compact notation $(A_s^{\mathfrak{A}}\,; R_1^{\mathfrak{A}}, \dots, R_n^{\mathfrak{A}})$, leaving the signature implicit. A structure is *finite*, if both its signature and all of its domains are finite sets.

Given two Σ-structures $\mathfrak{A}$ and $\mathfrak{B}$, a *homomorphism* from $\mathfrak{A}$ to $\mathfrak{B}$ is an indexed family $\langle\, h_s \mid s \in \mathcal{S}_\Sigma \,\rangle$ in which, for each given sort $s \in \mathcal{S}_\Sigma$, the object h_s is a total function $h_s\colon \mathrm{dom}(\mathfrak{A})_s \to \mathrm{dom}(\mathfrak{B})_s$ with the property that

$$(a_1, \dots, a_n) \in R_\sigma^{\mathfrak{A}} \quad\Longrightarrow\quad (h_s(a_1), \dots, h_s(a_n)) \in R_\sigma^{\mathfrak{B}}\,,$$

for every $\sigma\colon s_1 \cdots s_n$, $a_i \in \mathrm{dom}(\mathfrak{A})_{s_i}$, and $i \in [n]$. The notation $h : \mathfrak{A} \to \mathfrak{B}$ refers to a homomorphism between Σ-structures $\mathfrak{A}$ and $\mathfrak{B}$, treating it as a single mapping rather than as an indexed family of mappings. Furthermore, to avoid subscript clutter, for $s \in \mathcal{S}_\Sigma$ and $a \in \mathrm{dom}(\mathfrak{A})_s$, we write $h(a)$ rather than $h_s(a)$ for the image of a under the homomorphism h, assuming that h keeps type discipline. A homomorphism is called *epi* or an *epimorphism*, if every member function is injective; it is called *mono* or a *monomorphism*, if every member function is surjective; it is called *iso* or an *isomorphism*, if every member function is bijective. We do not distinguish between structures $\mathfrak{A}$ and $\mathfrak{B}$ for which there exists an isomorphism $h : \mathfrak{A} \to \mathfrak{B}$. Specifically, a *$\Sigma$-structure* is an equivalence class of concrete Σ-structures modulo isomorphism.

Ordered Sets

An *ordered set* is a structure with a single binary relation that is reflexive, transitive, and anti-symmetric. Let $\mathfrak{R} = (A\,; \preceq)$ be an ordered set, and let $a, b \in A$. We write $a \prec b$ to assert that $a \preceq b$, and $a \neq b$. We say that a *immediately precedes* b (with respect to $\preceq$) if $a \preceq b$ and there is no element $c \in A$ such that $a \preceq c \preceq b$. The *up-closure* and the *down-closure* of a are defined as $\lfloor a \rfloor := \{\, c \in A \mid a \preceq c \,\}$ and $\lceil a \rceil := \{\, c \in A \mid c \preceq a \,\}$, respectively. For a given subset $B \subseteq A$, we say that $\mathfrak{R}$ is *total* on B, if $a \preceq b$ or $b \preceq a$ holds for all $a, b \in B$. The structure $\mathfrak{R}$ is called a *chain*, if it is total on A; it is called a *forest*, if it is total on all down-closures; it is called a *tree*, if it is a forest and additionally contains an element r, the *root node* of $\mathfrak{R}$, with the property that $\lfloor r \rfloor = A$.

Observe that what we call 'forests' and 'trees' are the reflexive-transitive closures of the corresponding objects from graph theory. The elements of the domains of trees are called *nodes*. We use the symbols u, v and w for variables that range over nodes. Let $T = (V\,; \trianglelefteq)$ be a tree. If $u \trianglelefteq v$, we say that u *dominates* v. We write $u \to v$ if u immediately precedes v with respect to dominance; this relation corresponds to the edge relation in the formalization of trees as special directed graphs. We use the standard genealogical terminology to refer to relations between nodes in a tree: If $u \to v$, then we say that v is a *child* of u, and, symmetrically, that u is the *parent* of v. Distinct children of the same node are called *siblings*. We use the term *yield* as a synonym for the down-closure of u; notice that $u \trianglelefteq v$ if and only if $v \in \lfloor u \rfloor$. The set of *descendants* and *ancestors* of u are defined as $\lceil u \rceil - \{u\}$ and $\lfloor u \rfloor - \{u\}$, respectively. Two nodes u and v are *disjoint*, if $\lfloor u \rfloor \cap \lfloor v \rfloor = \emptyset$. Each pair v, w of disjoint nodes has a greatest common ancestor u; for this situation, we write $v \perp_u w$.

For chains, we define the notion of an *interval*: the interval with endpoints a and b is the set $[a, b] := (\lfloor a \rfloor \cap \lceil b \rceil) \cup (\lceil a \rceil \cap \lfloor b \rfloor)$. We also put $(a, b) := [a, b] - \{a, b\}$. A set is *convex*, if it is an interval.

Dependency Structures

A *dependency structure* is a structure D with two binary relations: one relation forms a tree, the second relation forms a chain on the domain of D. Thus, dependency structures are trees with a total order on their nodes. They differ from *ordered trees*, where the order relation is only defined on the children of each node, but not on the set of all nodes of the tree.

The tree relation of a dependency structure is called *governance*, the total order is called *precedence*. Just like in trees, the elements of the domains of dependency structures are called *nodes*. Given a dependency structure D, for nodes $u, v \in \mathrm{dom}(D)$, we write $u \trianglelefteq v$ to assert that u *governs* v, and $u \preceq v$ to assert that u *precedes* v in D. To talk about dependency structures, we import all terminology for trees and chains.

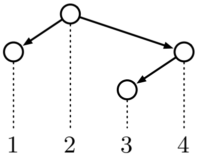


Fig. 2.1. A dependency structure

Figure 2.1 shows how we visualize dependency structures. A picture of a dependency structure contains the nodes of the structure (drawn as circles), *edges* (drawn as pointed arrows), and *projection lines* (drawn as dotted lines). Specifically, we draw an edge between two nodes u and v if and only if $u \to v$. The nodes are ordered from left-to-right; we place u before v if and only if $u \prec v$. The projection lines are used to make the left-to-right order more explicit.

When discussing algorithms that operate on dependency structures, we assume a concrete representation of these structures as a collection of objects, where each object u has access to at least the object representing its parent node, $\mathit{parent}[u]$, and its position in the precedence order among all the nodes of the structure, $\mathit{pos}[u]$. We also make the (reasonable) assumption that both attributes can be accessed in constant time. With this representation mind, it is straightforward that the following auxiliary mappings can be constructed in time linear in the size of the structure: a mapping $\mathit{children}[u]$ that maps each object u to the set of objects representing the children of u; a mapping $\mathit{node}[i]$ that maps each position i (where i ranges over the size of the structure) to the object u for which $\mathit{pos}[u] = i$. Similarly, it is straightforward that the following iterations over the nodes in the structure can be supported in linear time: pre-order, post-order, and left-to-right.

Algebraic Structures and Terms

A Σ-structure $\mathfrak{A} = (\langle A_s^{\mathfrak{A}} \mid s \in \mathcal{S}_\Sigma \rangle, \langle f_\sigma^{\mathfrak{A}} \mid \sigma \in \Sigma \rangle)$ is called *algebraic* or an *algebra*, if for every symbol $\sigma\colon s_1 \cdots s_m s$, the relation $f_\sigma^{\mathfrak{A}}$ is a total function, meaning that for every $i \in [m]$ and $a_i \in A_{s_i}^{\mathfrak{A}}$, there is exactly one $a \in A_s^{\mathfrak{A}}$ such that $(a_1, \dots, a_m, a) \in f_\sigma^{\mathfrak{A}}$. In the context of algebras, we use the notation $\sigma\colon s_1 \times \cdots \times s_m \to s$ instead of $\sigma\colon s_1 \cdots s_m s$, and call m the *arity* of the symbol σ and the corresponding function f_σ.

Let Σ be a sorted set. The set of *terms* over Σ is the smallest $\mathcal{S}_\Sigma$-indexed family T_Σ such that if $\sigma\colon s_1 \times \cdots \times s_m \to s$ and $t_i \in T_{\Sigma,s_i}$ for all $i \in [m]$, then $\sigma(t_1, \dots, t_m) \in T_{\Sigma,s}$. Let $t \in T_\Sigma$ be a term. The set of *nodes* of t, $\mathrm{nod}(t)$, is the subset of $\mathbb{N}^*$ that is defined by the equation

$$\mathrm{nod}(\sigma(t_1, \dots, t_m)) \;\; := \;\; \{\varepsilon\} \cup \{\, i \cdot u \mid i \in [m],\, u \in \mathrm{nod}(t_i) \,\}\,.$$

The empty string ε represents the root node of t, and the string $i \cdot u$ represents the ith child of the node u. The *subterm* of t at node u is denoted by t/u, the *substitution* of the term s at node u in t is denoted by $t[u \leftarrow s]$, and the *label* of t at node u is denoted by $t(u)$. We also put $\mathrm{alph}(t) := \{\, t(u) \mid u \in \mathrm{nod}(t) \,\}$, which thereby denotes the set of all labels in t. A *context* over Σ is a pair (t, u), where $t \in T_\Sigma$ is a term, and u is a leaf node in t. We write C_Σ for the set of all contexts over Σ, and make free use of all term-related concepts even for contexts. Given a context $(c, u) \in C_\Sigma$ and a term $t \in T_{\Sigma,s}$ with $s = \mathrm{type}_\Sigma(c(u))$, we write $c \cdot t$ for the term obtained as $c[u \leftarrow t]$.

The *term algebra* over Σ is the algebra $\mathfrak{A}$ in which $\mathrm{dom}(\mathfrak{A}) = T_\Sigma$, and in which each function $f_\sigma^{\mathfrak{A}}$ is interpreted as a term constructor in the obvious way. We use the notation T_Σ for both the set of terms over Σ and the term algebra over Σ. For every signature Σ and every Σ-algebra $\mathfrak{A}$, there is a uniquely determined homomorphism $[\![\cdot]\!]_{\mathfrak{A}}\colon T_\Sigma \to \mathfrak{A}$ that evaluates terms in T_Σ as values in $\mathfrak{A}$.

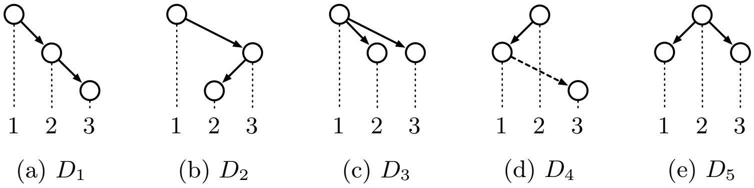
3

Projective Dependency Structures

The exact relation between dependency and word order is a major point of debate in dependency theory. Over the years, various authors have made proposals for formal constraints to restrict this relation. In this chapter, we study the best-known of these proposals, *projectivity.* We start by reviewing three standard characterizations of projectivity (Section 3.1), and then introduce a new, algebraic characterization (Section 3.2). This gives rise to an efficient algorithm to test whether a given dependency structure is projective (Section 3.3). We use this algorithm to evaluate the practical relevance of projectivity on data from three dependency treebanks (Section 3.4).

3.1 Projectivity

Figure 3.1 shows pictures of five dependency structures. One of these, Figure 3.1d, is different from the others in that it displays *crossing edges*—the edge 1 → 3 crosses the projection line of node 2. The projectivity condition is often summarized in the slogan that 'it disallows dependency structures with pictures that contain crossing edges'. This is a nice mnemonic, but whether a dependency edge crosses a projection line or not of course mainly is a matter of how we *draw* dependency structures, not a property of the structures themselves. For example, Figure 3.1d can be re-drawn without crossing edges

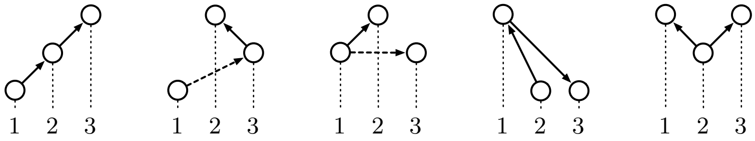


(a) D_1 (b) D_2 (c) D_3 (d) D_4 (e) D_5

Fig. 3.1. Five (of nine) dependency structures with three nodes

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 17–32, 2010.

if node 2 is moved to a vertical position below node 1 (see Figure 3.2d), while Figure 3.1b and Figure 3.1c exhibit crossing edges when modified such that node 2 is positioned above the root node (see Figure 3.2b and Figure 3.2c). It is clear then that to obtain a precise characterization of projectivity, we need a definition that formalizes the idea of crossing edges without referring to differences in visualization.



(a) D_1 (b) D_2 (c) D_3 (d) D_4 (e) D_5

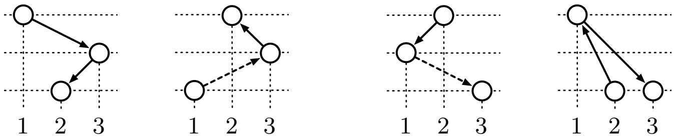
Fig. 3.2. Alternative pictures for the dependency structures from Figure 3.1

3.1.1 Projectivity in the Sense of Harper and Hays

A crucial difference between the pictures in Figure 3.1 and the alternative versions in Figure 3.2 is that in the former, all tree edges point downwards, while in the latter, some of them also point upwards. Let us call a picture of a dependency structure *canonical*, if the vertical position of each node is chosen according to its level of depth in the tree, with the root node taking the highest position.

Example 3.1.1. Figure 3.3 shows canonical and non-canonical pictures of the dependency structures D_2 and D_4. The horizontal lines visualize the depth levels. □

In canonical pictures of dependency structures, all tree edges point to the next level of depth. As a consequence, an edge $u \to v$ can cross the projection line of a node w only if the vertical position of u is the same as or below the vertical position of w. To ban crossing edges in canonical pictures, it is sufficient then to require the node u to govern the node w; this guarantees that the vertical position of u is strictly above the vertical position of w. The requirement is



(a) D_2 (b) D_4

Fig. 3.3. Canonical and non-canonical pictures of two dependency structures

made formal in the following implication, attributed to Kenneth Harper and David Hays [72]:

$$u \to v \ \wedge \ w \in (u,v) \implies u \trianglelefteq w \tag{3.1}$$

We introduce some useful terminology: Let u, v, w be witnesses for the premises of the implication (3.1). We then say that the edge $u \to v$ *covers* the node w, and call an edge that covers a node but does not govern it a *non-projective edge*. A dependency structure is projective if and only if it does not contain non-projective edges.

Example 3.1.2. The edge $1 \to 3$ in Figure 3.1d and Figure 3.2d is non-projective, as it covers the node 2, but does not govern it. All other edges depicted in Figure 3.1 and Figure 3.2 are projective. □

3.1.2 Projectivity in the Sense of Lecerf and Ihm

The characterization of projectivity in the sense of Harper and Hays links projectivity to edges. The second characterization that we consider, attributed to Yves Lecerf and Peter Ihm [72], anchors projectivity at *paths*:

$$u \trianglelefteq v \ \wedge \ w \in (u,v) \implies u \trianglelefteq w \tag{3.2}$$

Note that the only difference between this requirement and (3.1) is the first premise of the implication: projectivity in the sense of Lecerf and Ihm requires not only every edge, but every (directed) path from a node u to a node v to cover only nodes w that are governed by u. Since every path consists of a finite sequence of edges, the characterizations of projectivity in the sense of Harper and Hays and in the sense of Lecerf and Ihm are fully equivalent [72, chapter 6, Theorem 10].

3.1.3 Projectivity in the Sense of Fitialov

We now present a third characterization of projectivity. This characterization formalizes the observation that in a projective dependency analysis, a word and its (transitive) dependents form a contiguous substring of the full sentence. It is usually attributed to Sergey Fitialov [72].

$$u \trianglelefteq v_1 \ \wedge \ u \trianglelefteq v_2 \ \wedge \ w \in (v_1, v_2) \implies u \trianglelefteq w \tag{3.3}$$

This condition is equivalent to the preceding two [72, chapter 6, Theorem 11]. Using our terminology for chains, we can rephrase it more succinctly as follows:

Definition 3.1.1. A dependency structure D is called *projective*, if every yield in D is convex with respect to precedence. □

This is the formulation that we adopt as our formal definition of projectivity. We write $\mathcal{D}_1$ to refer to the class of all projective dependency structures.

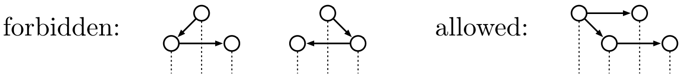
Example 3.1.3 (continued). In the dependency structure shown in Figure 3.1d, the yield of the node 1 (the set $\{1, 3\}$) does not form a convex set: the node 2 is missing from it. All other yields in Figure 3.1 are convex. Therefore, the only non-projective dependency structure in Figure 3.1 is structure D_4. □

3.1.4 Related Work

While the fundamental intuitions behind projectivity are already inherent in work on machine translation from the 1950s, the characterizations in the sense of Harper and Hays and in the sense of Lecerf and Ihm appear to be the first formal definitions of the condition; they were both published in 1960. Marcus [72] collects and compares several definitions of projectivity that circulated in the second half of the 1960s. In particular, he proves the equivalence of the characterizations of projectivity in the senses of Harper and Hays, Lecerf and Ihm, and Fitialov.

There are several other equivalent characterizations of projectivity; we only name two here. The following formulation is due to Robinson [100], p. 260; it is sometimes referred to as the 'adjacency principle', a term that appears to have been coined by Hudson [50], p. 98: 'If A depends directly on B and some element C intervenes between them (in linear order of string), then C depends directly on A or on B or on some other intervening element.' Havelka [43] presents an original edge-centric characterization of projectivity based on the difference between the depth of the dependent node of a dependency edge and the material covered by it, and uses it as the basis for an algorithm to test whether a given dependency structure is projective.

Veselá et al. [117] propose a characterization of projectivity based on 'forbidden elementary configurations' in a dependency structure, but the condition that they define still allows some non-projective structures:



The characterization of projectivity in terms of convex yields sheds some light on the relation between dependency grammar and phrase-structure grammar: If one accepts that yields reconstruct the notion of constituents that is familiar from phrase-structure grammars, then the projectivity condition amounts to the standard requirement that a constituent should be contiguous. In this sense, projective dependency structures are closely related to standard phrase-structure trees. This correspondence was first investigated by Hays [46]. The survey by Dikovsky and Modina [18], section 3.2 summarizes some of the formal results obtained since then.

3.2 Algebraic Framework

We have characterized projectivity as a relational property of dependency structures. The immediate value of this characterization is that it is empirically transparent: from a canonical picture of a given dependency structure, we can immediately *see* whether it is projective.

What is not clear at this point is how the projectivity constraint can be fitted into a grammatical framework, where a dependency structure is not given directly, but specified as the outcome of a derivational process. In this section, we clarify this issue: We equip the class of projective dependency structures with a set of algebraic operations. Each application of an operation can be understood as the application of a single production of some grammar. Our setup guarantees that all operations only yield projective structures, and that all projective structures can be decomposed into elementary operations. In this way, every projective dependency structure can be understood as the outcome of a complete derivation in a grammar with a suitable set of productions.

Table 3.1. Pre-order and post-order traversal of a children-ordered tree

```
Pre-Order-Collect($u$)
$L \leftarrow$ nil
$L \leftarrow L \cdot [u]$
foreach $v$ in $children[u]$
    do $L \leftarrow L \cdot$ Pre-Order-Collect($v$)
return $L$
```

```
Post-Order-Collect($u$)
$L \leftarrow$ nil
foreach $v$ in $children[u]$
    do $L \leftarrow L \cdot$ Post-Order-Collect($v$)
$L \leftarrow L \cdot [u]$
return $L$
```

3.2.1 Tree Traversal Strategies

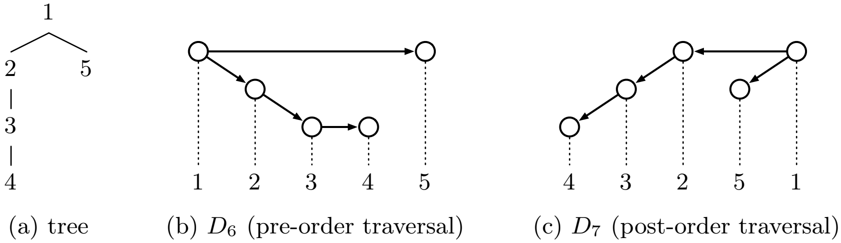
To convey the basic intuitions behind our algebraic setting, we start this section by looking at *tree traversal strategies*. A *tree traversal* is the process of systematically visiting all nodes of a tree. Two well-known strategies for tree traversal are *pre-order traversal* and *post-order traversal* of children-ordered trees. For the sake of concreteness, let us assume that a children-ordered tree is represented as a collection of nodes, where each node u is annotated with a list *children*$[u]$ of its children. We can then specify procedures to collect the nodes of a tree as in Table 3.1.[1] The result of a call to Pre-Order-Collect(u) or Post-Order-Collect(u) is a list of the nodes in the tree rooted at the node u; each node of the tree occurs in this list exactly once.

[1] The format of our pseudo-code follows Cormen et al. [11]. We write $[x]$ for the singleton list that contains the element x, and $L_1 \cdot L_2$ for the concatenation of two lists L_1 and L_2.

Both Pre-Order-Collect and Post-Order-Collect extend the orders among the children of each node into a global order, defined on all nodes of the tree. When we impose this global order on the nodes of the tree that was traversed, we obtain a dependency structure. This observation is formalized in the following definitions.

Definition 3.2.1. Let T be a tree. A *linearization* of T is a list of nodes of T in which each node occurs exactly once. The dependency structure *induced* by a linearization $\vec{u}$ of T is the structure in which the governance relation is isomorphic to T, and the precedence relation is isomorphic to $\vec{u}$. □

Example 3.2.1. Figure 3.4a shows a children-ordered tree. The pre-order linearization of this tree yields the node sequence 12345. When we impose this order on the nodes in the tree, we obtain the dependency structure D_6 shown in Figure 3.4b. In contrast, the post-order linearization of the children-ordered tree yields the node sequence 43251; this induces the dependency structure D_7 shown in Figure 3.4c. □



(a) tree (b) D_6 (pre-order traversal) (c) D_7 (post-order traversal)

Fig. 3.4. Dependency structures obtained by tree traversals of a children-ordered tree

We now sketch our plan for the remainder of this section: Our first goal is to find a traversal strategy and a class of order-annotated trees that fully characterize the class of projective dependency structures—traversals should only induce projective structures, and every projective structure should be inducible by some traversal. In a second step, we formalize this framework by regarding the set of all order annotations as an algebraic signature, order-annotated trees as terms over this signature, and tree traversal as the evaluation of these terms in an algebra over dependency structures.

With this roadmap in mind, let us see how far pre-order and post-order traversal take us. One property that both strategies have in common is that they visit the nodes in each subtree as a contiguous block. As a consequence, every dependency structure that is induced by pre-order or post-order traversal is projective. On the other hand, not every projective dependency structure can be obtained as the pre-order or post-order interpretation of a children-ordered tree. Specifically, all structures induced by pre-order traversal are *monotonic* in the sense that $u \trianglelefteq v$ implies that $u \preceq v$, while all structures obtained by post-order traversal are *anti-monotonic*. The fundamental reason for these specific properties is that in both pre-order and post-order traversal, the position of a node relative to its children is hard-wired: it is not specified in the order annotations, but in the traversal strategy itself.

3.2.2 Traversal of Treelet-Ordered Trees

To overcome the restrictions of pre-order and post-order traversal, we include the position of a node relative to its children in the order annotations, and make the traversal strategy sensitive to this information. Let us call the local tree formed by a node u and its children (if there are any) the *treelet* rooted at u, and let us say that a tree is *treelet-ordered*, if each of its nodes is annotated with a total order on the nodes in the treelet rooted at that node. Table 3.2 gives the pseudo-code of a procedure that traverses a treelet-ordered tree and returns a list of its nodes. We assume that each node u in the tree is annotated with a list $\mathit{order}[u]$ that contains the nodes in the treelet rooted at u in the intended order.

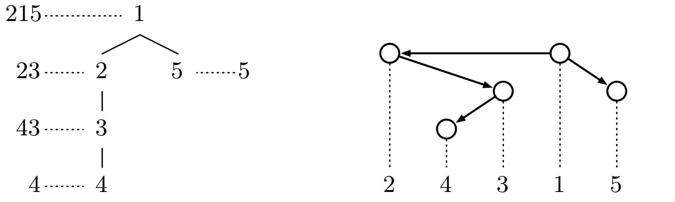
Table 3.2. Traversal of a treelet-ordered tree

```
TREELET-ORDER-COLLECT($u$)
$L \leftarrow$ NIL
foreach $v$ in $\mathit{order}[u]$
    do if $v = u$
        then $L \leftarrow L \cdot [u]$
        else $L \leftarrow L \cdot$ TREELET-ORDER-COLLECT($v$)
return $L$
```

Example 3.2.2. Figure 3.5a visualizes a treelet-ordered tree; the sequences at the ends of the dotted lines represent the annotated orders. The traversal of this tree according to the procedure in Table 3.2 yields the sequence 24315. When we impose this order on the nodes in the tree, we obtain the dependency structure D_8 shown in Figure 3.5b. Note that this structure is neither monotonic nor anti-monotonic. □

We now show that treelet-ordered trees and our procedure for their traversal are expressive enough to fully characterize the class of projective dependency structures. What is more, distinct treelet-ordered trees induce distinct structures. In our proofs, we use two functions on treelet-ordered trees: a function lin that maps each tree to its linearization according to TREELET-ORDER-COLLECT, and a function dep that maps each tree T to the dependency structure induced by $\mathrm{lin}(T)$. We then have

$$\begin{array}{lll} u \trianglelefteq v \text{ in } \mathrm{dep}(T) & \text{if and only if} & u \text{ dominates } v \text{ in } T\text{, and} \\ u \preceq v \text{ in } \mathrm{dep}(T) & \text{if and only if} & u \text{ precedes } v \text{ in } \mathrm{lin}(T). \end{array}$$



(a) tree

(b) D_8 (treelet-order traversal)

Fig. 3.5. A treelet-ordered tree and its corresponding dependency structure

The next three lemmata show that the function dep forms a bijection between the set of all treelet-ordered trees and the set of all projective dependency structures.

Lemma 3.2.1. *Let T be a treelet-ordered tree. Then dep(T) is projective.* □

Proof. Let T be a treelet-ordered tree, and let u be the root node of T. We show that every yield of T is convex with respect to the total order on the nodes of T that is represented by the linearization $\mathrm{lin}(T)$. Our proof proceeds by induction on the depth d of T.

First, assume that $d = 0$. In this case, the node u is the only node in T, and we have $\mathit{order}[u] = [u]$. Therefore, $\mathrm{lin}(T) = [u]$, and $\mathrm{dep}(T)$ is the trivial dependency structure. The yield $\lfloor u \rfloor$, like all singleton sets, is convex with respect to $\mathrm{lin}(T)$.

Now, assume that $d > 0$. In this case, the tree T can be decomposed into the node u and the collection of subtrees rooted at the children of u. Let $w \neq u$ be a node in T, and let v be the uniquely determined child of u that dominates w. By the induction hypothesis, we may assume that the yield $\lfloor w \rfloor$ is convex with respect to the linearization that was computed by the recursive call TREELET-ORDER-COLLECT(v). The result $\mathrm{lin}(T)$ of the call TREELET-ORDER-COLLECT(u) is a concatenation of these linearizations and the singleton list $[u]$; thus, the yield $\lfloor w \rfloor$ is convex even with respect to $\mathrm{lin}(T)$. The yield $\lfloor u \rfloor$, being the set of all nodes in T, is trivially convex with respect to $\mathrm{lin}(T)$. ■

For the next lemma, we introduce an important auxiliary concept.

Definition 3.2.2. Let D be a dependency structure, and let u be a node in D. The set of *constituents* of u is defined as $\mathcal{C}(u) := \{\{u\}\} \cup \{\, \lfloor v \rfloor \mid u \to v \,\}$. □

Lemma 3.2.2. *For every projective dependency structure D, there is a treelet-ordered tree T such that $D = dep(T)$.* □

Proof. Let D be a projective dependency structure with root node u. We show how to construct a treelet-ordered tree T such that $D = \mathrm{lin}(T)$. The tree structure underlying T is isomorphic to the tree structure underlying D.

Therefore, it suffices to show that we can find appropriate order annotations for the nodes in T such that $\mathrm{lin}(T)$ corresponds to the precedence relation in D. We proceed by induction on the depth d of the tree structure underlying D.

First, assume that $d = 0$. In this case, we have only one choice to assign an order annotation to u, $\mathit{order}[u] = [u]$. With this annotation, we indeed have $\mathrm{lin}(T) = [u]$.

Now, assume that $d > 0$. In this case, the node u has an out-degree of $n > 0$. The set $\mathcal{C}(u)$ of constituents of u forms a partition of the yield $\lfloor u \rfloor$. Furthermore, every constituent is convex with respect to the order underlying D: the set $\{u\}$ trivially so, and each set $\lfloor v \rfloor$ because the structure D is projective. We can also verify that for every constituent $C \in \mathcal{C}(u)$, the restriction $D|_C$ forms a projective dependency structure on C. Thus, by the induction hypothesis, we may assume that for every child v of u, we can annotate the subtree T/v such that $\mathrm{dep}(T/v) = D|_{\lfloor v \rfloor}$. What remains to be shown is that we can annotate u such that the call Treelet-Order-Collect(u) arranges the constituents $\mathcal{C}(u)$ according to their relative precedence in D.

To construct the order annotation for the node u, let $\pi \colon \lfloor u \rfloor \to \lfloor u \rfloor$ be the function that maps u to itself and every other node $v \in \lfloor u \rfloor$ to the uniquely determined child of u that governs v. Now, let L be the list of all nodes in D in the order of their precedence, and let L' be the list obtained from L by replacing each node w with the node $\pi(w)$ if $w \in \lfloor u \rfloor$, and with the symbol $\square$ if $w \notin \lfloor u \rfloor$. Finally, let $\mathit{order}[u]$ be the list obtained from L' by collapsing all adjacent occurrences of the same symbol into a single occurrence, and removing all leading and trailing symbols $\square$. For this order annotation of u, we can verify that the call Treelet-Order-Traversal(u) returns the constituents $\mathcal{C}(u)$ in the order that they have in D. ■

Example 3.2.3 (continued). For the dependency structure D_7 shown in Figure 3.5b, the construction described in the proof yields

$$\begin{array}{llll}
\text{for node 1:} & L = 24315 & L' = 22215 & \mathit{order}[1] = 215\,, \\
\text{for node 2:} & L = 24315 & L' = 233\,\square\square & \mathit{order}[2] = 23\,, \\
\text{for node 3:} & L = 24315 & L' = \square\,43\,\square\square & \mathit{order}[3] = 43\,, \\
\text{for node 4:} & L = 24315 & L' = \square\,4\,\square\square\square & \mathit{order}[4] = 4\,, \\
\text{for node 5:} & L = 24315 & L' = \square\square\square\square\,5 & \mathit{order}[5] = 5\,.
\end{array}$$

Note that these are the order annotations shown in Figure 3.5a. □

Lemma 3.2.3. *For every projective dependency structure D, there is at most one treelet-ordered tree T such that $\mathrm{dep}(T) = D$.* □

Proof. Let D be a projective dependency structure, and let T be a treelet-ordered tree such that $\mathrm{dep}(T) = D$. Now let T' be another treelet-ordered tree, distinct from T, and consider the dependency structure $\mathrm{dep}(T')$. We distinguish two cases: If the tree structures underlying T and T' are non-isomorphic, then $\mathrm{dep}(T)$ and $\mathrm{dep}(T')$ are non-isomorphic as well. Otherwise, the tree T' differs from T with respect to some order annotation. Then the call to Treelet-Order-Collect returns a different order for T than for T'; hence, $\operatorname{dep}(T)$ and $\operatorname{dep}(T')$ are non-isomorphic. ∎

3.2.3 Order Annotations

We now translate our framework into the language of terms: we regard the list-based order annotations that we used in Treelet-Order-Collect as a ranked set Ω, and treelet-ordered trees as terms over this set. This allows us to reinterpret the function dep as a bijection between T_Ω and the class of projective dependency structures.

While our list-based order annotations were sequences of (pointers to) concrete nodes in a treelet-ordered tree, the ranked set Ω should be defined independently of any particular term over this set. Therefore, we add a layer of indirection: each order annotation in Ω refers to nodes not directly, but by *names* for these nodes; these names are then resolved given the term structure. Specifically, let T be a treelet-ordered tree with root node u. We need two auxiliary sequences: the vector $\vec{v} = v_1 \cdots v_m$ obtained from $\mathit{order}[u]$ by removing the node u, and the string $\vec{\imath}$ obtained from $\mathit{order}[u]$ by replacing every child of u by its position in $\vec{v}$, and u itself by the symbol 0. The vector $\vec{v}$ orders the children of u; this order will become the left-to-right order on the children of u in our term representation. The string $\vec{\imath}$ provides an ‘abstract’ order annotation that makes use of node names rather than concrete nodes: the name 0 denotes the root node of the treelet rooted at u, a name $i \in [m]$ denotes the ith node in the sequence $\vec{v}$. The *term* $t(T)$ corresponding to T is then defined recursively as

$$t(T) \quad := \quad \langle\vec{\imath}\rangle(t(T/v_1), \ldots, t(T/v_m))\,.$$

In this definition, the string $\langle\vec{\imath}\rangle$ is understood as a term constructor of rank m. We write Ω_m for the set of all such constructors, and put $\Omega := \bigcup_{m \in \mathbb{N}} \Omega_m$. Every term over Ω encodes a treelet-ordered tree in the way that we have just described, and every such tree can be encoded into a term. In this way, we can view the function dep as a bijection $\operatorname{dep}\colon T_\Omega \to \mathcal{D}_1$ in the obvious way. We put $\operatorname{term} := \operatorname{dep}^{-1}$.

Example 3.2.4. Figure 3.6 shows the term for the treelet-ordered tree from Figure 3.5a. □

3.2.4 Dependency Algebras

Using the ranked set Ω and the bijection $\operatorname{dep}\colon T_\Omega \to \mathcal{D}_1$ between terms over Ω and projective dependency structures, we now give the set $\mathcal{D}_1$ an algebraic structure: with every order annotation, we associate an operation on projective dependency structures.

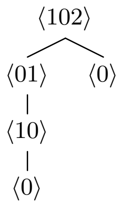


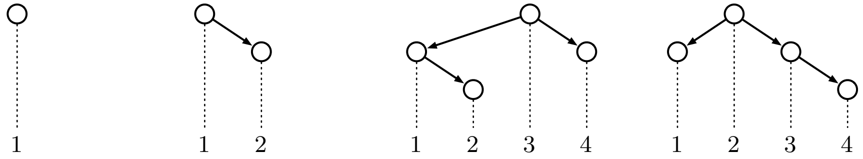
Fig. 3.6. The term for the treelet-ordered tree from Figure 3.5a

Definition 3.2.3. Let $m \in \mathbb{N}$, and let $\omega \in \Omega_m$ be an order annotation. The *composition operation* corresponding to ω is the function $f_\omega \colon \mathcal{D}_1^m \to \mathcal{D}_1$ defined as

$$f_\omega(D_1, \dots, D_m) \;:=\; \mathrm{dep}(\omega(\mathrm{term}(D_1), \dots, \mathrm{term}(D_m)))\,. \qquad \square$$

Composition operations are well-defined because the function dep is bijective. Each composition operation f_ω simulates a single step of the treelet-order traversal: given a sequence of argument structures, it returns the dependency structure that is obtained by taking the disjoint union of the arguments, adding a new root node, and arranging the nodes of the arguments and the root node in the order specified by ω.

Example 3.2.5. Figure 3.7 shows some examples for the results of composition operations. The composition of zero arguments, $f_{\langle 0\rangle}$, is the trivial dependency structure with one node (Figure 3.7a). Starting from this structure, more and more complex dependency structures can be built (Figures 3.7b–3.7d). □



(a) D_0 (b) $D_9 = f_{\langle 01\rangle}(D_0)$ (c) $f_{\langle 102\rangle}(D_9, D_0)$ (d) $f_{\langle 102\rangle}(D_0, D_9)$

Fig. 3.7. Examples for composition operations

We now have everything we need to define our algebraic setting. In the following definition, we use the function dep lifted to sets in the obvious way.

Definition 3.2.4. Let $\Sigma \subseteq \Omega$ be a finite set of order annotations. The *dependency algebra* over Σ is the Σ-algebra that has $\mathrm{dep}(T_\Sigma)$ as its carrier set, and interprets each symbol $\omega \in \Sigma$ by the composition operation corresponding to ω. □

By definition, dependency algebras are isomorphic to term algebras:

Theorem 3.2.1. *Let $\Sigma \subseteq \Omega$ be a finite set of order annotations. Then the dependency algebra over Σ is isomorphic to the term algebra over Σ, T_Σ.* □

Proof. Let $\mathfrak{D}$ be the dependency algebra over Σ. The restriction h of dep to the set of all terms over Σ is a bijection between T_Σ and the set $\mathrm{dep}(T_\Sigma)$, the carrier of $\mathfrak{D}$. Furthermore, from the definition of the composition operations we see that h forms a Σ-homomorphism between the term algebra T_Σ and $\mathfrak{D}$:

$$h(\omega(t_1,\dots,t_m)) = h(\omega(h^{-1}(h(t_1)),\dots,h^{-1}(h(t_m)))) = f_\omega(h(t_1),\dots,h(t_m)) .$$

Hence, T_Σ and $\mathfrak{D}$ are isomorphic. ■

One convenient consequence of the isomorphism between dependency algebras and their corresponding term algebras is that we can make use of all the terminology and notations available for terms when reasoning about dependency structures.

3.3 Algorithmic Problems

We now address three algorithmic problems associated with projectivity: the problems of encoding a projective dependency structure into its corresponding term, the symmetric problem of decoding a term into a dependency structure, and the problem of deciding whether a given dependency structure is projective.

3.3.1 Encoding and Decoding

The encoding problem for projective dependency structures is to compute, for a given dependency structure D, the term $\mathrm{term}(D)$. Since the tree relation of a dependency structure and its corresponding term are isomorphic, the crucial task when encoding a projective structure into a term is to extract the order annotations for the nodes of the structure. A naïve procedure to solve this task is inherent in our proof of the result that the function dep is onto (page 25). This procedure can be implemented to run in time $O(n^2)$, where n is the number of nodes in D. Each order annotation reflects the restriction of the precedence relation to the nodes in the treelet rooted at u. Consequently, each list *order*$[u]$ contains the nodes in the treelet rooted at u in the order in which they appear in D. It is not hard to see that we can populate all of these lists in a single iteration over the nodes in the order of their precedence. Pseudocode for the procedure is given in Table 3.3. Assuming that all elementary operations on D take constant time, and that an iteration takes time $O(n)$, extracting the order annotations and hence encoding can be done in time $O(n)$ as well.

Table 3.3. Extracting the order annotations for a projective structure

```
Extract-Order-Annotations($D$)
foreach $u$ in $D$
      do $order[u] \leftarrow$ NIL
foreach $u$ in $D$                      ▷ in the order of their precedence in $D$
      do if $parent[u] \neq$ UNDEFINED   ▷ $u$ is an inner node
            then $order[parent[u]] \leftarrow order[parent[u]] \cdot [u]$
         $order[u] \leftarrow order[u] \cdot [u]$
```

Lemma 3.3.1. *Let D be a dependency structure with n nodes. Then the term $term(D)$ can be computed in time $O(n)$.* □

The problem of decoding a term $t \in T_{\Omega}$ into its corresponding dependency structure is solved by the tree-traversal procedure that we gave in Table 3.2. Assuming that all elementary operations of that procedure take constant time, it is clear that the full precedence relation can be constructed in time linear in the size of the input term.

Lemma 3.3.2. *Let $t \in T_{\Omega}$ be a term with n nodes. Then the projective dependency structure $dep(t)$ can be computed in time $O(n)$.* □

3.3.2 Testing whether a Dependency Structure Is Projective

The isomorphism between projective dependency structures and terms over Ω gives rise to a simple and efficient algorithm for testing whether a given structure is projective. Note that nothing in our encoding procedure hinges on the input structure being projective. At the same time, only for projective structures this encoding produces terms that can be decoded back into the original structures. Therefore, the following algorithm is a correct test for projectivity of a given input structure D: encode D into the term $\mathrm{term}(D)$, decode $\mathrm{term}(D)$ into the dependency structure $D' := \mathrm{dep}(\mathrm{term}(D))$, and test whether D' and D are isomorphic. This test will succeed if and only if D is projective. Since encoding and decoding are linear-time operations, and since checking that two dependency structures are isomorphic is a linear-time operations as well, we obtain the following result:

Lemma 3.3.3. *Let D be a dependency structure with n nodes. The question whether D is projective can be decided in time $O(n)$.* □

3.3.3 Related Work

Havelka [43] presents two algorithms for testing whether a given dependency structure is projective. The first algorithm, very much like ours, makes use of the one-to-one correspondence between projective dependency structures and treelet-ordered trees. The second algorithm searches for certain types of non-projective dependency edges. Both algorithms run in linear time.

Another projectivity test is proposed by Möhl [80]. It uses a post-order traversal of the input dependency structure to compute, for each node u, a bit vector representing the yield of u, and afterwards checks whether this bit vector represents a convex set. The number of bit vector operations used by this procedure is linear in the size of the input structure. It is difficult however to compare this machine-dependent measure with the asymptotic runtime that we have given for our algorithm.

3.4 Empirical Evaluation

In this section, we evaluate the practical relevance of the projectivity condition. Should it turn out that all interesting dependency structures of natural language utterances are projective, then that result would indicate that theories that do not obey the projectivity restriction fail to reflect a deeper truth about the nature of dependency. Of course, we cannot hope to ever have access to 'all interesting dependency structures'. However, we can estimate the empirical adequacy of projectivity by looking at representative samples of practically relevant data.

3.4.1 The Projectivity Hypothesis

Before we describe our experimental setup, we take a brief look at the historical assessment of projectivity as a constraint on dependency analyses.

Early work on formal dependency grammar shows conviction that projectivity has the status of a linguistic universal. To witness, Marcus [72], p. 230 cites Lecerf, who claimed that 'almost 100 percent of French strings are projective. The same seems to be true for German, Italian, Danish, and other languages'. This rather radical *projectivity hypothesis* is disputable even without empirical evaluation. In particular, one should note that projectivity is a property of theory-specific *analyses* of sentences, not of the sentences themselves. Consequently, not 'almost 100 percent of French strings', but at most all of their dependency analyses can be projective. This fundamental flaw of the argument may have been varnished over by the supremacy in the 1960s of dependency grammar formalisms that embraced projectivity as a central grammatical principle [27, 47]: there simply was no dependency grammar beyond 'projective dependency grammar'. In the linguistic schools of Eastern Europe, where the objects of linguistic description are languages with a word order far less rigid than English, the status of projectivity as a linguistic universal was early mistrusted (see e.g. [18, 69, 92]). This assessment eventually became accepted even in the Western literature, and today, 'most theoretical formulations of dependency grammar regard projectivity as the norm, but also recognize the need for non-projective representations of certain linguistic constructions, e.g., long-distance dependencies' [86].

With the availability of large corpora of dependency analyses, *dependency treebanks*, we are able today to complement theoretical considerations concerning projectivity by collecting data on its *practical* relevance: the data that we are evaluating forms the basis for many current applications that build on dependency-based representations, and the degree of projectivity in this data may have direct consequences for the design of these applications. Furthermore, under the assumption that the treebank data forms a representative sample of the set of useful dependency structures, these data also provide an indirect evaluation of the empirical adequacy of projectivity.

3.4.2 Experimental Setup

Our experiments are based on data from the Prague Dependency Treebank (PDT) [40, 41] and the Danish Dependency Treebank (DDT) [63]. The PDT was used in two versions: version 1.0 contains 1.5M, version 2.0 contains 1.9M tokens of newspaper text. Sentences in the PDT are annotated in three layers according to the theoretical framework of Functional Generative Description [42]. Our experiments concern only the analytical layer, and are based on the dedicated training section of the treebank. The DDT comprises 100k words of text selected from the Danish PAROLE corpus, with annotation of primary and secondary dependencies based on Discontinuous Grammar [64]. Only primary dependencies are considered in the experiments, which are based on the pseudo-randomized training portion of the treebank.[2] A total number of 19 analyses in the DDT were excluded because they contained annotation errors.

3.4.3 Results and Discussion

The results of our experiments are given in Table 3.4; we report the number and percentage of structures in each data set that satisfy or violate the projectivity condition.

Under the assumption that the three treebanks constitute a representative sample of the set of practically relevant dependency structures, our experiments clearly show that non-projectivity cannot be ignored without also ignoring a significant portion of real-world data. For the DDT, we see that about 15% of all analyses are non-projective; for the PDT, the number is even higher, around 23% in both versions of the treebank. Neither theoretical frameworks nor practical applications that are confined to projective analyses can account for these analyses, and hence cannot achieve perfect recall even as an ideal goal. In a qualification of this interpretation, one should note that projectivity fares much better under an evaluation metric that is based on the set of individual edges, rather than on the set of complete analyses: less than

[2] Since the DDT does not have a dedicated training section, it is custom practice to create such a section by splitting the entire data into blocks of 10 analyses each, and keeping blocks 1 to 8 for training.

Table 3.4. The number of projective dependency structures in three treebanks

	DDT		PDT 1.0		PDT 2.0	
projective	3 730	84.95%	56 168	76.85%	52 805	77.02%
non-projective	661	15.05%	16 920	23.15%	15 757	22.98%
TOTAL	4 391	100.00%	73 088	100.00%	68 562	100.00%

2% of the edges in the PDT data, and just around 1% of the edges in the DDT data are non-projective [76, 88].

3.4.4 Related Work

Our experiments confirm the findings of recent studies on data-driven parsing of non-projective dependency grammar [76, 88]. They are particularly similar in vein to a study presented by Nivre [85]. Nivre's main objective was to evaluate, how large a proportion of the structures found in the DDT and the PDT can be parsed using several restricted versions of the 'Fundamental Algorithm' for dependency parsing [13]. Using a version of that algorithm that only recognizes projective structures, and employing the treebanks as oracles to resolve ambiguities, Nivre effectively tested for projectivity. For the PDT part of the data, our results are identical to his, which in turn agree with counts previously reported by Zeman [122], p. 95. The minor deviation between our results and Nivre's for the DDT part of the data is explained by the 19 analyses that we excluded because they contained annotation errors. Havelka [44] provides data on the frequency of non-projective structures in data sets for Arabic, Bulgarian, Czech, Danish, Dutch, German, Japanese, Portuguese, Slovene, Spanish, Swedish, and Turkish. Notice however, that some of these data sets are no dependency treebanks, but result from the automatic conversion of treebanks that were originally annotated using constituent structures.

A qualitative rather than quantitative approach towards the evaluation of projectivity was taken by Pericliev and Ilarionov [92]. They used a hand-written dependency grammar for Bulgarian to create example sentences for all non-projective structures with 4 nodes (every larger non-projective structure contains such a structure) and found that about 85% of these structures could be instantiated with a grammatical sentence.[3] Just as our experiments, this result indicates that projectivity cannot be used as a language-theoretic universal. Nevertheless, Pericliev and Ilarionov concede that most non-projective analyses in Bulgarian correspond to word orders that are stylistically marked.

[3] Pericliev and Ilarionov are misled in assuming that 'the total number of non-projective situations [in dependency structures with 4 nodes] is 32' (p. 57): since the number of unrestricted dependency structures with 4 nodes is 64, and the corresponding number of projective structures is 30, there are in fact 34 non-projective dependency structures with four nodes.

4

Dependency Structures of Bounded Degree

As we have seen in the previous chapter, the phenomenon of non-projectivity cannot be ignored in practical applications. At the same time, the step from projectivity to unrestricted non-projectivity is quite a dramatic one. In this chapter, we study non-projective dependency structures under a gradual relaxation of projectivity, the *block-degree restriction*.

Following our program from the previous chapter, we first characterize the class of dependency structures with restricted block-degree in terms of a structural constraint (Section 4.1), then build an algebraic framework for this class (Section 4.2), next present an efficient algorithm that encodes dependency structures into terms (Section 4.3), and finally evaluate the practical relevance of the block-degree restriction on treebank data (Section 4.4).

4.1 The Block-Degree Measure

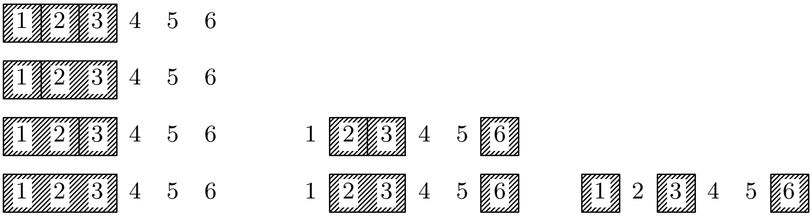
In projective dependency structures, each yield forms a set that is convex with respect to the precedence relation. In non-projective structures, yields may be discontinuous. In this section, we develop a formal measure that allows us to classify dependency structures based on their degree of non-projectivity: the minimal number of convex sets needed to cover all nodes of a yield.

4.1.1 Blocks and Block-Degree

The formal cornerstone of our measure is the notion of a *congruence relation* on a chain. In general, congruence relations (or simply: congruences) are equivalence relations that are compatible with certain properties of the underlying mathematical structure. For chains, a natural notion of congruence is obtained by requiring each equivalence class to form a convex set.

Definition 4.1.1. Let $\mathfrak{C} = (A\,;\preceq)$ be a chain, and let $S \subseteq A$ be a set. An equivalence relation on S is called a *congruence* on S, if each of its classes is convex with respect to $\mathfrak{C}$. □

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 33–49, 2010.



(a) $S_1 = \{1, 2, 3\}$ (b) $S_2 = \{2, 3, 6\}$ (c) $S_3 = \{1, 3, 6\}$

Fig. 4.1. Examples for congruence relations

Example 4.1.1. Let $\mathfrak{C}_6$ be the set $[6]$, equipped with the standard order on natural numbers, and consider the set $S_1 := \{1, 2, 3\}$. There are four possible congruence relations on S_1. Using shades to mark the elements of S_1, and boxes to mark equivalence classes, these relations can be visualized as in Figure 4.1a. Similarly, there are two possible congruence relations on the set $S_2 := \{2, 3, 6\}$ (depicted in Figure 4.1b), and one congruence on the set $S_3 := \{1, 3, 6\}$ (Figure 4.1c). □

The quotient of a set S by a congruence relation forms a partition of S in which every class is convex; we call such partitions *convex partitions*. Congruences on the same chain can be compared with respect to the *coarseness* of their quotients: given a set S and two partitions Π_1, Π_2 of S, we say that Π_1 is *coarser* than Π_2 (and that Π_2 is *finer* than Π_1), if for every class $C_2 \in \Pi_2$, there is a class $C_1 \in \Pi_1$ such that $C_2 \subseteq C_1$. The set of convex partitions of a given set together with the 'coarser-than' relation forms a complete lattice. As a consequence, there is a *coarsest congruence* on a given set.

Example 4.1.2 (continued). The lowermost congruence relation in Figure 4.1a is the coarsest congruence on the set S_1 in $\mathfrak{C}_6$ (all other congruence relations on S_1 have more equivalence classes), the topmost relation is the finest congruence on S_1. □

The coarsest congruence relation on a set can also be characterized directly:

Lemma 4.1.1. *Let $\mathfrak{C} = (A\,; \preceq)$ be a chain, and let $S \subseteq A$ be a set. Define a binary relation on S by putting $a \equiv_S b$ if and only if $\forall c \in [a, b].\ c \in S$. Then $\equiv_S$ is the coarsest congruence relation on S.* □

The cardinality of the quotient of a set S modulo the coarsest congruence relation on S provides us with a way to measure the 'non-convexity' of S: the more convex sets we need to cover all the elements of S, the less convex it is.

Definition 4.1.2. Let $\mathfrak{C} = (A\,; \preceq)$ be a chain, and let $S \subseteq A$ be a set. A *block* of S with respect to $\mathfrak{C}$ is an element of the quotient $S/{\equiv_S}$. The *block-degree* of S with respect to $\mathfrak{C}$ is the cardinality of the set $S/{\equiv_S}$, that is, the number of different blocks of S. □

Example 4.1.3 (continued). In the pictures in Figure 4.1, the blocks of a set are visualized as contiguous shaded regions, and the block-degree corresponds to the number of these regions. The block-degree of the set S_1 in $\mathfrak{C}_6$ is 1: the coarsest congruence relation on S_1 has only one block. More generally, every convex set has block-degree 1. The block-degree of the set S_2 is 2: there is no way to cover S_2 with less than 2 blocks. Finally, the block-degree of the set S_3 is 3. □

Rather than counting the number of blocks of a set, we can also count the number of discontinuities or *gaps* between the blocks. Formally, these concepts can be defined on the complement of a set relative to its *convex hull*.

Definition 4.1.3. Let $\mathfrak{C} = (A\,;\preceq)$ be a chain, and let $S \subseteq A$ be a set. The *convex hull* of S, $\mathbf{H}(S)$, is the smallest convex superset of S. The elements of the set $\overline{S} := \mathbf{H}(S) - S$ are called *holes* in S. □

Applying the definition of blocks and block-degree to sets of holes, we say that a *gap* in S is a class in the quotient $\overline{S}/{\equiv_{\overline{S}}}$, and the *gap-degree* of S is the cardinality of the quotient $\overline{S}/{\equiv_{\overline{S}}}$. The gap-degree of a set is obtained as its block-degree, minus 1.

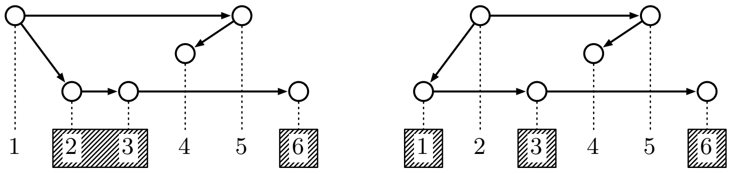
Example 4.1.4 (continued). We return to Figure 4.1, where gaps are visualized as contiguous non-shaded regions between blocks. The convex hull of the set S_1 is the set S_1 itself; thus, the set $\overline{S_1}$ is empty, and there are no gaps in S. The convex hull of the set S_2 is $\mathbf{H}(S_2) = \{2, 3, 4, 5, 6\}$, and the set $\{4, 5\}$ is the set of holes in S_2. This set also forms a gap in S_2, so the gap-degree of S_2 is 1. Finally, for the set S_3 we have $\mathbf{H}(S_3) = [6]$ and $\overline{S_3} = \{2, 4, 5\}$; the gap-degree of S_3 is 2. □

In the following, we usually talk about blocks and block-degree, but all our results could also be expressed in terms of gaps and the gap-degree measure.

4.1.2 A Hierarchy of Non-projective Dependency Structures

We now apply the block-degree measure to dependency structures. With the definition of projectivity in mind, the interesting congruences on dependency structures are the coarsest congruences on their yields: two nodes v_1, v_2 belong to the same block of a yield $\lfloor u \rfloor$, if all nodes between v_1 and v_2 belong to $\lfloor u \rfloor$ as well. The maximal number of blocks per yield is a measure for the non-projectivity of a dependency structure.

Definition 4.1.4. Let D be a dependency structure, and let u be a node of D. The set of *blocks* of u is the set $\lfloor u \rfloor/{\equiv_{\lfloor u \rfloor}}$, where the congruence relation $\equiv_{\lfloor u \rfloor}$ is defined relative to the precedence relation underlying D. □



(a) D_1, block-degree 2

(b) D_2, block-degree 3

Fig. 4.2. Two non-projective dependency structures

Definition 4.1.5. Let D be a dependency structure. The *block-degree* of a node u of D is the number of blocks of u. The block-degree of D is the maximum among the block-degrees of its nodes. □

Example 4.1.5. Figure 4.2 shows two examples of non-projective dependency structures. For both structures, consider the yield of the node 2. In structure D_1, the yield $\lfloor 2 \rfloor$ falls into two blocks, $\{2,3\}$ and $\{6\}$. Since this is also the maximal number of blocks per yield, the block-degree of D_1 is 2. In structure D_2, the yield $\lfloor 2 \rfloor$ consists of three blocks, $\{1\}$, $\{3\}$, and $\{6\}$; the block-degree of D_2 is 3. □

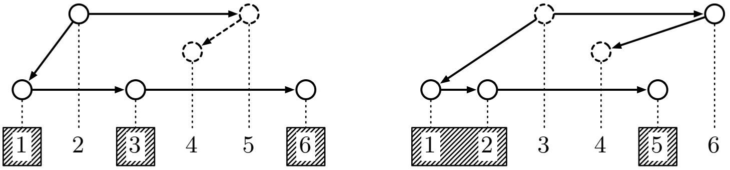
Let us say that a dependency structure is *block k*, if its block-degree is at most k. We write $\mathcal{D}_k$ for the class of all dependency structures that are block k. It is immediate from this definition that the class $\mathcal{D}_k$ is a proper subclass of the class $\mathcal{D}_{k+1}$, for all $k \in \mathbb{N}$. It is also immediate that a dependency structure is projective if and only if it belongs to the class $\mathcal{D}_1$. Thus, the block-degree measure induces an infinite hierarchy of ever more non-projective dependency structures, with the class of projective structures at the lowest level of this hierarchy. This is interesting because it allows us to scale the complexity of our formal models with the complexity of the data: the transition from projectivity to full non-projectivity becomes gradual. A crucial question is, of course, whether block-degree is a useful measure in practice. To answer this question, we evaluate the practical relevance of the block-degree measure in Section 4.4.

4.1.3 Related Work

The gap-degree measure (and hence, the block-degree measure) is intimately related to the notion of *node-gaps complexity*, due to Holan et al. [48]. Node-gaps complexity was originally introduced as a complexity measure for derivations in a dependency grammar formalism. Later, it was also applied to the empirically more transparent *results* of these derivations, objects essentially the same as our dependency structures. In this latter application, node-gaps complexity and gap-degree are identical. Note however that some authors [42, 122] use the term ‘gap’ to refer to a node—rather than a set of nodes—between two blocks of a yield. This is what we have called a 'hole' in Definition 4.1.3. Havelka [44] defines the 'gap' of a dependency edge $u \to v$ as the set of holes in the set $\lfloor u \rfloor \cap (u, v)$.

We can view the block-degree of a set as a descriptive complexity measure, similar to Kolmogorov complexity in algorithmic information theory: Once a chain $(A\,;\preceq)$ is given, a convex subset of A can be represented by a pair of elements from A, namely the minimal and the maximal element of the set. In this way, even very large sets can be represented with little information. However, the higher the block-degree of a set, the more elements of A we need to represent it, and the less benefit an interval representation has over an explicit representation.

The block-degree measure quantifies the non-projectivity of a dependency structure by counting the number of contiguous blocks in the yields of the structure. A similar measure, based on edges, was introduced by Nivre [85]. For an edge $e = u \to v$ in a dependency structure D, let us write $\mathfrak{F}_e$ for the forest that results from restricting the governance relation in D to the nodes in the open interval (u, v). The *degree* of the edge e (in the sense of Nivre) is the number of those components in $\mathfrak{F}_e$ that are not governed by u in D; the degree of D is the maximum among the degrees of its edges. This degree measure is incomparable to our block-degree measure. To see this, consider the two dependency structures depicted in Figure 4.3. The left structure (Figure 4.3a) has block-degree 3 and edge-degree 1, as the open interval $(3, 6)$ that corresponds to the edge $3 \to 6$ contains one component not governed by 3, and this is the maximal number of components per edge. On the other hand, the right structure (Figure 4.3b) has block-degree 2 and edge-degree 2, as the edge interval $(2, 5)$ contains two distinct components not governed by the node 2.



(a) block degree 3, edge degree 1 (b) block degree 2, edge degree 2

Fig. 4.3. Block degree and edge degree are incomparable

4.2 Algebraic Framework

In this section, we generalize the algebraic framework developed in Section 3.2 to dependency structures with restricted block-degree.

4.2.1 Traversal of Block-Ordered Trees

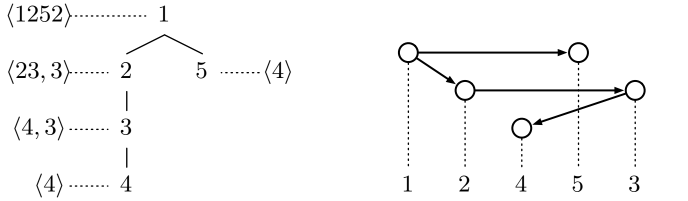
One of the essential properties of our procedure for the traversal of treelet-ordered trees is that for each node u of the input tree T, the call Treelet-Order-Collect(u) returns a linearization of the nodes in the subtree of T that is rooted at u. This property ensures that we can interpret the result of a call as a dependency structure, but at the same time constrains this structure to be projective. We now develop a procedure Block-Order-Collect that returns a linearization not of a complete yield, but only of some given block of that yield. To do so, we allow the procedure to be called on a node more than once: the ith call on u produces a linearization of the ith block of u, where blocks are assumed to be numbered in the order of their precedence.

Figure 4.1 shows pseudo-code for Block-Order-Collect. The implementation assumes the existence of a global array *calls* that records for each node u the number of times that the procedure has been called on u. It further assumes that each node u is annotated with lists $\mathit{order}[u][i]$ of nodes in the treelet rooted at u.

Example 4.2.1. Figure 4.4 shows an order-annotated tree and the dependency structure induced by its traversal according to the procedure in Table 4.1 when called on the root node of that tree. We assume that the array *calls* is initialized with all zeros. The tuples at the ends of the dotted lines represent the annotated orders. Specifically, the list $\mathit{order}[u][i]$ can be found as the ith

Table 4.1. Traversal of a block-ordered tree

```
Block-Order-Collect($u$)
$L \leftarrow$ nil; $\mathit{calls}[u] \leftarrow \mathit{calls}[u] + 1$
foreach $v$ in $\mathit{order}[u][\mathit{calls}[u]]$
      do if $v = u$
            then $L \leftarrow L \cdot [u]$
            else  $L \leftarrow L \cdot$ Block-Order-Collect($v$)
return $L$
```



(a) tree (b) D_3 (block-order traversal)

Fig. 4.4. A block-ordered tree and its corresponding dependency structure

component of the tuple annotated at the node u. To give an example, the order annotations for the node 2 are $\mathit{order}[2][1] = 23$ and $\mathit{order}[2][2] = 3$. □

Our first aim in this section is to show that suitably annotated trees together with the procedure BLOCK-ORDER-COLLECT are expressive enough to fully characterize the class of dependency structures with finite block-degree—just as treelet-ordered trees and our procedure for traversing them are expressive enough to fully characterize the class of projective structures. More specifically, we want to show that trees in which no node is annotated with more than k lists are expressive enough to characterize the class of structures with block-degree at most k.

For our proofs to go through, we need to be more specific about the exact form of the order annotations in the inputs to BLOCK-ORDER-COLLECT. Without further constraints, the procedure may fail to induce dependency structures:

- Assume that two distinct calls BLOCK-ORDER-COLLECT(u) return lists that contain the node u, or that a single call returns a list that contains the node u more than once. In both of these cases, the result of the traversal does not qualify as a linearization of the input tree. A similar situation arises if none of the calls BLOCK-ORDER-COLLECT(u) returns a list that contains the node u.
- Assume that BLOCK-ORDER-COLLECT(u) is called more often than the number of lists annotated at u. In this case, the results of some of the calls are undefined. Similarly, if BLOCK-ORDER-COLLECT(u) is called less often than there are lists annotated at u, then the linearization may be incomplete.

To prevent these problems, we require the inputs to BLOCK-ORDER-COLLECT to be well-typed, in the following sense: For each node u, let $k(u)$ be the number of lists annotated at u. We require that, in union, these lists contain exactly one occurrence of the node u, and exactly $k(v)$ occurrences of the node v, for all children v of u. These restrictions ensure that the mapping from trees to dependency structures is well-defined. It is not necessarily injective:

- Assume that some list $\mathit{order}[u][i]$ is empty. Then we can modify the order annotations without altering the induced dependency structure as follows: delete the list $\mathit{order}[u][i]$, and re-index the remaining annotations at u accordingly.
- Assume that some list $\mathit{order}[u][i]$ contains two adjacent occurrences of some child v of u. Then we can modify the order annotations without altering the induced dependency structure as follows: delete the second occurrence, append the corresponding order annotation to the annotation corresponding to the first occurrence, and re-index the remaining order annotations at v accordingly.

To prevent these ambiguities, we require that no list *order*$[u][i]$ is empty, and that no list *order*$[u][i]$ contains two or more adjacent occurrences of the same node v. We call trees that satisfy all of these requirements *block-ordered trees*.

4.2.2 Segmented Dependency Structures

There is one more thing that we need to take care of. Consider a block-ordered tree in which the root node is annotated with more than one list of nodes. When we call Block-Order-Collect on the root node of this tree, all but the first of these lists are ignored, and hence, the linearization of the tree is incomplete, and fails to induce a dependency structure. One way to remedy this problem is to require the root nodes of block-ordered trees to be annotated with exactly one list; but this would break inductive arguments like the one that we used in the proof of Lemma 3.2.1. We therefore opt for another solution, motivated by the following observation: Let T be a block-ordered tree with root node r, and let u be a non-root node of T. For notational convenience, put $k := k(u)$. The well-typedness conditions ensure that the calls to Block-Order-Collect on u can be understood as a tuple $\langle\, \vec{v}_i \mid i \in [k]\,\rangle$ of lists of nodes, where $\vec{v}_i$ is the result of the ith call to Block-Order-Collect(u), for $i \in [k]$, and $\vec{v}_1 \cdots \vec{v}_k$ forms a linearization of the subtree rooted at u. Only for the root node r, the procedure is called only once, independently of the number of annotated lists. In order to do away with this asymmetry, we stipulate that the call of Block-Order-Collect on r should return the $k(r)$-tuple $\langle\, \text{Block-Order-Collect}(r)_i \mid i \in [k(r)]\,\rangle$, where Block-Order-Collect$(r)_i$ stands for the ith call to the node r. Of course, in order for this to make sense, we need to say what such an output should mean in the context of dependency structures. This gives rise to the notion of *segmented dependency structures*, which essentially are dependency structures where even the root nodes can have block-degrees greater than one.

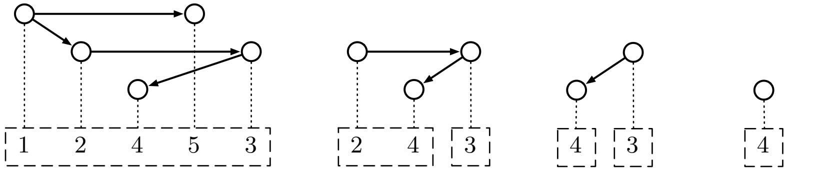
Definition 4.2.1. Let $D = (V\,; \trianglelefteq, \preceq)$ be a dependency structure, and let $\equiv$ be a congruence relation on D. The *segmentation* of D by $\equiv$ is the structure $D' := (V\,; \trianglelefteq, \preceq, R)$, where R is a new ternary relation on V defined as follows:

$$(u, v_1, v_2) \in R \quad :\Longleftrightarrow \quad v_1 \equiv v_2 \;\wedge\; \forall w \in [v_1, v_2].\; w \in \lfloor u \rfloor\,.$$

The elements of the set $V/{\equiv}$ are called the *segments* of D'. □

We write $v_1 \equiv_u v_2$ instead of $(u, v_1, v_2) \in R$.

Example 4.2.2. Figure 4.5 shows how we visualize segmented dependency structures: we use boxes to group nodes that belong to the same segment; all other congruences $\equiv_u$ are uniquely determined by this choice. As an example, $4 \equiv_1 3$ holds in D_4 because both 4 and 3 lie in the same segment of D_4, and all nodes between them are governed by the node 1. At the same time, $4 \not\equiv_2 3$ holds in D_4: while both 4 and 3 belong to the same segment of D_4, the node 5, which is situated between 4 and 3, is not governed by 2. The non-congruence



(a) $D_4 \cong D_3$ (b) $D_4/2$ (c) $D_4/3$ (d) $D_4/4$

Fig. 4.5. Segmented dependency structures

$4 \not\equiv_2 3$ also holds in the substructure $D_4/2$, this time because 4 and 3 do not even lie in the same segment. □

For each node u, the relation $\equiv_u$ is the coarsest congruence on $\lfloor u \rfloor$ that is finer than $\equiv$. Based on this observation, we adapt our definition of blocks (Definition 4.1.4):

Definition 4.2.2. Let D be a segmented dependency structure, and let u be a node of D. The set of *blocks* of u is the set $\lfloor u \rfloor / {\equiv_u}$. □

In segmented dependency structures with just one segment, this definition coincides with the old one. In other structures, the new definition ensures that elements from different segments belong to different blocks of all nodes of the structure.

We call the number of segments of a segmented dependency structure the *sort* of that structure, and write $\mathcal{D}^{\equiv}_{k}$ for the class of all segmented dependency structures of sort k. By our definition of block-degree, the block-degree of a segmented dependency structure is at least as high as its sort. The class $\mathcal{D}^{\equiv}_{1}$ is essentially the same as the class $\mathcal{D}$ of all dependency structures, and it will be convenient not to distinguish them to carefully. We now connect segmented dependency structures to our modified tree traversal.

Definition 4.2.3. Let T be a tree, and let $k \in \mathbb{N}$. A *linearization* of T with k components is a k-tuple $L = \langle\, \vec{u}_i \mid i \in [k] \,\rangle$ such that $\vec{u} := \vec{u}_1 \cdots \vec{u}_k$ is a list of the nodes of T in which each node occurs exactly once. The segmented dependency structure *induced* by a linearization L of T is the structure in which the governance relation is isomorphic to T, the precedence relation is isomorphic to $\vec{u}$, and the segments are isomorphic to the tuple components of L. □

We can now show the correspondents of the Lemmata 3.2.1, 3.2.2, and 3.2.3 for projective structures. To do so, we regard the functions lin and dep as *sorted* functions: given a block-ordered tree T in which the root node is annotated with k lists, the function lin maps T to the linearization of T with k components that is computed by the traversal of the input tree according to Block-Order-Collect, and the function dep maps T to the segmented dependency structure of sort k that is induced by $\mathrm{lin}(T)$. Apart from this change, the proofs carry over without larger modifications. We therefore get the following Lemma:

Lemma 4.2.1. *For every segmented dependency structure D, there exists exactly one block-ordered tree T such that $dep(T) = D$. Furthermore, if T is a block-ordered tree in which each node is annotated with at most k lists, for some $k \in \mathbb{N}$, then $dep(T)$ is a segmented dependency structure with block-degree at most k.* □

4.2.3 Order Annotations

Next we take the step from the algorithmic to the algebraic and encode block-ordered trees as terms over an extended set of order annotations. This encoding is a relatively straightforward generalization of the procedure that we presented in Section 3.2.3. The major novelty comes from the requirement that we need to ensure that all decodings of terms satisfy the well-typedness conditions. To do so, we now understand the set Ω of order annotations as a *sorted* set; as sorts, we use the natural numbers.

Let T be a block-ordered tree with root node u, and put $k := k(u)$. We need the following auxiliary sequences: first, the vector $\vec{v} = v_1 \cdots v_m$ obtained from the concatenation $\mathit{order}[u][1] \cdots \mathit{order}[u][k]$ of the list-based order annotations by removing the node u and all but the first occurrence of each other node; second, for each $j \in [k]$, the string $\vec{\imath}_j$ obtained from the list $\mathit{order}[u][j]$ by replacing every child of u by its position in $\vec{v}$, and u itself by the symbol 0. For each $j \in [m]$, put $k_j := k(v_j)$. The *term* $t(T)$ corresponding to T is then defined recursively as

$$t(T) \ := \ \langle \vec{\imath}_1, \dots, \vec{\imath}_k \rangle (t(T/v_1), \dots, t(T/v_m)) \,.$$

In this definition, the string $\langle \vec{\imath}_1, \dots, \vec{\imath}_k \rangle$ is understood as a term constructor of type $k_1 \times \cdots \times k_m \to k$. From now on, we use Ω to denote the set of all such constructors. We define the *degree* of a symbol $\omega \in \Omega$, $\deg(\omega)$, as the maximum over its input and output sorts, and put $\Omega(k) := \{\, \omega \in \Omega \mid \deg(\omega) \leq k \,\}$. Note that, by this definition, the set $\Omega(k)$ is exactly what we need in order to encode the set of all block-ordered trees with up to k lists per node, and therefore, the set of all segmented dependency structures with block-degree at most k. Specifically, the set $\Omega(1)$ is essentially identical to our previous definition of Ω for projective dependency structures.

Example 4.2.3. For the dependency structure D_3 shown in Figure 4.4, the generic procedure to extract the order annotations from a dependency structure described in the proof of Lemma 3.2.2 (page 25) yields the following list-based order annotations:

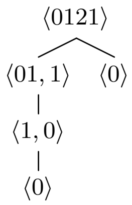


Fig. 4.6. The term for the block-ordered tree from Figure 4.4

$$\begin{array}{llll}
\text{for node 1:} & L = 12453\,, & L' = 12252\,, & \mathit{order}[1] = 1252\,; \\
\text{for node 2:} & L = 12453\,, & L' = \square 23 \square 3\,, & \mathit{order}[2] = 23 \square 3\,; \\
\text{for node 3:} & L = 12453\,, & L' = \square\square 4 \square 3\,, & \mathit{order}[3] = 4 \square 3\,; \\
\text{for node 4:} & L = 12453\,, & L' = \square\square 4 \square\square\,, & \mathit{order}[4] = 4\,; \\
\text{for node 5:} & L = 12453\,, & L' = \square\square\square 5 \square\,, & \mathit{order}[5] = 5\,.
\end{array}$$

If we read the values of the strings $\mathit{order}[u]$ as tuples, where the symbol $\square$ separates tuple components, we obtain the annotations shown in Figure 4.4. From these, by the construction above, we construct the following order annotations:

$$\langle 0121 \rangle\colon 2 \times 1 \to 1\,, \quad \langle 01, 1 \rangle\colon 2 \to 2\,, \quad \langle 1, 0 \rangle\colon 1 \to 1\,, \quad \langle 0 \rangle\colon 1\,.$$

Figure 4.6 shows a term built over these constructors; this term encodes the block-ordered tree from Figure 4.4. □

4.2.4 Dependency Structure Algebras

We have now reached a situation very similar to the situation at the beginning of Section 3.2.4: we have identified a sorted set Ω and a sorted bijection dep: $T_\Omega \to \mathcal{D}^{\equiv}$ between terms over Ω and segmented dependency structures. This means that we can give the set $\mathcal{D}^{\equiv}$ an algebraic structure.

Definition 4.2.4. Let $\omega\colon k_1 \times \cdots \times k_m \to k$ be an order annotation. The *composition operation* corresponding to ω is the map $f_\omega\colon \mathcal{D}^{\equiv}_{k_1} \times \cdots \times \mathcal{D}^{\equiv}_{k_m} \to \mathcal{D}^{\equiv}_{k}$ defined as

$$f_\omega(D_1, \dots, D_m) \quad := \quad \mathrm{dep}(\omega(\mathrm{term}(D_1), \dots, \mathrm{term}(D_m)))\,.$$ □

Each composition operation f_ω simulates a single step of the block-order traversal: given a sequence of argument structures, it returns the segmented dependency structure that can be decomposed into the given argument structures in the way that is specified by the order annotation ω.

Definition 4.2.5. Let $\Sigma \subseteq \Omega$ be a finite set of order annotations. The *dependency algebra* over Σ is the Σ-algebra that has the $\mathcal{S}_\Sigma$-indexed set $\langle\, \mathrm{dep}(\Sigma_i) \mid i \in [k] \,\rangle$ as its carrier, and interprets each $\omega \in \Sigma$ by the composition operation corresponding to ω. □

Theorem 4.2.1. *Let $\Sigma \subseteq \Omega$ be a finite set of order annotations. Then the dependency structure algebra over Σ is isomorphic to the (many-sorted) term algebra over Σ.* □

4.3 Algorithmic Problems

In this section, paralleling Section 3.3, we address three of the algorithmic problems related to our algebraic view on non-projective dependency structures: decoding, encoding, and computing the block-degree for a given dependency structure. The decoding problem is essentially solved by the procedure Block-Order-Collect that we gave in Table 4.1. The main contribution of this section is an efficient algorithm to encode a non-projective dependency structure into a term. The algorithm that computes the block-degree is a straightforward extension of the encoding algorithm.

4.3.1 Encoding

On page 25, we described a generic procedure to extract the order annotations from a dependency structure. A naïve implementation of this procedure takes time quadratic in the number of nodes of the input structure. The algorithm that we present in this section may perform significantly better; it runs in time linear in the number of blocks in the input structure, of which there are *at most* quadratically many, but often less.

The crucial component of our encoding algorithm is a procedure that transforms the input structure into a certain tree representation called *span tree*, from which all order annotations can be easily read off. The span tree T for a dependency structure D is a labelled, ordered tree in which each node has one of two types: it can be a *block node* or an *anchor*. The block nodes of T stand in one-to-one correspondence with the blocks of D; the anchors stand in one-to-one correspondence with the singletons. For a node u of T, we write $S(u)$ for the set of nodes in D that corresponds to u. The dominance relation of T represents an inclusion relation in D: u strictly dominates v if and only if $S(u) \supseteq S(v)$ and either u is a block node and v is an anchor, or both u and v are block nodes. By this relation, all block nodes are inner nodes, and all anchors are leaf nodes in T. The precedence relation of T represents a precedence relation in D: u strictly precedes v if and only if all nodes in $S(u)$ precede all nodes in $S(v)$.

Example 4.3.1. Figure 4.7 shows the dependency structure from Figure 4.4 and the span tree for this structure. Consider the root node of the structure, the node a. The yield of a consists of a single block, which contains all the nodes of the structure, positions 1 to 5 in the precedence order. This information is represented in the span tree in that the root node of this tree is labelled with the triple $(a, 1, 5)$. The block of a decomposes into four components; read

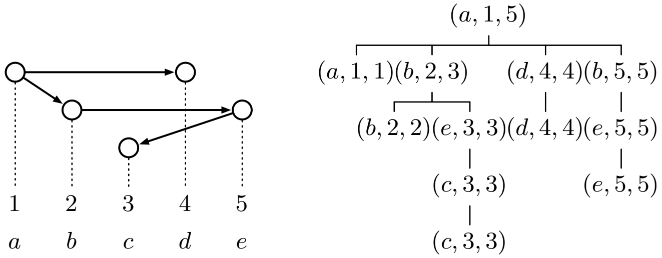


Fig. 4.7. A non-projective structure and the corresponding span tree

in their order of precedence, these are: the singleton $\{a\}$, the first block of the node b, the block of the node d, and the second block of b. Note that, since b contributes two blocks to the block of a, we find two nodes of the form (b, i, j) as children of $(a, 1, 5)$ in the span tree. Note furthermore that the precedence order on the components of the block of a is reflected by the sibling order on their corresponding nodes in the span tree. □

We now describe the general structure of an algorithm that transforms a dependency structure into its span tree. We assume that the input structure is given to us as a collection of nodes, where each node u is equipped with a set *children*$[u]$ of its children and an integer *pos*$[u]$ that represents the position of u with respect to the precedence relation. Our algorithm can be separated into two phases:

- In the first phase, we allocate two global data structures: an array *right* that will map the left endpoints of blocks to their right endpoints, and an array *sub* that will record the component structures of blocks. Each element of these arrays is initialized to the void value, which we write as $\perp$.
- In the second phase of the algorithm, we perform a post-order traversal of the input structure. For each node u, we compute a set *trees*$[u]$ that contains the span trees for the blocks of the yield of u. An element of *trees*$[u]$ is a four-tuple of the form (w, i, j, S), where w is a node in the treelet rooted at u, the position i is the left endpoint of the span represented by the tree, the position j is the right endpoint, and the list S contains the trees for the components in their left-to-right order. If u is a leaf node, then *trees*$[u]$ consists of the single tree $(u, \mathit{pos}[u], \mathit{pos}[u], [(u, \mathit{pos}[u], \mathit{pos}[u], \textsc{nil})])$. If u is an inner node, then the set *trees*$[u]$ is obtained by exhaustive *merging* of this trivial tree and the span trees that were constructed for the children of u. Two trees can be merged if they correspond to adjacent spans of positions.

In the following, we concentrate on the procedure STEP(u) that processes a single node u during the post-order traversal; pseudo-code for this procedure is given in Table 4.2. In line 1, we collect the trees for the subblocks of u. In lines

Table 4.2. Constructing the span tree for a dependency structure

```
STEP($u$)
 1  $T \leftarrow \bigcup\{\, \mathit{trees}[v] \mid v \in \mathit{children}[u] \,\} \cup \{(u, \mathit{pos}[u], \mathit{pos}[u], \textsc{nil})\}$
 2  foreach $(w, i, j, S)$ in $T$
 3        do $\mathit{right}[i] \leftarrow j$; $\mathit{sub}[i] \leftarrow$ if $i = \mathit{pos}[u]$ then $[(u, \mathit{pos}[u], \mathit{pos}[u])]$ else $S$
 4  foreach $(w, i, j, S)$ in $T$
 5        do $k \leftarrow j + 1$
 6           while $\mathit{right}[k] \neq \bot$
 7                 do $\mathit{right}[i] \leftarrow \mathit{right}[k]$; $\mathit{sub}[i] \leftarrow \mathit{sub}[i] \cdot \mathit{sub}[k]$
 8                    $\mathit{right}[k] \leftarrow \bot$; $\mathit{sub}[k] \leftarrow \bot$; $k \leftarrow \mathit{right}[i] + 1$
 9  foreach $(w, i, j, S)$ in $T$
10        do if $\mathit{right}[i] \neq \bot$
11              then $\mathit{trees}[u] \leftarrow \mathit{trees}[u] \cup \{(u, i, \mathit{right}[i], \mathit{sub}[i])\}$
12                   $\mathit{right}[i] \leftarrow \bot$; $\mathit{sub}[i] \leftarrow \bot$
```

2–3, we register these trees in the global data structures—in particular, for each tree (w, i, j, S), we register the position j as the right endpoint of a span that starts at position i. In lines 4–8, we merge the spans of adjacent trees into larger spans: we try each tree as a trigger to merge all right-adjacent spans into a new span, removing all traces of the old spans (line 8). The result of a merger spans the positions from the left endpoint of the trigger to the right endpoint of the span that was right-adjacent to it. In lines 9–12, we construct the set $\mathit{trees}[u]$ from the spans that remain after merging, and remove all traces from the global data structures.

We now look at the asymptotic complexity of the algorithm. The following invariant is essential for the analysis (and for the correctness proof, which we will omit):

> Every element $\mathit{right}[i]$ that receives a non-void value during a call to the procedure STEP is void again when this call finishes.

This can be seen as follows. The only places in a call to STEP where non-void values are assigned to *right* are in lines 3 and 7. The assignment in line 3 is witnessed by a tree from the set T, which is not altered after line 1; the assignment in line 7 merely overwrites a previous assignment. For all trees in T, it is checked in line 10 whether the element $\mathit{right}[i]$ is assigned, and if so, the element is made void. Therefore, every element of *right* is void when a call to STEP finishes. Since every element is void before the first call to STEP, and is not altered in between calls to STEP, it is void both before and after any call to STEP. A similar argument holds for the array *sub*.

Lemma 4.3.1. *Let D be a dependency structure with n nodes and g gaps. Then the span tree corresponding to D can be constructed in time $O(n + g)$.* □

Proof. Let us assume that we can use the algorithm that we have outlined above to construct the span tree for D. The algorithm breaks down into two phases: the initialization of the global data structures, and the tree traversal with calls to Step. The first phase takes time $O(n)$; the second phase can be implemented to take time linear in the sum over the times taken by the calls to Step. We show that each such call can be implemented to take time linear in the number of subblocks of the visited node. The total number of subblocks is asymptotically identical to the number of blocks in the input structure, which is $n + g$: each node has one more block than gaps.

Fix some node u of D, and consider the call Step(u). Let m be the number of subblocks of u; this number is obtained as the number of blocks of the children of u, plus 1 for the trivial block containing the singleton u. Note that $m = |T|$. We now check the runtime of the constructs in Table 4.2: The assignment in line 1 can be implemented to run in time $O(m)$; all other elementary operations can be implemented to run in time $O(1)$. Each of the **foreach** loops is executed $O(m)$ times. Finally, we analyse the runtime of the **while** loop. During each iteration of that loop, one of the elements of *right* is made void. Because of the invariant given above, all elements of *right* are void before the call Step(u), and since only m fields of *right* are initialized in line 3, the **while** loop cannot make more than m iterations *during a single call to* Step. Putting everything together, we see that every call to Step takes time $O(m)$. ■

This finishes our discussion of the transformation of the input dependency structure into a span tree. We now explain how to read off the order annotations from this tree. This can be done in a single pre-order traversal.[1] We first initialize an array $\mathit{order}[u]$ that maps nodes to their order annotations. For each non-leaf node (u, i, j, S) in the span tree, we construct a list L that contains the first components of the children of that node in their left-to-right order, and add this list to the array $\mathit{order}[u]$. At the end of the tree traversal, each list $\mathit{order}[u]$ contains the full order annotation for the node u.

Example 4.3.2 (continued). Consider the root node of the span tree depicted in Figure 4.7, the node $(a, 1, 5)$. The children of this node are $(a, 1, 1)$, $(b, 2, 3)$, $(d, 4, 4)$, and $(b, 5, 5)$. The corresponding order annotation is $\mathit{order}[a] = \langle abdb \rangle$. Similarly, for node b, we get $\mathit{order}[b] = \langle be, e \rangle$. This yields the order annotations $\langle 0121 \rangle$ and $\langle 01, 1 \rangle$, which can also be found at the corresponding nodes of the term for the encoded dependency structure (Figure 4.6). □

In conclusion, we get the following result:

Lemma 4.3.2. *Let D be a dependency structure with n nodes and g gaps. Then the term $\mathit{term}(D)$ can be computed in time $O(n + g)$.* □

Note that, in a dependency structure with block-degree k, the number $n + g$ of blocks is bounded by the number $k \cdot n$. Therefore, a coarser bound on the complexity of our encoding algorithm is $O(k \cdot n)$.

[1] Note that, given that the order in the span tree reflects the precedence order of the blocks in the dependency structure, during a pre-order traversal we visit the blocks from left-to-right.

Kuhlmann and Satta [68] present an alternative algorithm for extracting the order annotations from a dependency tree in time $O(n + g)$.

4.3.2 Computing the Block-Degree of a Dependency Structure

It is very easy to extend the encoding algorithm into an algorithm that computes the block-degree of a dependency structure. Since there is a direct correspondence between the block-degree of a node u in a dependency structure D and the degree of the order annotation at u in the term $\text{term}(D)$ that encodes D (see our discussion on page 42), it suffices to compute the encoding and count the relevant numbers. This can be done in time $O(n + g)$.

Lemma 4.3.3. *Let D be a dependency structure with n nodes and g gaps. Then the block-degree of D can be computed in time $O(n + g)$.* □

It is also possible to parameterize the algorithm that constructs the span tree for the input structure by a constant k such that it terminates as soon as it discovers that the tree to be constructed contains at least one node with a block-degree that exceeds k. In this way, a test whether a given dependency structure has block-degree k can be carried out in time $O(k \cdot n)$.

4.4 Empirical Evaluation

Using the algorithms developed in the previous section, we now evaluate the coverage of different block-degrees on treebank data. Specifically, we check how many and how large a percentage of the structures in the three treebanks that we used in the experiments reported in Section 3.4 have a block-degree of exactly k, for increasing values of k. Table 4.3 shows the results of the evaluation. The numbers and percentages for block-degree 1 reiterate the results for projective dependency structures from Table 3.4; the counts for structures with block-degree greater than 1 partition the figures for non-projective structures in that table.

Table 4.3. Dependency structures of various block-degrees in three treebanks

block-degree	DDT		PDT 1.0		PDT 2.0	
1 (projective)	3 730	84.95%	56 168	76.85%	52 805	77.02%
2	654	14.89%	16 608	22.72%	15 467	22.56%
3	7	0.16%	307	0.42%	288	0.42%
4	–	–	4	0.01%	1	< 0.01%
5	–	–	1	< 0.01%	1	< 0.01%
TOTAL	4 391	100.00%	73 088	100.00%	68 562	100.00%

The general impression that we get from the experiments is that even a small step beyond projectivity suffices to cover virtually all of the data in the three treebanks. Specifically, it is sufficient to go up to block-degree 2: the structures with block-degree greater than 2 account for less than half a percent of the data in any treebank. These findings confirm similar results for other measures of non-projectivity, such as Nivre's *degree* [85] and Havelka's *level types* [44]. Together, they clearly indicate that to contrast only projective and non-projective structures may be too coarse a distinction, and that it may be worthwhile to study classes of dependency structures with intermediate degrees of non-projectivity. The class of dependency structures with block-degree at most 2 appears to be a promising starting point.

Note that, from a linguistic point of view, block-degree is a measure of the discontinuity of a syntactic unit. While we have formulated it for dependency trees, recent work has also evaluated it as a measure for phrase-structure trees [71].

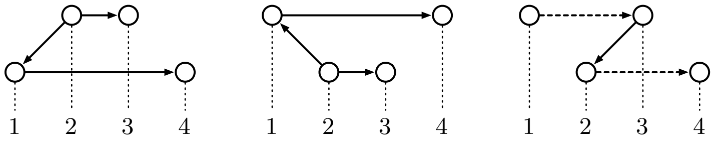
5

Dependency Structures without Crossings

The block-degree of a dependency structure is a quantitative property—it measures the independence of governance and precedence along an infinite scale of possible values. In this chapter, we study *well-nestedness*, a property related not to the degree, but to the form of non-projectivity in dependency structures. To motivate the well-nestedness restriction, we first look at another structural constraint, *weak non-projectivity*, and investigate its entanglement with the block-degree measure (Section 5.1). From there we are led to the relational and the algebraic characterization of well-nestedness, and to an efficient test for this property (Section 5.2). At the end of the chapter, we evaluate and compare the empirical adequacy of weak non-projectivity and well-nestedness (Section 5.3).

5.1 Weakly Non-projective Dependency Structures

Let us go back to our motivation of projectivity in Chapter 3. Recall that we had to refine the slogan that projectivity should 'disallow dependency analyses with pictures that contain crossing edges' because some pictures of dependency structures *with* crossing edges can be 'fixed' by changing the vertical positions of some nodes—to witness, consider the two pictures of the non-projective structure D_1 that are shown in Figures 5.1a and 5.1b. In this



(a) D_1 (b) D_1 (c) D_2

Fig. 5.1. Three pictures of non-projective dependency structures. The structure shown in pictures (a) and (b) is weakly non-projective; the structure shown in picture (c) contains overlapping edges.

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 51–62, 2010.

section, we return to our original slogan, and take a closer look at dependency structures that can be redrawn without crossing edges. We call such structures *weakly non-projective*, a term that we borrow from the Russian tradition of dependency grammar [18].

5.1.1 Definition of Weak Non-projectivity

In Chapter 3, we used the term ‘crossing edge’ to refer to a dependency edge that crosses a projection line. There is another natural meaning: that a ‘crossing edge’ is a dependency edge that crosses another edge. We formalize this situation as follows.

Definition 5.1.1. Let $\mathfrak{C} = (A\,; \preceq)$ be a chain. Two $\mathfrak{C}$-intervals B and C *overlap*, if one of the following holds true:

$$\min B \prec \min C \prec \max B \prec \max C \tag{a}$$

$$\min C \prec \min B \prec \max C \prec \max B \tag{b}$$

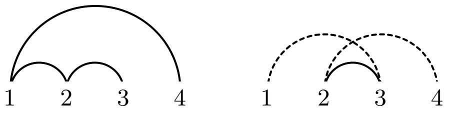
We write $B \between C$ to assert that B and C overlap.

Definition 5.1.2. Let D be a dependency structure. Two edges $v_1 \to v_2$, $w_1 \to w_2$ in D *overlap*, if $[v_1, v_2]$ and $[w_1, w_2]$ overlap as intervals. □

Definition 5.1.3. A dependency structure D is called *weakly non-projective*, if it does not contain overlapping edges. □

Example 5.1.1. The structure D_2 depicted in Figure 5.1c is not weakly non-projective: the edges $1 \to 3$ and $2 \to 4$ overlap. On the other hand, the structure D_1 depicted in Figures 5.1a and 5.1b does not contain overlapping edges; it is weakly non-projective. The difference between the two structures can be seen more clearly when drawing them with undirected edges as in Figure 5.2. □

A dependency structure that contains overlapping edges cannot be drawn without edges that cross a projection line. Conversely, the only non-projective edges $u \to v$ that a weakly non-projective dependency structure can contain are such that all ancestors of the node u and all nodes governed by these ancestors, except for the nodes also governed by u, fall into the interval $[u, v]$. Such structures can be redrawn by moving the material in the gap of the edge to vertical positions below the node u.



(a) D_1 (b) D_2

Fig. 5.2. Alternative pictures for the dependency structures from Figure 5.1

Example 5.1.2 (continued). The picture in Figure 5.1a contains one edge that crosses a projection line: the edge 1 → 4. When we take the set of all ancestors of 1 (the set $\{2\}$) and all nodes governed by these ancestors ($\{1, 2, 3, 4\}$), minus the nodes also governed by 1 ($\{1, 4\}$), we obtain the set $\{2, 3\}$. By moving these nodes to vertical positions below the node 1, we get to the picture in Figure 5.1b. □

5.1.2 Relation to the Block-Degree Measure

We now show that, from the perspective of the block-degree measure, weak non-projectivity is a weak extension of projectivity indeed: it does not even take us up one step in the block-degree hierarchy. We first prove an auxiliary lemma.

Lemma 5.1.1. *Let D be a weakly non-projective dependency structure. Then every gap in D contains the root node of D.* □

Proof. We show the contrapositive of the statement: If at least one gap in D does not contain the root node, then D is not weakly non-projective. Let D be a dependency structure with a gap that does not contain the root node. We can then choose four pairwise distinct nodes r, u, h, v as follows: Choose r to be the root node of D. Choose u to be a node such that G is a gap of u and $r \notin G$; since the root node does not have a gap, it is certain that $r \neq u$. Choose $h \in G$; we then have $h \notin \lfloor u \rfloor$ and $h \neq r$. Choose v to be a node governed by u such that $h \in (u, v)$; since the root node is governed only by itself, we have $v \neq r$. Based on the relative precedences of the nodes, we now distinguish four cases, shown schematically in Figure 5.3. In all four cases, some edges on the paths from r to h and from u to v overlap. ■

Note that the converse of Lemma 5.1.1 does *not* hold: there are dependency structures that are not weakly non-projective, but in which all gaps contain the root node. As an example, consider the structure D_2 depicted in Figure 5.1c.

Lemma 5.1.2. *Every weakly non-projective dependency structure has a block-degree of at most 2. Furthermore, there is at least one dependency structure with block-degree 2 that is not weakly non-projective.* □

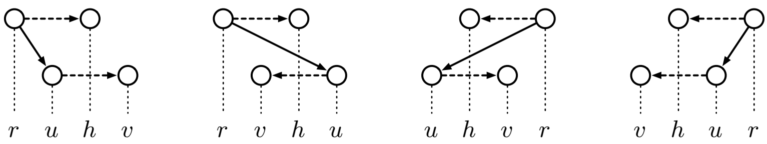


Fig. 5.3. The four cases in the proof of Lemma 5.1.1

Proof. To see the inclusion, let D be weakly non-projective. By Lemma 5.1.1, we know that every gap in D contains the root node. Since distinct gaps of one and the same node are set-wise disjoint, this implies that no node in D can have more than one gap. Therefore, the structure D has a block-degree of at most 2. To see that the inclusion is proper, consider again structure D_2 in Figure 5.1c: this structure has block-degree 2, but contains overlapping edges. ■

Weak non-projectivity appears like a very interesting constraint at first sight, as it expresses the intuition that 'crossing edges are bad', but still allows a limited form of non-projectivity. On the other hand, the property stated in Lemma 5.1.1 seems rather peculiar. To get a better understanding of the explanatory force of weak non-projectivity, we evaluate its empirical relevance in Section 5.3.

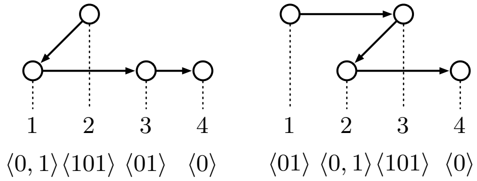
5.1.3 Algebraic Opaqueness

From the perspective of our algebraic setting, there is a fundamental difference between weak non-projectivity and the block-degree restriction. Recall from Section 4.3.2, that in order to check whether a dependency structure D has block-degree at most k, it suffices to check whether the corresponding term term(D) only contains symbols from the sub-signature $\Omega(k)$ of order annotations with degree at most k. In this sense, the block-degree measure is *transparent*: it is directly related to the set of composition operations used to build a structure. The class of weakly non-projective dependency structures cannot be characterized in the same way. To see this, consider Figure 5.4, which shows a weakly non-projective structure (D_3) and a structure that is not weakly non-projective (D_2). Both of these structures are composed using the same set of algebraic operations.

5.1.4 Related Work

In the Western literature, weak non-projectivity is more widely known as *planarity* [111]. Unfortunately, the latter term clashes with the concept of planarity known from graph theory, for at least two reasons: First, while a planar graph is a graph that can be drawn into the plane such that no edges intersect, a 'planar' dependency structure is a graph that is drawn into the *half plane* above the words of the sentence. Second, to show that a graph is planar in the graph-theoretic sense, its nodes may be rearranged on the plane in any arbitrary way; in the context of dependency structures, the order of the nodes is fixed. Due to these incompatibilities, it seems wise to avoid the term 'planarity', and use a less biased name instead.

Projectivity and weak non-projectivity are closely related. Some authors in fact *define* projectivity by requiring the weak non-projectivity of the structure that is obtained when the dependency structure proper is extended by an



(a) D_3

(b) D_2 (repeated)

Fig. 5.4. Weak non-projectivity is algebraically opaque

artificial root node, preceding all other nodes (see e.g. [76]). In Link Grammar for example, the artificial root node is called *the wall*, ‘an invisible word which the [parser] program inserts at the beginning of every sentence’ [110]. As a corollary of Lemma 5.1.1, every weakly non-projective dependency structure in which the root node occupies the leftmost position in the precedence order is projective. Another way to enforce the projectivity of weakly non-projective dependency structures is to require that no edge covers the root node [50, 77].

Yli-Jyrä [121] proposes a generalization of weak non-projectivity, and evaluates its empirical adequacy using data from the Danish Dependency Treebank. He calls a dependency structure *m-planar*, if its governance relation can be partitioned into m sets, called *planes*, such that each of the substructures induced by such a plane is weakly non-projective. Since every dependency structure is m-planar for some sufficiently large m (put each edge onto a separate plane), an interesting question in the context of multiplanarity is about the *minimal* values for m that occur in real-world data. To answer this question, one not only needs to show that a dependency structure *can* be decomposed into m weakly non-projective structures, but also, that this decomposition is the one with the smallest possible number of planes. Up to now, no tractable algorithm to find the minimal decomposition has been given, so it is not clear how to evaluate the significance of the concept as such. The evaluation presented by Yli-Jyrä [121] makes use of additional constraints that are sufficient to make the decomposition unique.

In combinatorics, weakly non-projective dependency structures are known as *non-crossing rooted trees*. Their number is given by sequence A001764 in Sloane [112].[1] Using Lemma 5.1.1, we see that every projective dependency structure can be decomposed into two halves—one with the root node at the right border, one with the root node at the left—such that each half is a non-crossing rooted tree. One can then obtain the number of projective dependency structures as the convolution of sequence A001764 with itself; this is sequence A006013 in Sloane [112].

[1] Sequence A001764 actually gives the number of non-crossing *unrooted* trees. In order to get the right numbers for weakly non-projective structures, one has to read the sequence with an offset of 1.

5.2 Well-Nested Dependency Structures

In this section, we develop the notion of *well-nestedness*. Well-nestedness is similar to weak non-projectivity in that it disallows certain ‘crossings’, but different in that it is transparent at the level of our algebraic signatures. Well-nestedness was introduced by Bodirsky, Kuhlmann, and Möhl [3], and subsequently studied in detail by Möhl [80].

5.2.1 Definition of Well-Nestedness

The definition of well-nestedness specializes the definition of weak non-projectivity in that it bans overlapping edges only if they belong to disjoint subtrees. Overlapping configurations in which one of the edges governs the other are allowed.

Definition 5.2.1. A dependency structure D is called *well-nested*, if the following implication holds for all edges $v_1 \to v_2$, $w_1 \to w_2$ in D:

$$[v_1, v_2] \between [w_1, w_2] \quad \Longrightarrow \quad v_1 \trianglelefteq w_1 \;\vee\; w_1 \trianglelefteq v_1 \,.$$

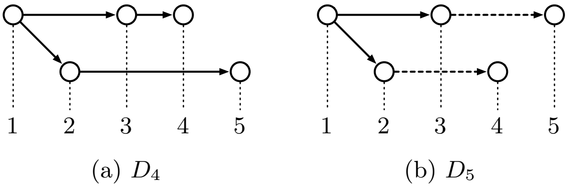
Dependency structures that are not well-nested are called *ill-nested*. □

We write $\mathcal{D}_{wn}$ for the class of all well-nested dependency structures. From the definition, it is straightforward that every weakly non-projective dependency structure is also well-nested. As the following example shows, the converse does not hold.

Example 5.2.1. Figure 5.5 shows pictures of two non-projective dependency structures. Structure D_4 is not weakly non-projective, as the edges $1 \to 3$ and $2 \to 5$ overlap; however, it is well-nested, as $1 \trianglelefteq 2$. Structure D_5 is not even well-nested: the spans $2 \to 4$ and $3 \to 5$ overlap, but 2 and 3 belong to disjoint subtrees. □

In contrast to weak non-projectivity, well-nestedness is independent of the block-degree measure: it is not hard to see that for every block-degree $k > 1$, there are both well-nested and ill-nested dependency structures of degree k. Projective structures are both weakly non-projective and well-nested. In summary, we obtain the following hierarchy of classes of dependency structures:

$$\text{projective} \subsetneq \text{weakly non-projective} \subsetneq \text{well-nested} \subsetneq \text{unrestricted}\,.$$



(a) D_4 (b) D_5

Fig. 5.5. Two dependency structures: one well-nested, the other one ill-nested

5.2.2 Non-crossing Partitions

Our next aim is to show that well-nestedness is algebraically transparent. To do so, we develop an alternative relational characterization of well-nestedness based on the notion of *non-crossing partitions.*

Definition 5.2.2. Let $\mathfrak{C} = (A\,;\preceq)$ be a chain. A partition Π of A is called *non-crossing*, if whenever there exist four elements $a_1 \prec b_1 \prec a_2 \prec b_2$ in A such that a_1 and a_2 belong to the same class of Π, and b_1 and b_2 belong to the same class of Π, then these two classes coincide. A partition that is not non-crossing is called *crossing.* □

Non-crossing partitions enjoy a number of interesting formal properties. In particular, the number of non-crossing partitions of a chain with n elements is the Catalan number, $C_n = \frac{1}{n+1}\binom{2n}{n}$, and by this property, non-crossing partitions are connected to a large family of mathematical structures—such as binary trees, Catalan paths in the plane, pattern-avoiding permutations, and (most important in the context of this study) well-bracketed strings and children-ordered trees. Consequently, non-crossing partitions appear in a large number of mathematical applications. Simion [109] provides a comprehensive overview.

Example 5.2.2. Consider the following partitions on the chain $([6]\,;\leq)$:

$$\Pi_1 = \{\{1\},\{2,3,4\},\{5,6\}\}\,, \qquad \Pi_2 = \{\{1\},\{2,3,6\},\{4,5\}\}\,,$$
$$\Pi_3 = \{\{1\},\{2,4,6\},\{3,5\}\}\,.$$

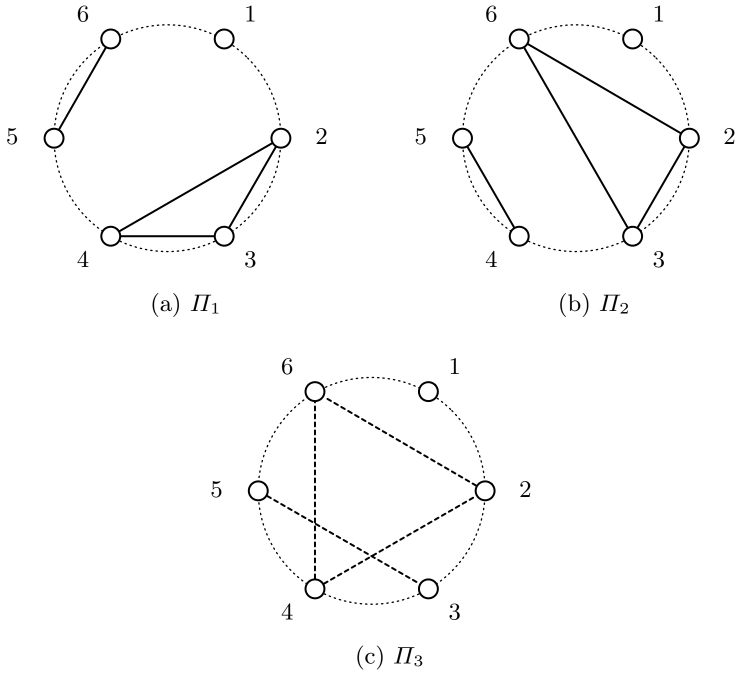
Both Π_1 and Π_2 are non-crossing. Partition Π_3 is crossing, as witnessed by the sequence $2 < 3 < 4 < 5$: the elements 2 and 4 and the elements 3 and 5 belong to the same class of Π_2, but these two classes do not coincide. □

Example 5.2.3. Let $n \in \mathbb{N}$. A neat way to visualize a partition Π on the canonical chain $([n]\,;\leq)$ goes as follows: Consider a regular n-gon inscribed into a circle, and assume that the points where the n-gon touches the circle are numbered clockwise from 1 to n. Now, for every class of Π of size k, connect the corresponding points on the circle with straight lines to form a convex k-gon. The partition Π is non-crossing if and only if no of these k-gons intersect. Figure 5.6 shows such pictures for the partitions from example 5.2.2. □

We now use non-crossing partitions to characterize well-nestedness.

Lemma 5.2.1. *A dependency structure D is well-nested if and only if for every node u of D, the set $\mathcal{C}(u)$ of constituents of u (see Definition 3.2.2) is non-crossing with respect to the chain $\mathfrak{C} := (\lfloor u \rfloor\,; \preceq\!|_{\lfloor u \rfloor})$.* □

Proof. We prove the contrapositive of the claim: a dependency structure D is ill-nested if and only if there exists a node $u \in \mathrm{dom}(D)$ such that the partition $\mathcal{C}(u)$ forms a crossing partition with respect to $\mathfrak{C}$. The proof falls into two parts.



(a) Π_1

(b) Π_2

(c) Π_3

Fig. 5.6. Two non-crossing and one crossing partition

- ⇒ Assume that D is ill-nested. In this case there exist overlapping edges $v_1 \to v_2$ and $w_1 \to w_2$ such that $v_1 \perp w_1$. Let u be the greatest (farthest from the root node) common ancestor of v_1 and w_1. The node sets $\{v_1, v_2\}$ and $\{w_1, w_2\}$ belong to the yields of distinct children of u, and hence, to distinct constituents of u. Furthermore, the intervals $[v_1, v_2]$ and $[w_1, w_2]$ overlap with respect to $\mathfrak{C}$. Thus we deduce that $\mathcal{C}(u)$ is crossing with respect to $\mathfrak{C}$.
- ⇐ Let $u \in \mathrm{dom}(D)$ be a node, and assume that the partition $\mathcal{C}(u)$ is crossing with respect to $\mathfrak{C}$. In this case, there exist two distinct constituents C_v and C_w in $\mathcal{C}(u)$ and elements $v_1, v_2 \in C_v$, $w_1, w_2 \in C_w$ such that $[v_1, v_2] \between [w_1, w_2]$. By the definition of $\between$, both C_v and C_w have a cardinality of at least 2; therefore, they correspond to the yields of distinct and hence disjoint children of the node u, say $C_v = \lfloor v \rfloor$ and $C_w = \lfloor w \rfloor$. For every arrangement of the nodes v_1, v_2 and w_1, w_2, we can choose edges $v_1' \to v_2'$ in $\lfloor v \rfloor$ and $w_1' \to w_2'$ in $\lfloor w \rfloor$ such that these edges overlap. Furthermore, by construction we have $v_1 \perp w_2$, and hence, $v_1' \perp w_1'$. Thus we deduce that D is ill-nested. ■

Example 5.2.4 (continued). Consider the constituents of the root nodes in Figure 5.5:

$$D_4 \;:\; \mathcal{C}(1) = \{\{1\},\{2,5\},\{3,4\}\} \qquad D_5 \;:\; \mathcal{C}(1) = \{\{1\},\{2,4\},\{3,5\}\}$$

The first of these partitions is crossing, the second non-crossing. □

Lemma 5.2.1 shows that by restricting ourselves to composition operations that arrange their arguments into non-crossing partitions, we produce exactly the well-nested dependency structures. In this sense, well-nestedness is a transparent property.

5.2.3 Algebraic Characterization

We now give an explicit characterization of the composition operations that generate the well-nested dependency structures. More specifically, we state a syntactic restriction on order annotations that identifies a sub-signature Ω_{wn} of Ω such that the dependency structures that are obtained as the values of the terms over Ω_{wn} are exactly the well-nested dependency structures. The syntactic restriction ensures that all constituents form non-crossing partitions.

Let $\vec{x} \in A^*$ be a string. We say that $\vec{x}$ contains the string $\vec{y}$ as a *scattered substring*, if, for some $k \in \mathbb{N}$, $\vec{x}$ can be written as

$$\vec{x} \;=\; \vec{z}_0 \cdot \vec{y}_1 \cdot \vec{z}_1 \cdots \vec{z}_{k-1} \cdot \vec{y}_k \cdot \vec{z}_k \,,$$

where $\vec{z}_0 \cdots \vec{z}_k \in A^*$, and $\vec{y}_1 \cdots \vec{y}_k = \vec{y}$.

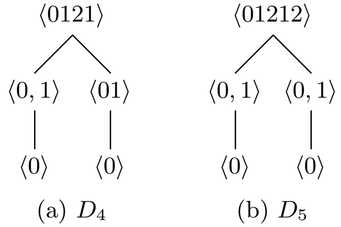
Definition 5.2.3. An order annotation $\omega \in \Omega$ is called *well-nested*, if it does not contain a string of the form $ij\,ij$ as a scattered substring, for $i \neq j \in \mathbb{N}$. □

We write Ω_{wn} for the set of all well-nested order annotations.

Example 5.2.5. The order annotation $\langle 0121 \rangle$ is well-nested, the annotation $\langle 01212 \rangle$ is not: it contains the string 1212 as a scattered substring. Figure 5.7 shows terms that make use of the two order annotations; these terms evaluate to the dependency structures shown in Figure 5.5. □

We now present the main result of this section:

Theorem 5.2.1. *A dependency structure D is well-nested if and only if term$(D) \in T_{\Omega_{wn}}$.* □



(a) D_4 (b) D_5

Fig. 5.7. Terms for the dependency structures in Figure 5.5

Proof. This is a corollary of Lemma 5.2.1. The presence of the scattered substring $ij\,ij$ implies that for some node u, there exist distinct constituents $C_1, C_2 \in \mathcal{C}(u)$ and nodes $v_1, v_2 \in C_1$, $w_1, w_2 \in C_2$ such that $v_1 \prec w_1 \prec v_2 \prec w_2$; then, $\mathcal{C}(u)$ would be crossing. Conversely, the encoding algorithm translates every constituent set that is crossing into an order annotation that contains the forbidden pattern. ■

5.2.4 Testing whether a Dependency Structure Is Well-Nested

Given that the class of well-nested dependency structures forms a subclass of the class of all dependency structures, the algorithmic problems of encoding and decoding can be solved using the algorithms that we have presented in Section 4.3. Here we address the problem of testing whether a given dependency structure D is well-nested.

Lemma 5.2.2. *Let D be a dependency structure with n nodes and g gaps. The question whether D is well-nested can be decided in time $O(n + g)$.* □

Proof. To check whether D is well-nested, we first encode D into a term using the algorithm presented in Section 4.3; this takes time $O(n+g)$. In a traversal over this term, we then check whether any of the order annotations contains the forbidden scattered substring (see Theorem 5.2.1); using a stack data structure, this can be done in time linear in the accumulated size of the order annotations, which is again $O(n + g)$. By Theorem 5.2.1, the structure D is well-nested if and only if we do not find the forbidden substring. ■

Similar to the situation in Lemma 4.3.2, we can also bound the complexity of the algorithm as $O(k \cdot n)$, where k is the block-degree of D.

5.2.5 Related Work

Nasr [82] proposes a restriction on non-projective dependency structures that he calls the *pseudo-projectivity principle*.[2] Formally, 'a dependency [edge] is pseudo-projective, if its dependent D is not situated, in the linear sequence, between two dependents of a node that is not an ancestor of D.' Pseudo-projectivity is incomparable with both weak non-projectivity and well-nestedness. To see this, consider the dependency structure in Figure 5.8. The edge 3 → 4 in this structure is not pseudo-projective, as the node 4 is situated between two dependents (1 and 5) of a node that is not an ancestor of 4 (the node 2). On the other hand, the structure is weakly non-projective, and therefore well-nested. The structure D_5 (Figure 5.5b) is not well-nested, but pseudo-projective.

[2] This principle should not be confused with the notion of pseudo-projectivity introduced by Kahane et al. [56], and subsequently used by Gerdes and Kahane [29].

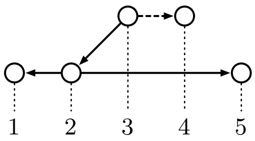


Fig. 5.8. Pseudo-projectivity in the sense of Nasr [82]

The first algorithms for a well-nestedness test were presented by Möhl [80]. His first algorithm is based on a characterization of well-nestedness in terms of *interleaving yields*. The algorithm performs a tree traversal of the input structure, in which it first computes the yield of each node, and then checks for each pair of sibling nodes whether their yields interleave. In doing so, it uses $O(m^2 \cdot n)$ operations on bit vectors, where m is the out-degree of D. Möhl's second algorithm is built on the notion of the *gap graph*. The gap graph for a dependency structure is an extension of the governance relation by information about interleaving yields. Möhl shows that a dependency structure D is well-nested if and only if the gap graph for D contains a cycle. The size of the gap graph for a dependency structure D is tightly bounded by the square of the size of D, and the existence of a cycle in a graph can be checked in time linear in the size of that graph. Consequently, the run-time of Möhl's second algorithm is $O(n^2)$.

Havelka [45] studies the relationship between well-nestedness and the *level types* of non-projective edges [43] and presents an algorithm that tests for well-nestedness in time $O(n^2)$.

5.3 Empirical Evaluation

To conclude this chapter, we now evaluate and compare the empirical adequacy of weak non-projectivity and well-nestedness on the treebank data. The corresponding counts and percentages are given in Table 5.1.

The experimental results for weak non-projectivity mirror its formal restrictiveness: enforcing weak non-projectivity excludes more than 75% of the non-projective data in both versions of the PDT, and 90% of the data in the DDT. Given these figures, weak non-projectivity appears to be of little use as a generalization of projectivity. The relatively large difference in coverage between the two treebanks may at least partially be explained with their different annotation schemes for sentence-final punctuation: In the DDT, sentence-final punctuation marks are annotated as dependents of the main verb of a dependency subtree. This places severe restrictions on permitted forms of non-projectivity in the remaining sentence, as every discontinuity that includes the main verb must also include the dependent punctuation marks (see the discussion in Section 5.1). On the other hand, in the PDT, a sentence-final punctuation mark is annotated as a separate root node with no dependents.

Table 5.1. The number of weakly non-projective and well-nested dependency structures in three treebanks

All dependency structures

	DDT		PDT 1.0		PDT 2.0	
projective	3 730	84.95%	56 168	76.85%	52 805	77.02%
weakly non-proj.	3 794	86.40%	60 048	82.16%	56 367	82.21%
well-nested	4 386	99.89%	73 010	99.89%	68 481	99.88%
TOTAL	4 391	100.00%	73 088	100.00%	68 562	100.00%

Non-projective dependency structures only

	DDT		PDT 1.0		PDT 2.0	
weakly non-proj.	64	9.68%	3 880	22.93%	3 562	22.61%
well-nested	597	90.32%	16 842	99.54%	15 676	99.49%
TOTAL	661	100.00%	16 920	100.00%	15 757	100.00%

(Analyses in the PDT may be forests.) This scheme does not restrict the remaining discontinuities at all.

In contrast to weak non-projectivity, the well-nestedness constraint appears to constitute a very attractive extension of projectivity. For one thing, the almost perfect coverage of well-nestedness on both DDT and PDT (around 99.89%) could by no means be expected on purely combinatorial grounds: only 7% of all possible dependency structures for sentences of length 17 (the average sentence length in the PDT), and only slightly more than 5% of all possible dependency structures for sentences of length 18 (the average sentence length in the DDT) are well-nested.[3] Similar results have been reported on other data sets [44]. Moreover, a cursory inspection of the few problematic cases at least in the DDT indicates that violations of the well-nestedness constraint may, at least in part, be due to properties of the annotation scheme, such as the analysis of punctuation in quotations. However, a more detailed analysis of the data from both treebanks is needed before any stronger conclusions can be drawn concerning well-nestedness.

[3] The number of unrestricted dependency structures on n nodes is given by sequence A000169, the number of well-nested dependency structures is given by sequence A113882 in [112]. The latter sequence was discovered by the author and Manuel Bodirsky. It can be calculated using a recursive formula derivable from the correspondence indicated in Example 5.2.3.

6

Structures and Grammars

In the last three chapters, we have developed an algebraic framework for dependency structures. We now put this framework to use and classify several lexicalized grammar formalisms with respect to the classes of dependency structures that are induced by derivations in these formalisms. Each section of this chapter associates a grammar formalism with a class of dependency structures:

Section	*Formalism*	*Class*
6.1	Context-Free Grammar (CFG)	$\mathcal{D}_1$
6.2	Linear Context-Free Rewriting Systems (LCFRS(k))	$\mathcal{D}_k$
6.3	Coupled Context-Free Grammar (CCFG(k))	$\mathcal{D}_k \cap \mathcal{D}_{wn}$
6.4	Tree Adjoining Grammar (TAG)	$\mathcal{D}_2 \cap \mathcal{D}_{wn}$

6.1 Context-Free Grammars

Let us go back to the notion of *induction* that we sketched in Chapter 1. Consider a derivation d of a terminal string by means of a context-free grammar. A *derivation tree* for d is a tree in which the nodes are labelled with (occurrences of) the productions used in the derivation, and the edges indicate how these productions were combined. If the underlying grammar is lexicalized, then there is a one-to-one correspondence between the nodes in the derivation tree and the positions in the derived string: every production that participates in the derivation contributes exactly one terminal symbol to this string. If we now order the nodes of the derivation tree according to the string positions of their corresponding anchors, then we get a dependency structure. We say that this dependency structure is *induced* by the derivation d. Induction identifies the governance relation of the induced dependency structure with the derivation relation, and the precedence relation with the left-to-right order in the derived string: the dependency structure contains an edge $u \to v$ if and only if the production that corresponds to the node v was

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 63–83, 2010.

used to rewrite some non-terminal in the production that corresponds to the node u; the node u precedes the node v if and only if the anchor contributed by the production that corresponds to u precedes the anchor contributed by the production that corresponds to v. In this section, we formalize the correspondence between derivations and induced dependency structures and show that the class of dependency structures that can be induced by context-free grammars is exactly the class of projective dependency structures.

6.1.1 Definition

We start with the familiar definition of a context-free grammar.

Definition 6.1.1. A *context-free grammar* is a construct $G = (N, T, S, P)$, where N and T are alphabets of *non-terminal* and *terminal symbols*, respectively, $S \in N$ is a distinguished *start symbol*, and $P \subseteq N \times (N \cup T)^*$ is a finite set of productions. □

We use indexed symbols (N_G, T_G, S_G, P_G) to refer to the components of a specific context-free grammar G.

Example 6.1.1. To illustrate the ideas and constructions of this section, we use the following grammar $G = (N, T, S, P)$ as a running example:

$$N = \{S, B\}\,, \quad T = \{a, b\}\,, \quad P = \{S \to aSB,\, S \to aB,\, B \to b\}\,.$$

This grammar generates the string language $\{\, a^n b^n \mid n \in \mathbb{N} \,\}$. □

Following the approach of Goguen et al. [31], we treat context-free grammars as many-sorted algebras, in the following way. Let $G = (N, T, S, P)$ be a context-free grammar. For every string $\vec{x} \in (N \cup T)^*$, we define $\mathrm{res}_N(\vec{x})$ to be the restriction of $\vec{x}$ to letters in N. Specifically, res_N is the homomorphism from $(N \cup T)^*$ to N^* that is defined by $\mathrm{res}_N(A) := A$ for $A \in N$, and $\mathrm{res}_N(a) := \varepsilon$ for $a \in T$. We now turn the set P of productions of G into an N-sorted set $\Sigma(G)$ by defining

$$\mathrm{type}_{\Sigma(G)}(A \to \vec{x}) \quad := \quad \mathrm{res}_N(\vec{x}) \cdot A\,,$$

for every production $A \to \vec{x}$ in P. The set $T_{\Sigma(G)}$ of terms over the sorted set $\Sigma(G)$ forms an N-sorted algebra. This algebra represents the set of all derivations of G: the sortedness enforces the ith child of a node labelled with a production p to be labelled with a production that can be used to rewrite the ith non-terminal in p. More formally, there is a one-to-one correspondence between $T_{\Sigma(G)}$ and the set of all leftmost derivations in G. The set $T_{\Sigma(G),S}$ of all terms of sort S (the start symbol of the grammar) then corresponds to the set of all complete derivations in G. We call $T_{\Sigma(G)}$ the *derivation algebra* for G, and the terms $T_{\Sigma(G)}$ the *derivation trees* of G.

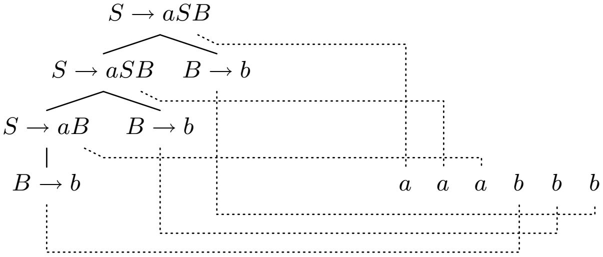


Fig. 6.1. A derivation in a lexicalized context-free grammar

Example 6.1.2 (continued). For our example grammar, we get the following sorted set:

$$[S \to aSB]\colon S \times B \to S\,, \qquad [S \to aB]\colon B \to S\,, \qquad [B \to b]\colon B\,.$$

(We enclose productions in square brackets to avoid ambiguities.) A derivation tree of the grammar is shown in the left half of Figure 6.1. □

Definition 6.1.2. A context-free grammar is called *lexicalized*, if each of its productions contains exactly one terminal symbol, called the *anchor* of that production. □

We only consider lexicalized grammars. Each production in such a grammar has the form $A \to A_1 \cdots A_{k-1} \cdot a \cdot A_k \cdots A_m$, for some $m \in \mathbb{N}$ and $k \in [m+1]$. Note that our example grammar is lexicalized.

6.1.2 String Semantics

An immediate benefit of our algebraic take on context-free grammars is that we can use every $\Sigma(G)$-algebra $\mathfrak{A}$ as a semantic domain for the derivations of G: since $T_{\Sigma(G)}$ is a term algebra, it gives us the unique homomorphism $[\![\cdot]\!]_{\mathfrak{A}}\colon T_{\Sigma(G)} \to \mathfrak{A}$ that evaluates the derivation trees of G in $\mathfrak{A}$. In this way it is straightforward to derive the usual notion of the string language generated by a grammar G:

Definition 6.1.3. Let G be a context-free grammar. The *string algebra* for G is the $\Sigma(G)$-algebra $\mathfrak{A}$ in which $\mathrm{dom}(\mathfrak{A})_A = T_G^+$, for all $A \in N_G$, and

$$f_p(\vec{a}_1, \dots, \vec{a}_m) \quad = \quad \vec{a}_1 \cdots \vec{a}_{k-1} \cdot a \cdot \vec{a}_k \cdots \vec{a}_m\,,$$

for each production $p = A \to A_1 \cdots A_{k-1} \cdot a \cdot A_k \cdots A_m$ in $\Sigma(G)$. The *string language* generated by G is the set $L(G) := [\![T_{\Sigma(G),S_G}]\!]_{\mathfrak{A}}$. □

Each composition operation f_p of the string algebra for a grammar G concatenates the anchor of p and the strings obtained from the subderivations in the order specified by the production p. This implements the usual rewriting semantics for context-free grammars. In the following, given a derivation tree $t \in T_{\Sigma(G)}$ of some CFG G, we write $[\![t]\!]_{\mathbf{S}}$ for the evaluation of that tree in the string algebra for G. We also extend this notation to sets of derivation trees in the obvious way.

Example 6.1.3 (continued). The right half of Figure 6.1 shows the string corresponding to the evaluation of the derivation that is shown in the left half in the string algebra for our example grammar. □

6.1.3 Linearization Semantics

In a lexicalized context-free grammar, there is a one-to-one correspondence between the nodes in a derivation t and the positions of the string $[\![t]\!]_{\mathbf{S}}$: every production participating in a derivation contributes exactly one terminal to the derived string.

Example 6.1.4 (continued). In Figure 6.1, the one-to-one correspondence between the nodes of the derivation tree and the positions of the string is indicated by dashed lines. □

We now show how to compute the mapping between nodes in the derivation and positions in the derived string that forms the basis of our notion of induction. To do so, we evaluate derivations t not in the string algebra, but in an algebra of *term linearizations*. A term linearization is a list of the nodes of a term in which each node occurs exactly once. In the following, we write $V := \mathbb{N}^*$ for the set of all nodes in terms; V^* then stands for the set of all strings over nodes. To avoid ambiguity, we use the symbol $\circ$ for the concatenation operation on $\mathbb{N}$ (which builds nodes), and $\cdot$ for the concatenation operation on V (which builds strings of nodes). For every $i \in \mathbb{N}$, let pfx_i be the function that prefixes every node u in a given string by the number i. More formally, pfx_i is the string homomorphism from V to V that is defined by $\mathrm{pfx}_i(u) = i \circ u$.

Definition 6.1.4. Let G be a context-free grammar. The *linearization algebra* for G is the $\Sigma(G)$-algebra $\mathfrak{A}$ in which $\mathrm{dom}(\mathfrak{A})_A = V^+$, for all $A \in N_G$, and

$$f_p(\vec{u}_1, \dots, \vec{u}_m) \quad = \quad \mathrm{pfx}_1(\vec{u}_1) \cdots \mathrm{pfx}_{k-1}(\vec{u}_{k-1}) \cdot \varepsilon \cdot \mathrm{pfx}_k(\vec{u}_k) \cdots \mathrm{pfx}_m(\vec{u}_m)\,,$$

for each production $p = A \to A_1 \cdots A_{k-1} \cdot a \cdot A_k \cdots A_m$ in $\Sigma(G)$. The *linearization language* generated by G is the set $\Lambda(G) := [\![T_{\Sigma(G),S_G}]\!]_{\mathfrak{A}}$. □

Each composition operation f_p of a linearization algebra concatenates a root node (representing the anchor of p) and the appropriately prefixed linearizations for the subderivations in the same order as they would be concatenated in the string algebra. Since the grammar G is lexicalized, the result of the evaluation in the linearization algebra defines a bijection between the set $\mathrm{nod}(t)$

of nodes in the derivation tree t, and the set $\mathrm{pos}([\![t]\!]_{\mathbf{S}})$ of positions in the derived string. Similar to the case of string algebras, for a derivation tree $t \in T_{\Sigma(G)}$ of some CFG G, we write $[\![t]\!]_{\mathbf{L}}$ for the evaluation of the tree t in the linearization algebra for G.

Example 6.1.5 (continued). For the derivation tree t shown in Figure 6.1,

$$[\![t]\!]_{\mathbf{L}} \;=\; \varepsilon \cdot 1 \cdot 11 \cdot 111 \cdot 12 \cdot 2 \,.$$

This linearization defines a mapping from the set $\mathrm{nod}(t)$ of nodes in t to the set $\mathrm{pos}([\![t]\!]_{\mathbf{S}})$ of positions in the derived string and back in the obvious way. Notice that, if we read the anchors of the productions in the derivation tree in the order specified by this linearization, then we obtain the string $[\![t]\!]_{\mathbf{S}}$. □

6.1.4 Dependency Semantics

With the mapping between nodes in the derivation tree and positions in the derived string at hand, we can now formalize the notion of induction:

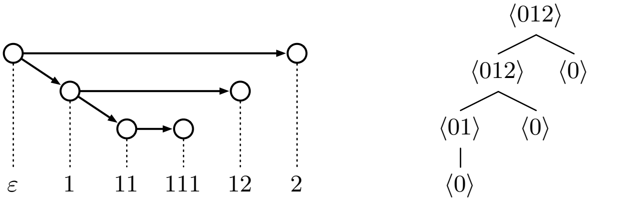
Definition 6.1.5. Let G be a context-free grammar, and let $t \in T_{\Sigma(G)}$ be a derivation tree. The dependency structure *induced* by t is the structure $D := (\mathrm{nod}(t)\,; \trianglelefteq, \preceq)$ where

$$\begin{array}{lll} u \trianglelefteq v & \text{if and only if} & u \text{ dominates } v \text{ in } t\text{, and} \\ u \preceq v & \text{if and only if} & u \text{ precedes } v \text{ in } [\![t]\!]_{\mathbf{L}}. \end{array}$$

□

Example 6.1.6. Figure 6.2a shows the dependency structure induced by the derivation given in Figure 6.1. To illustrate the correspondence with the linearization, we have labelled the nodes with their addresses in the derivation tree. □

It is straightforward that the linearization semantics of derivations in CFGs directly mirrors our procedure for the traversal of treelet-ordered trees from Section 3.2.2. More specifically, we can understand the productions of a CFG as order annotations, and each derivation tree as a treelet-ordered tree. The



(a) induced dependency structure (b) relabelled term

Fig. 6.2. The dependency structure induced by the derivation in Figure 6.1

behaviour of TREELET-ORDER-COLLECT is reflected in the evaluation of derivation trees in their corresponding linearization algebras. Taken together, this allows us to show that the class of dependency structures that is induced by lexicalized context-free grammars is exactly the class of projective dependency structures. In the following, we write $\mathcal{D}(\mathrm{CFG})$ for the class of all dependency structures that can be induced by a context-free grammar.

Theorem 6.1.1. $\mathcal{D}(\mathit{CFG}) = \mathcal{D}_1$ □

Proof. The proof of this statement is essentially identical to the proofs of Lemma 3.2.1 and Lemma 3.2.2. ■

On a formal level, the translation between derivation trees of a CFG and projective dependency structures (represented by their corresponding terms) can be understood as a simple form of *relabelling*. Specifically, we can replace each production by an order annotation as follows, while maintaining the typing information:

$$A \to A_1 \cdots A_{k-1} \cdot a \cdot A_k \cdots A_m \qquad \leftrightarrow \qquad \langle 1 \cdots (k-1) \cdot 0 \cdot k \cdots m \rangle \,.$$

The two terms are equivalent with respect to their linearization semantics. The only information that we loose is the label of the anchor, but that information is irrelevant with respect to induction anyway. (In Chapter 8, we will consider labelled dependency structures, where this information can be preserved.) As the essence of our discussion, we can define a dependency semantics for context-free grammars as follows. For each production p of a given context-free grammar G, let us write $\mathrm{relab}(p)$ for the relabelling defined above.

Definition 6.1.6. Let G be a context-free grammar. The *dependency algebra* for G is the $\Sigma(G)$-algebra $\mathfrak{D}$ in which $\mathrm{dom}(\mathfrak{D})_A = \mathcal{D}_1$, for all $A \in N_G$, and

$$f_p(D_1, \dots, D_m) \;=\; \mathrm{dep}((\mathrm{relab}(p))(\mathrm{term}(D_1), \dots, \mathrm{term}(D_m)))\,,$$

for each production $p = A \to A_1 \cdots A_{k-1} \cdot a \cdot A_k \cdots A_m$. The *dependency language* generated by G is the set $\mathcal{D}(G) := [\![T_{\Sigma(G),S_G}]\!]_{\mathfrak{D}}$. □

6.2 Linear Context-Free Rewriting Systems

We now extend our results from context-free grammars to the class of Linear Context-Free Rewriting Systems, LCFRS [118, 119]. This class was proposed as a generalization of a broad range of mildly context-sensitive formalisms. In this section, we show that the dependency structures induced by LCFRS are exactly the dependency structures of bounded degree. More specifically, we see that the block-degree measure for dependency structures is the structural correspondent of the *fan-out* measure that is used to identify sub-classes of LCFRS.

6.2.1 Definition

Linear Context-Free Rewriting Systems can be understood as generalizations of context-free grammars in which derivations evaluate to tuples of strings. Our formal definition of LCFRS is essentially the same as the definitions proposed by Vijay-Shanker et al. [118] and Satta [102]. In contrast to these, we make use of an explicit typing regime; this will simplify both the presentation and our formal arguments.

Definition 6.2.1. Let A be an alphabet, and let $m \in \mathbb{N}$, and $\langle\, k_i \mid i \in [m]\,\rangle \in \mathbb{N}^m$, and $k \in \mathbb{N}$. A *generalized concatenation function* over A of type $k_1 \times \cdots \times k_m \to k$ is a function

$$f\colon\ (A^*)^{k_1} \times \cdots \times (A^*)^{k_m} \to (A^*)^k$$

that can be defined by an equation of the form

$$f(\langle x_{1,1}, \ldots, x_{1,k_1}\rangle, \ldots, \langle x_{m,1}, \ldots, x_{m,k_m}\rangle) \;=\; \langle \vec{y}_1, \ldots, \vec{y}_k\rangle\,,$$

where $\vec{y}_1 \cdots \vec{y}_k$ is a string over the variables on the left-hand side of the equation and the alphabet A in which each variable $x_{i,j}$, $i \in [m]$, $j \in [k_i]$, appears exactly once. □

The semantics of a generalized concatenation function of type $k_1\times\cdots\times k_m \to k$ is that it takes m tuples of strings and arranges the components of these tuples and a constant number of symbols from the alphabet A into a new k-tuple. The arity of the ith argument tuple is specified by the sort k_i. We regard generalized concatenation functions as syntactic objects and identify them with their defining equations. We call the right-hand sides of these equations the *bodies* of the (defining equations of) the corresponding generalized concatenation functions.

Definition 6.2.2. A *linear context-free rewriting system* is a construct $G = (N, T, S, P)$, where N is an alphabet of *non-terminal symbols*, each of which is associated with a number $\varphi(A) \in \mathbb{N}$ called the *fan-out* of A; T is an alphabet of *terminal symbols*; $S \in N$ is a distinguished *start symbol* with $\varphi(S) = 1$; and P is a finite set of productions of the form $A \to f(A_1, \ldots, A_m)$, $m \in \mathbb{N}$, where $A, A_i \in N$, $i \in [m]$, and f is a generalized concatenation function over T of type $\varphi(A_1) \times \cdots \times \varphi(A_m) \to \varphi(A)$. □

Example 6.2.1. The following productions define an LCFRS. We use this LCFRS as our running example in this section.

$$\begin{array}{lcllcl}
S & \to & f_1(A) & f_1(\langle x_{1,1}\rangle) & := & \langle x_{1,1}b\rangle \\
S & \to & f_2(A,B) & f_2(\langle x_{1,1}\rangle, \langle x_{2,1}, x_{2,2}\rangle) & := & \langle x_{1,1}x_{2,1}bx_{2,2}\rangle \\
A & \to & f_3 & f_3 & := & \langle a\rangle \\
B & \to & f_4(A,B) & f_4(\langle x_{1,1}\rangle, \langle x_{2,1}, x_{2,2}\rangle) & := & \langle x_{1,1}x_{2,1}, bx_{2,2}\rangle \\
B & \to & f_5(A) & f_5(\langle x_{1,1}\rangle) & := & \langle x_{1,1}, b\rangle
\end{array}$$

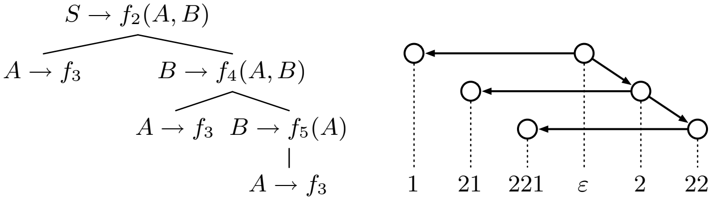
This LCFRS generates the string language $\{\, a^n b^n \mid n \in \mathbb{N}\,\}$. □

Our algebraic view on grammars generalizes to LCFRS without greater problems. Let $G = (N, T, S, P)$ be an LCFRS. We turn the set P of productions of G into a sorted set $\Sigma(G)$ by defining $\text{type}_{\Sigma(G)}(A \to f(A_1, \ldots, A_m)) := A_1 \cdots A_m \cdot A$, for every production $A \to f(A_1, \ldots, A_m)$ in P. Just as in the case of context-free grammars, the set of all terms over $\Sigma(G)$ forms an N-sorted algebra that represents the set of all derivation trees of G.

Example 6.2.2 (continued). For our example grammar, we get the following set $\Sigma(G)$:

$$[S \to f_1(A)]\colon A \to S\,, \quad [S \to f_2(A, B)]\colon A \times B \to S\,, \quad [A \to f_3]\colon A\,,$$
$$[B \to f_4(A, B)]\colon A \times B \to B\,, \quad [B \to f_5(A)]\colon A \to B\,.$$

A corresponding derivation tree is shown in Figure 6.3a. □



(a) derivation tree (b) dependency structure

Fig. 6.3. A derivation tree of an LCFRS, and the induced dependency structure

The concept of lexicalization is as for context-free grammars:

Definition 6.2.3. An LCFRS is called *lexicalized*, if each of its productions contains exactly one terminal symbol. □

6.2.2 String Semantics

We now give LCFRS their usual string semantics. The string language generated by an LCFRS can be defined in terms of the evaluation of its derivation trees in an algebra over tuples of strings over the terminal alphabet. The arity of these tuples is specified by the fan-out of the non-terminal symbols that derive them. In the following, we use the Greek letters α and γ for tuple components.

Definition 6.2.4. Let $G = (N, T, S, P)$ be an LCFRS. The *string algebra* for G is the $\Sigma(G)$-algebra $\mathfrak{A}$ in which $\text{dom}(\mathfrak{A})_A = (T^*)^k$, for all $A \in N$ and $k = \varphi(A)$, and

$$f_p(\vec{\alpha}_1, \ldots, \vec{\alpha}_m) \;=\; \vec{\gamma}[\, x_{i,j} \leftarrow \alpha_{i,j} \mid i \in [m] \wedge j \in [k_i]\,]\,,$$

for each production $p = A \to f(A_1, \ldots, A_m)$ with $f\colon k_1 \times \cdots \times k_m \to k$ and body $\vec{\gamma}$. The *string language* generated by G is the set

$$L(G) \;:=\; \{\, \vec{a} \mid \exists t \in T_{\Sigma(G),S}.\ \langle \vec{a} \rangle \in [\![t]\!]_{\mathfrak{A}} \,\}\,. \qquad \square$$

Each composition operation f_p of the string algebra uses the body of the production p to construct a new tuple of strings. This tuple is obtained by replacing, for every $i \in [m]$ and $j \in [k_i]$, the variable $x_{i,j}$ with the jth component of the ith argument tuple. The string language generated by the LCFRS is obtained by extracting the strings from the (necessarily unitary) tuples derived from the start symbol of the grammar. We use the notation $[\![t]\!]_{\mathbf{S}}$ as for context-free grammars.

6.2.3 Non-essential Concatenation Functions

Unfortunately, the construction of the linearization semantics of LCFRS does not go through as smoothly as in the case of CFG. The fundamental problem is the fact that the mapping from derivation trees to strings is not injective in the case of LCFRS—to phrase it as a slogan, in LCFRS there are 'more derivations than denotations'. At the root of this problem we find two types of ambiguity in LCFRS:

The first type of ambiguity concerns concatenation functions like

$$f \;=\; \langle a, \varepsilon \rangle \qquad \text{and} \qquad f(\langle x, y \rangle) = \langle axy \rangle\,.$$

Such function definitions cannot be translated into order annotations. In fact, we specifically excluded them in our definition of block-ordered trees (page 39) because they are essentially superfluous from the perspective of the linearization semantics: in the first function, the second component of the result does not contain any string material and could have been omitted to begin with; in the second definition, the two argument components are concatenated and would not have needed to be distributed over two components from the start.

The second ambiguity is inherent in the syntactic definition of LCFRS. In each derivation tree of a CFG, the left-to-right order on the subterms corresponds the left-to-right order on the material derived in these subterms. In LCFRS, this is not necessarily the case. Consider the following productions:

$$\begin{array}{lrcl} A \to f_1(B, C) & f_1(\langle x_{1,1} \rangle, \langle x_{2,1} \rangle) & := & \langle a x_{1,1} x_{2,1} \rangle \\ A \to f_2(C, B) & f_2(\langle x_{1,1} \rangle, \langle x_{2,1} \rangle) & := & \langle a x_{2,1} x_{1,2} \rangle \end{array}$$

For each pair of strings $\vec{a}_1, \vec{a}_2$, both of the following two derivations evaluate to the same string $\vec{a}_1 \cdot \vec{a}_2$:

$$[A \to f_1(B, C)](\langle \vec{a}_1 \rangle, \langle \vec{a}_2 \rangle)\,, \qquad [A \to f_2(C, B)](\langle \vec{a}_2 \rangle, \langle \vec{a}_1 \rangle)\,.$$

Let us call the body of a concatenation function $f\colon k_1 \times \cdots \times k_m \to k$ *monotone*[1], if 1. the variables of the form $x_{i,1}$, $i \in [m]$, are numbered in increasing

[1] A slightly weaker condition of the same name is discussed by 62, p. 408.

sequence from left-to-right, such that the ith occurrence of such a variable has the name $x_{i,1}$; 2. for each $i \in [m]$, the variables of the form $x_{i,j}$, $j \in [k_i]$, are numbered in increasing sequence from left-to-right, such that the jth occurrence of such a variable has the name $x_{i,j}$. By requiring every definition of a concatenation function to be monotone, we can avoid the order ambiguity of our example: we simply disallow the definition of the function f_2.

Let us call concatenation functions that fall into one of the two classes described above *non-essential*. We will now state a lemma (and give a rather technical and arduous proof) that will allow us to assume without loss of generality that an LCFRS is free of non-essential functions. For those functions that remain, the definitions of a linearization semantics go through without further problems.

To prove the lemma, we make use of more complicated versions of the relabelling function that we used for context-free grammars. Such a general relabelling is a function from terms over some input alphabet Σ into terms over an output alphabet Δ; it works by replacing, at each node u of the term, the label from the input alphabet by some label from the output alphabet. Both labels must have the same type. The choice of the output label is conditioned on the input label, and on some finite-state information coming from the children (bottom-up) or the parent (top-down) of u. Our formal definition in addition allows the relabelling to perform a permutation of subterms.

Definition 6.2.5. A *bottom-up relabelling* is a construct $M = (Q, \Sigma, \Delta, R)$, where Q is a finite set of *states*, Σ and Δ are sorted alphabets of *input* and *output symbols*, respectively, and R is a finite set of rules such that for every symbol $\sigma{:_\Sigma}\ s_1 \times \cdots \times s_m \to s$ and all states $q_1, \dots, q_m \in Q$, R contains exactly one rule of the form

$$\sigma(\langle q_1, x_1\rangle, \dots, \langle q_m, x_m\rangle) \quad\to\quad \langle q, \delta(x_{\pi(1)}, \dots, x_{\pi(m)})\rangle\,,$$

where $q \in Q$, $\delta{:_\Delta}\ s_1 \times \cdots \times s_m \to s$, and π is some permutation $[m] \to [m]$. Dually, a *top-down relabelling* has rules of the form

$$\langle q, \sigma(x_1, \dots, x_m)\rangle \quad\to\quad \delta(\langle q_{\pi(1)}, x_{\pi(1)}\rangle, \dots, \langle q_{\pi(m)}, x_{\pi(m)}\rangle)\,. \qquad \square$$

The *derivation relation* induced by a bottom-up relabelling is the binary relation on the set $T_{\langle Q, T_\Delta\rangle \cup \Sigma}$ that is obtained by interpreting the rules of the relabelling as rewriting rules in the obvious way.[2] Similarly, a top-down relabelling gives rise to a relation on the set $T_{\langle Q, T_\Sigma\rangle \cup \Delta}$. In the proof of the next lemma, we will give (informal) descriptions of relabellings by specifying their state sets and explaining the behaviour of their translation rules.

Lemma 6.2.1. *Let $k \in \mathbb{N}$. For each lexicalized LCFRS $G \in$ LCFRS(k), there exists a lexicalized LCFRS $G' \in$ LCFRS(k) such that the derivation trees of G and G' are isomorphic modulo relabelling, $[\![G]\!]_{\mathbf{S}} = [\![G']\!]_{\mathbf{S}}$, and G' only contains essential concatenation functions.* □

[2] For details, see [22, 23].

Proof. Let $k \in \mathbb{N}$, and let $G \in \mathrm{LCFRS}(k)$ be a lexicalized LCFRS. Furthermore, assume that the rank of G is bounded by m, for some $m \in \mathbb{N}$. We define three relabellings on the set of derivation trees of G that transform each derivation tree t into a term over an alphabet Δ that does not contain the non-essential functions.

The first transformation is a top-down relabelling. As the set of states, use the set of all permutations $\pi\colon\ [k'] \to [k']$, for $k' \in [k]$, and start with the identity permutation on the set $[1]$. Given a state q and a production p with body $\vec{\alpha}$, replace $\vec{\alpha}$ by the tuple $\vec{\alpha}'$ that is obtained from $\vec{\alpha}$ by permuting the components of $\vec{\alpha}$ according to the permutation q, and re-index the variables in p from left to right. Let $\pi\colon\ [m] \to [m]$ be the permutation that maps each $i \in [m]$ to the position of the variable $x_{i,1}$ from $\vec{\alpha}$ in the total order on all variables of the form $x_{j,1}$ in $\vec{\alpha}'$, $j \in [m]$. For each $i \in [m]$, let $q_i\colon\ [k_i] \to [k_i]$ be the permutation that maps each $j \in [k_i]$ to the position of the variable $x_{i,j}$ from $\vec{\alpha}$ in the total order on all variables of the form $x_{i,j'}$ in $\vec{\alpha}'$, $j' \in [k_i]$.

The second transformation is a top-down relabelling. As the set of states, use the set of all subsets of $[k]$. Start with the empty set. Given a state q and a production p with body $\vec{\alpha}$, replace $\vec{\alpha}$ by the tuple $\vec{\alpha}'$ that is obtained from $\vec{\alpha}$ by (i) merging, for every $i \in q$, the ith component of $\vec{\alpha}$ with the $(i+1)$st component, and (ii) deleting, for every $i \in [m]$ and $j \in [k_i]$, each maximal substring of variables of the form $x_{i,j'}$, $j < j' \leq k_i$, that is adjacent to $x_{i,j}$; then re-index the variables of p from left to right. For each $i \in [m]$ let q_i be the set of all indices $j \in [k_i]$ such that the variable $x_{i,j}$ was removed in step (ii), and let π be the identity permutation.

The third transformation is a bottom-up relabelling. As the set of states, use the set of all subsets of $[m'] \times [k']$, for $m' \in [m]$, $k' \in [k]$. Start with the empty set. Given a production p with body $\vec{\alpha}$ and states $\langle\, q_i \mid i \in [m]\,\rangle$ for the subterms, replace $\vec{\alpha}$ by the tuple $\vec{\alpha}'$ that is obtained from $\vec{\alpha}$ by (i) deleting all variables $x_{i,j}$, where $i \in [m]$ and $j \in q_i$, and (ii) deleting all empty components; then, re-index the variables of p from left-to-right. Let $\pi\colon\ [m] \to [m]$ be the permutation that maps each $i \in [m]$ to the position of the variable $x_{i,1}$ from $\vec{\alpha}$ in the total order on all variables of the form $x_{j,1}$ in $\vec{\alpha}'$, $j \in [m]$. Let q be the set of indices of all components deleted in step (ii).

None of these transformations alters the term structure of the original derivation tree (apart from the permutation of subterms, which is inessential with respect to tree-isomorphism), or the string derived from the tree: by induction on the derivation tree t, we can verify that its string semantics remain invariant under the relabelling. ∎

6.2.4 Linearization Semantics

With all non-essential concatenation functions out of our way, we are now ready to define the linearization semantics for LCFRS.

Definition 6.2.6. Let G be an LCFRS. The *linearization algebra* for G is the $\Sigma(G)$-algebra $\mathfrak{A}$ in which $\mathrm{dom}(\mathfrak{A})_A = (V^+)^k$, for all $A \in N_G$ and $k = \varphi(A)$, and

$$f_p(\vec{\alpha}_1, \dots, \vec{\alpha}_m) \quad = \quad \vec{\gamma}[a \leftarrow \varepsilon][\, x_{i,j} \leftarrow \mathrm{pfx}_i(\alpha_{i,j}) \mid i \in [m] \land j \in [k_i]\,]\,,$$

for each production $p = A \to f(A_1, \dots, A_m)$ with $f\colon k_1 \times \cdots \times k_m \to k$, anchor $a \in T$ and body $\vec{\gamma}$. The *linearization language* generated by G is the set

$$\Lambda(G) \quad := \quad \{\, \vec{u} \mid t \in T_{\Sigma(G),S_G} \land \langle \vec{u} \rangle \in [\![t]\!]_{\mathfrak{A}}\,\}\,. \qquad \square$$

Each composition operation f_p of a linearization algebra uses the body of the defining equation of p to concatenate a root node (representing the anchor of p) and the appropriately prefixed linearizations for the subderivations in the same order as they would be concatenated in the string algebra.

Example 6.2.3 (continued). For the derivation tree t shown in Figure 6.3a, we get the linearization $[\![t]\!]_{\mathbf{L}} = 1 \cdot 21 \cdot 221 \cdot \varepsilon \cdot 2 \cdot 22$. □

6.2.5 Dependency Semantics

We now define a dependency semantics for LCFRS.

Definition 6.2.7. Let G be an LCFRS, and let $t \in T_{\Sigma(G)}$ be a derivation tree. The dependency structure *induced* by t is the segmented structure $D := (\mathrm{nod}(t)\,; \trianglelefteq, \preceq, \equiv)$ where

$u \trianglelefteq v$ if and only if u dominates v in t,

$u \preceq v$ if and only if u precedes v in $[\![t]\!]_{\mathbf{L}}$, and

$u \equiv v$ if and only if u and v appear in the same component of $[\![t]\!]_{\mathbf{L}}$.□

Example 6.2.4 (continued). The derivation tree t shown in Figure 6.3a induces the dependency structure shown in Figure 6.3b. Note that, while the string language generated by the LCFRS is the same as the string language generated by the CFG in the previous section, the dependency structure is fundamentally different. Just as in the case of context-free grammar, however, the generated string $[\![t]\!]_{\mathbf{S}}$ can be recovered from the linearization by reading the anchors of t in the order specified by $[\![t]\!]_{\mathbf{L}}$. □

Inspecting the linearization semantics, we see that there is a obvious similarity between the bodies of the generalized concatenation functions used in an LCFRS and the order annotations that we used for block-ordered trees: the jth occurrence of the symbol i in our order annotations has the same semantics as the variable $x_{i,j}$ in the body of a generalized concatenation function. Under this view, the linearization semantics of LCFRS mirrors the behaviour of the procedure Block-Order-Collect that we gave in Section 4.2.1. Just as in the case of projective dependency structures, the following theorem can be shown by replicating the proofs of Lemma 4.2.1. In the following, we write LCFRS(k) for the class of all LCFRS in which the fan-out of the non-terminal symbols is bounded by k.

Theorem 6.2.1. $\forall k \in \mathbb{N}.\ \mathcal{D}(\mathit{LCFRS}(k)) = \mathcal{D}_k$ □

We can also define a relabelling function that translates between productions of an LCFRS and order annotations:

$$\mathrm{relab}(p) \ := \ \vec{\alpha}[a \leftarrow 0][\, x_{i,j} \leftarrow i \mid i \in [m] \wedge j \in [\varphi(A_i)]\,]\,,$$

for every production $p = A \to f(A_1, \dots, A_m)$ with anchor a and body $\vec{\alpha}$. Based on this function, the concept of a dependency algebra for an LCFRS can be defined analogously to the corresponding definition for context-free grammars (Definition 6.1.6).

6.2.6 Related Work

The string languages generated by LCFRS have many characterizations; among other things, they are generated by Multiple Context-Free Grammars [105], and they are the images of regular tree languages under deterministic tree-walking transducers [120] and under finite-copying top-down tree transducers [116]. They can also be characterized as the yield languages of rational tree relations [99].

6.3 Coupled Context-Free Grammars

In this section, we look at the dependency structures that are induced by derivations in Coupled Context-Free Grammars (CCFGs) [38, 49]. This formalism can be understood as a variant of LCFRS where rewriting rules are restricted to words over a Dyck language, that is, a language that consists of balanced strings of parentheses.[3] We show that this syntactic restriction enforces the dependency structures induced by CCFG derivations to be well-nested.

6.3.1 Definition

We start with a formal definition of CCFGs. Our definition deviates from the one given by Hotz and Pitsch [49] in that we treat ‘parentheses’ as symbols from a ranked set.

Definition 6.3.1. Let $\mathit{\Pi}$ be a ranked alphabet. The alphabet of *components* of $\mathit{\Pi}$ is the set $\mathrm{comp}(\mathit{\Pi}) := \{\, \langle \pi, i \rangle \in \mathit{\Pi} \times \mathbb{N} \mid 1 \leq i \leq \mathrm{rank}_{\mathit{\Pi}}(\pi) \,\}$. □

[3] Dyck languages are named after the German mathematician Walther von Dyck (1856–1934), whose surname rhymes with ‘week’ rather than ‘dike’.

Table 6.1. Inference rules for the extended semi-Dyck set

$$\frac{}{\varepsilon \in \mathrm{ED}(\mathit{\Pi}, A)} \qquad \frac{a \in A}{a \in \mathrm{ED}(\mathit{\Pi}, A)} \qquad \frac{\vec{x} \in \mathrm{ED}(\mathit{\Pi}, A) \qquad \vec{y} \in \mathrm{ED}(\mathit{\Pi}, A)}{\vec{x} \cdot \vec{y} \in \mathrm{ED}(\mathit{\Pi}, A)}$$

$$\frac{\pi \in \mathit{\Pi}_1}{\langle \pi, 1\rangle \in \mathrm{ED}(\mathit{\Pi}, A)} \qquad \frac{\vec{x}_1 \in \mathrm{ED}(\mathit{\Pi}, A) \quad \cdots \quad \vec{x}_k \in \mathrm{ED}(\mathit{\Pi}, A) \quad \pi \in \mathit{\Pi}_{k+1}}{\langle \pi, 1\rangle \cdot \vec{x}_1 \cdot \langle \pi, 2\rangle \cdots \langle \pi, k\rangle \cdot \vec{x}_k \cdot \langle \pi, k+1\rangle \in \mathrm{ED}(\mathit{\Pi}, A)}$$

In the following, if no confusion can arise, we write π_i instead of $\langle \pi, i\rangle$.

Definition 6.3.2. Let $\mathit{\Pi}$ be a ranked alphabet, and let A be an alphabet. The *extended semi-Dyck set* over $\mathit{\Pi}$ and A is the smallest set $\mathrm{ED}(\mathit{\Pi}, A) \subseteq (\mathrm{comp}(\mathit{\Pi}) \cup A)^*$ that is closed under the inference rules given in Table 6.1. □

Example 6.3.1. Consider the ranked alphabet $\mathit{\Pi} := \{\circ\}$ where $\mathrm{rank}_{\mathit{\Pi}}(\circ) = 2$. The components of $\mathit{\Pi}$ can be understood as opening and closing brackets, respectively. Specifically, let us write [instead of $\langle \circ, 1\rangle$, and] instead of $\langle \circ, 2\rangle$. Then the set $\mathrm{ED}(\mathit{\Pi}, \{a, b\})$ consists of the set of all well-bracketed words over $\{a, b\}^*$. For example, the strings $[a][b]$ and $a[a[ba]][a]$ belong to this set, while the strings $[[a]$ and $[b]a]$ do not. □

One important property of an extended semi-Dyck set $\mathrm{ED}(\mathit{\Pi}, A)$ that we will make use of is that, modulo the associativity of the concatenation rule and the introduction of superfluous empty strings, every string $\vec{x} \in \mathrm{ED}(\mathit{\Pi}, A)$ has a unique decomposition in terms of the rules given in Table 6.1.

Definition 6.3.3. A *coupled context-free grammar* is a construct

$$G = (\mathit{\Pi}, T, S, P)\,,$$

where $\mathit{\Pi}$ is a ranked alphabet of *non-terminal symbols*, T is an alphabet of *terminal symbols*, $S \in \mathit{\Pi}_1$ is a distinguished *start symbol*, and P is a finite, non-empty set of productions of the form $A \to \langle \vec{x}_1, \ldots, \vec{x}_k\rangle$, where $k \in \mathbb{N}$, $A \in \mathit{\Pi}_k$, and $\vec{x}_1 \cdots \vec{x}_k \in \mathrm{ED}(\mathit{\Pi}, T)$. □

Example 6.3.2. We use the following CCFG G as our running example. The alphabet of non-terminal symbols is $\mathit{\Pi}_G := \{S/1, R/2, B/1, C/1, D/1\}$; the alphabet of terminal symbols is $T_G := \{a, b, c, d\}$. The start symbol of G is $S_G := S$. Finally, the set of productions is defined as follows. (We omit subscripts for non-terminals with rank 1.)

$$\begin{aligned} S &\to \langle aR_1BCR_2D\rangle \mid \langle aBCD\rangle \\ R &\to \langle aR_1B, CR_2D\rangle \mid \langle aB, CD\rangle \\ B &\to \langle b\rangle\,, \quad \text{and similarly for } C \text{ and } D. \end{aligned}$$

□

We now show how to construct the derivation algebra for a coupled context-free grammar $G = (\mathit{\Pi}, T, S, P)$. For every string $\vec{x} \in (\mathrm{comp}(\mathit{\Pi}) \cup T)^*$, we define $\mathrm{res}_{\mathit{\Pi}}(\vec{x})$ to be the restriction of $\vec{x}$ to the first components of $\mathit{\Pi}$. More

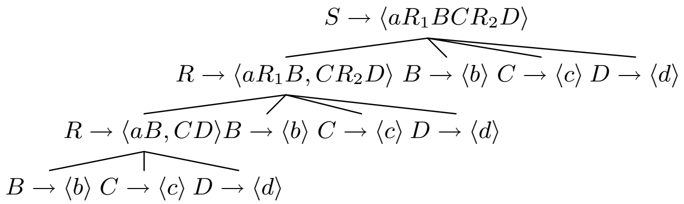


Fig. 6.4. A derivation tree for a CCFG

specifically, res_Π is the homomorphism from $(\mathrm{comp}(\Pi) \cup T)^*$ to Π^* that is defined by $\mathrm{res}_\Pi(\langle \pi, 1\rangle) = \pi$ for $\pi \in \Pi$, and $\mathrm{res}_\Pi(x) = \varepsilon$ for all other symbols. We turn the set P of productions of G into a Π-sorted set $\Sigma(G)$ by defining

$$\mathrm{type}_{\Sigma(G)}(A \to \langle \vec{x}_1, \dots, \vec{x}_k\rangle) \quad := \quad \mathrm{res}_\Pi(\vec{x}_1 \cdots \vec{x}_k) \cdot A\,,$$

for every production $A \to \langle \vec{x}_1, \dots, \vec{x}_k\rangle$, where $A \in \Pi_k$. The set of *derivation trees* of G is defined as for CFG and LCFRS.

Example 6.3.3. Figure 6.4 shows a derivation tree for our example grammar.□

The notion of lexicalization is defined as usual:

Definition 6.3.4. A CCFG is called *lexicalized*, if each of its productions contains exactly one terminal symbol. □

Notice that the example grammar is lexicalized.

6.3.2 String Semantics

The basic reading of a production p in a CCFG is, that like a production in an LCFRS it describes a generalized concatenation function that arranges material derived in subderivations into a new tuple. However, in order to define the semantics of CCFGs precisely, we need to make explicit which components on the right-hand side $\vec{\alpha}$ of p 'belong together' in the sense that they should be replaced with material derived in the same subderivation. In LCFRS, this correspondence is encoded by means of appropriately named variables: the variable $x_{i,j}$ in $\vec{\alpha}$ is a placeholder for the jth component of the ith argument of the generalized concatenation function used in p (see Definition 6.2.4). In CCFG, due to the Dyck-ness restriction on the productions, the correspondence can be left more implicit.

Let us say that two occurrences of components in the right-hand side $\vec{\alpha}$ of a production p in a CCFG are *synchronized*, if they were introduced in the same inference step in the derivation of $\vec{\alpha}$ according to the rules given in Table 6.1. Given that every right-hand side $\vec{\alpha}$ has a unique such derivation (modulo inessential ambiguities), the synchronization relation is well-defined; it defines an equivalence relation on the set of all occurrences of components in $\vec{\alpha}$. We can then index the synchronized sets of components according to the position of their leftmost member (a component of the form $\langle \pi, 1\rangle$), and use the components from the ith group in this sequence as placeholders for the material from the ith subderivation. More formally, we can rewrite $\vec{\alpha}$ into an explicit version $\exp(\vec{\alpha})$ by replacing the jth element of the ith synchronized group of component occurrences by the variable symbol $x_{i,j}$. Based on this explicit version, we can define the string semantics of CCFGs as in the case of LCFRS:

Definition 6.3.5. Let $G = (\Pi, T, S, P)$ be a CCFG. The *string algebra* for G is the $\Sigma(G)$-algebra $\mathfrak{A}$ in which $\operatorname{dom}(\mathfrak{A})_A = (T^*)^k$, for all $A \in \Pi_k$, and

$$f_p(\vec{\alpha}_1, \dots, \vec{\alpha}_m) \quad = \quad \exp(\vec{\gamma})[\, x_{i,j} \leftarrow \alpha_{i,j} \mid i \in [m] \wedge j \in [k_i]\,]\,,$$

for each production $p = A \to \vec{\gamma}$. The *string language* generated by G is the set

$$L(G) \quad := \quad \{\, \vec{a} \mid \exists t \in T_{\Sigma(G),S}.\ \langle \vec{a}\rangle \in [\![t]\!]_{\mathfrak{A}} \,\}\,. \qquad \square$$

Example 6.3.4 (continued). Our example grammar generates the context-sensitive string language $\{\, a^n b^n c^n d^n \mid n \in \mathbb{N} \,\}$. □

It is straightforward that every CCFG can be translated into an LCFRS that generates the same string language. In this translation, both the alphabet of non-terminal symbols, the alphabet of terminal symbols, and the start symbol remain unchanged; the only thing that we need to adapt is the form of the production rules, which can be done in the way that we just explained. In this sense, we can view CCFGs as a syntactically restricted form of LCFRS. We can then define the linearization semantics of CCFGs as for LCFRS.

6.3.3 Dependency Semantics

Given that every CCFG can be seen as a special LCFRS, it is clear that CCFGs cannot induce more dependency structures than LCFRS. In particular, we have the following lemma, which relates the block-degree of the induced dependency structures to the maximal rank of the inducing CCFG. Let us write CCFG(k) for the class of all CCFGs in which the maximal rank of a non-terminal is k.

Lemma 6.3.1. $\forall k \in \mathbb{N}.\ \mathcal{D}(\mathit{CCFG}(k)) \subseteq \mathcal{D}_k$ □

We now show that CCFGs in fact induce a proper subclass of the dependency structures inducible by LCFRS: every dependency structure induced by a CCFG is well-nested.

Lemma 6.3.2. $\mathcal{D}(\mathit{CCFG}) \subseteq \mathcal{D}_{wn}$ □

Proof. Let $G = (\Pi, T, S, P)$ be a CCFG, and let $t = p(t_1, \dots, t_m)$ be a derivation tree of G, where p: $k_1 \times \cdots \times k_m \to k$ and $t_i \in T_{\Sigma(G),k_i}$, for all $i \in [m]$. The proof proceeds by induction on the depth of t. Consider the linearization

$[\![t]\!]_{\mathbf{L}}$ of t; this linearization has the form $[\![t]\!]_{\mathbf{L}} = \langle \vec{u}_1, \dots, \vec{u}_k \rangle$. Now assume that there exist four nodes $v_1 = ui\vec{x}_1$, $v_2 = ui\vec{x}_2$ and $w_1 = uj\vec{y}_1$, $w_2 = uj\vec{y}_2$ in t such that $v_1 \to v_2$, $w_1 \to w_2$, and $[v_1, v_2] \between [w_1, w_2]$. Furthermore, and without loss of generality, assume that $\min(v_1, v_2) \prec \min(w_1, w_2)$. Then the string $\vec{u}_1 \cdots \vec{u}_k$ contains a substring of the form

$$ui\vec{x}_1 \cdots uj\vec{y}_1 \cdots ui\vec{x}_2 \cdots uj\vec{y}_2\,,$$

for some $u \in \mathrm{nod}(t)$, $i, j \in \mathbb{N}$, and $\vec{x}_1, \vec{x}_2, \vec{y}_1, \vec{y}_2 \in V^*$. Distinguish two cases: If $u \neq \varepsilon$, then both v_1, v_2 and w_1, w_2 belong to a proper subterm of t, and by the induction hypothesis, we may assume that $v_1 \trianglelefteq w_1$ or $w_1 \trianglelefteq v_1$. So assume that $u = \varepsilon$. The right-hand side of the production p has the form $\langle \gamma_1, \dots, \gamma_k \rangle$, and the string $\gamma := \gamma_1 \cdots \gamma_k$ is formed according to the inference rules given in Table 6.1. Given the specific form of the string $\vec{u}_1 \cdots \vec{u}_k$, the string γ contains a substring of the form

$$\langle \pi_i, i_1 \rangle \cdots \langle \pi_j, j_1 \rangle \cdots \langle \pi_i, i_2 \rangle \cdots \langle \pi_j, j_2 \rangle\,,$$

for some $\pi_i, \pi_j \in \Pi$ and $1 \leq i_1 < i_2 \leq \mathrm{rank}_\Pi(\pi_i)$, $1 \leq j_1 < j_2 \leq \mathrm{rank}_\Pi(\pi_j)$, where i is the left-to-right index of the synchronized group of (occurrences of) components to which $\langle \pi_i, i_1 \rangle$ and $\langle \pi_i, i_2 \rangle$ belong, and j is the corresponding index for $\langle \pi_j, j_1 \rangle$ and $\langle \pi_j, j_2 \rangle$. By the inferences rules in Table 6.1, it is then clear that all four occurrences are synchronized; consequently, $i = j$, and either $v_1 \trianglelefteq w_1$ or $w_1 \trianglelefteq v_1$ holds. This shows that the dependency structure induced by t is well-nested. ■

As in the case of CFGs and LCFRS, the converses of the preceding lemmata are easy to show. We thus obtain a characterization of CCFG in terms of the dependency structures that it can induce as follows:

Theorem 6.3.1. $\forall k \in \mathbb{N}.\ \mathcal{D}(\mathit{CCFG}(k)) = \mathcal{D}_k \cap \mathcal{D}_{wn}$ □

6.3.4 Related Work

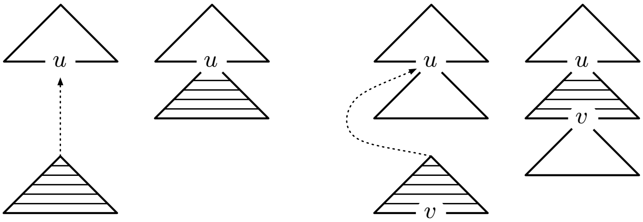
Coupled context-free grammars are closely related to several other formalisms considered in the literature, such as macro grammars [25] and the yield languages of linear, non-deleting context-free tree grammars [59]. The class of languages generated by these formalisms enjoys several properties that makes it more attractive than LCFRS from a formal point of view, such as a universal pumping lemma [58] and the existence of binary normal forms and efficient parsing algorithms [34].

6.4 Tree Adjoining Grammar

To conclude this chapter, we now look at the dependency structures that are induced by derivations in Tree Adjoining Grammars (TAG) [54, 55]. In contrast to LCFRS and CCFGs, TAGs manipulate trees rather than strings.

6.4.1 Definition

The building blocks of a TAG are called *elementary trees*. These are children-ordered trees in which each node has one of three types: it can be an *anchor* (or *terminal node*), a *non-terminal node*, or a *foot node*. Anchors and foot nodes are required to be leaves; non-terminal nodes may be either leaves or inner nodes. Each elementary tree can have at most one foot node. Elementary trees without a foot node are called *initial trees*; the remaining trees are called *auxiliary trees*. A TAG grammar is *lexicalized*, if each of its elementary trees contains exactly one anchor [104]. Trees in TAG can be combined using two operations (see Figure 6.5): *substitution* combines a tree τ with an initial tree τ' by identifying a non-terminal leaf node u of τ with the root node of τ' (Figure 6.5a); *adjunction* identifies an inner node u of a tree τ with the root node of an auxiliary tree τ'; the subtree of τ that is rooted at u is excised from τ and inserted below the foot node v of τ' (Figure 6.5b). Combination operations are disallowed at root and foot nodes.



(a) substitution (b) adjunction

Fig. 6.5. Combination operations in TAG

Example 6.4.1. Figure 6.6 shows an example for how TAGs are specified. The grammar contains 5 elementary trees, named τ_1 to τ_5. The elementary trees τ_1–τ_4 are initial trees. The tree τ_5 is an auxiliary tree; the foot node of this tree is marked with a star. Note that this grammar is lexicalized. By adjoining the tree τ_5 into the tree τ_1, and then repeatedly into the tree resulting from this first adjunction, we can produce the string language $\{ a^n b^n c^n d^n \mid n \in \mathbb{N} \}$. This language is beyond the string-generative capacity of context-free grammars.□

Just as in the case of the other grammar formalisms that we have looked at in this section, TAG *derivation trees* record information about how elementary structures are combined. Formally, derivation trees can be seen as terms over the signature of elementary trees; this set is finite for any given TAG. The root of each derivation tree is an initial tree. By repeated applications of the substitution and adjunction operations, larger and larger trees are built from this tree. TAG *derived trees* represent the results of complete derivations: they are standard children-ordered trees made up from the accumulated material of the elementary trees participating in the derivation. Just as in the case of

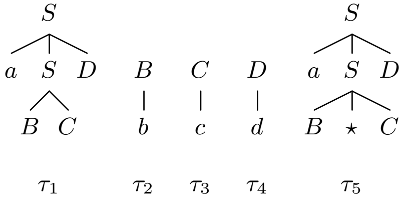
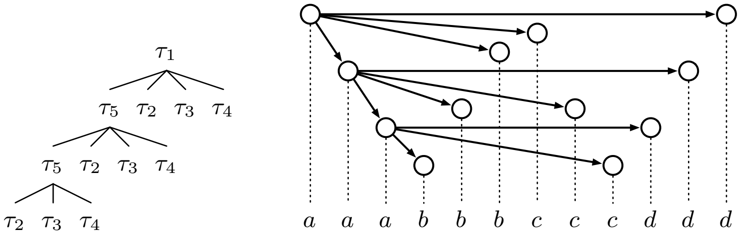


Fig. 6.6. A TAG grammar

lexicalized CFG there was a one-to-one correspondence between the nodes of the derivation tree and the positions of the derived string, in lexicalized TAGs there is a one-to-one correspondence between the nodes of the derivation tree and the leaves of the derived tree. Thus, in just the same way as derivations in CFG and LCFRS, derivations in TAG induce dependency structures. The major question that we have to answer in the context of TAGs is how to define the linearization semantics of the derivation trees. Essentially the same question needs to be addressed when trying to relate TAG to other mildly context-sensitive grammar formalisms. (See 5 for a formal version of the construction that we discuss here.)

Example 6.4.2 (continued). Figure 6.7 shows a derivation tree for our example grammar and the dependency structure that is induced by this derivation. □


τ_1
τ_5 τ_2 τ_3 τ_4
τ_5 τ_2 τ_3 τ_4
τ_2 τ_3 τ_4
a a a b b b c c c d d d

(a) derivation tree

(b) dependency structure

Fig. 6.7. A TAG derivation, and the induced dependency structure

6.4.2 Linearization Semantics

To understand the linearization semantics of a TAG elementary tree τ, we must read it with the derived tree in mind that would result from a derivation starting at τ. Let us do so for the elementary tree τ_1 that is shown in Figure 6.6. Since combination operations are disallowed at the root nodes of elementary trees, we see that the leftmost leaf in the derived tree that we produce as the result of the derivation is the anchor of τ_1, which is labelled with the symbol a. Now assume that an auxiliary tree adjoins at the central node of τ_1, the node that is labelled with S. Then in the derived tree, all the material in the left half of the adjoined tree precedes the material that is dominated by the adjunction site in τ_1, while all the material in the right half succeeds it (see again Figure 6.5b). Specifically, let us write S_1 for the material derived from the left half of the auxiliary tree, and S_2 for the material in the right half. Then S_1 precedes the material that gets substituted into the nodes labelled with B and C in τ_1, and this material in turn precedes S_2. Finally, at the right edge of the derived tree that we build from τ_1, we find the material substituted into the node labelled with D. To summarize, we have the following linear sequence of tree material:

$$a\ S_1\ B\ C\ S_2\ D\,.$$

Using reasoning similar to this, we see that every elementary tree specifies a generalized concatenation function over tuples of arity at most 2: for subderivations that correspond to adjunctions, there is not one single slot per subderivation, but two; the first of these slots is filled by the material in the left half of the adjoined tree, the second slot by the right half of the corresponding tree. A crucial observation now is that the linearization of the elementary trees always produces strings from an extended semi-Dyck set. More specifically, there can never be two distinct adjunction sites A and B such that the linearization of the corresponding elementary tree yields the sequence $A_1 \cdots B_1 \cdots A_2 \cdots B_2$. This is so because all material that is situated between the two slots of a given adjunction corresponds to material that in the elementary tree is dominated by that adjunction site. The forbidden sequence then would mean that both A dominates B, and B dominates A, which is only possible if $A = B$. Hence, from the linearization point of view, TAG corresponds to the class CCFG(2) of Coupled Context-Free Grammars with rank at most 2.

Theorem 6.4.1. $\mathcal{D}(\mathit{TAG}) = \mathcal{D}_2 \cap \mathcal{D}_{wn}$ □

6.4.3 Related Work

The result that the dependency structures induced by derivations of Tree Adjoining Grammar are exactly the well-nested dependency trees with block-degree at most 2 was first shown by Bodirsky, Kuhlmann, and Möhl [3]. Kuhlmann and Möhl [65] present an extension of this result to *multi-component* TAG. The ramifications of this extension were later discussed by Chen-Main and Joshi [9, 10].

The same reasoning that we have used for the linearization semantics is needed when designing left-to-right parsing algorithms for TAGs [5, 54]. It was also used by Guan [38] to link TAGs to CCFGs of rank 2.

Tree Adjoining Grammars are special forms of Ranked Node Rewriting Grammars [1] and context-free tree grammars [26, 59, 81]. Our results carry over to these extended formalisms.

Summary

In this chapter, we have presented a classification of lexicalized grammar formalisms in terms of the dependency structures that these formalisms can induce. Our classification provides a new measure for the generative capacity of a grammar formalism that is attractive as an alternative to both string-generative capacity and tree-generative capacity: dependency structures are more informative than strings, but less formalism-specific than parse trees.[4]

Together with the treebank studies that we presented in the previous three chapters, our classification also provides new insights into the practical relevance of grammar formalisms: If we accept our conclusion that the class of projective dependency structures is insufficient to cover all the data in the three treebanks that we looked at, then by Theorem 6.1.1, the same holds for lexicalized context-free grammars. At the same time, our treebank studies revealed that only a small step beyond projectivity is necessary to cover virtually all of the practically relevant data. Together with Theorem 6.2.1, we can interpret this result as saying that we only need LCFRS with a very small fan-out, say the class LCFRS(2). Perhaps most interestingly, we find that close to 99.5% of all the dependency analyses in the treebanks are well-nested and have a block-degree of at most 2. Given Theorem 6.4.1, this means that it is possible, at least in theory, to write a TAG that induces (almost) all the structures in the three treebanks.

[4] Kallmeyer [57] makes a similar argument.

7

Regular Dependency Languages

In the first part of this book, we have looked at formal properties of individual dependency structures. In this chapter, we turn our attention to sets of such structures, or *dependency languages*. Specifically, we investigate the languages that arise when we equip dependency structures with a 'regular' means of syntactic composition.

We start by defining regular dependency languages as the recognizable subsets in dependency algebras and provide natural notions of automata and grammars for this class of languages (Section 7.1). We then develop a powerful pumping lemma for regular dependency languages (Section 7.2) and apply it to show that the languages in this class are of constant growth, a property characteristic for mildly context-sensitive languages (Section 7.3).

7.1 Regular Sets of Dependency Structures

The primary goal of this book is to illuminate the connections between language-theoretic properties such as generative capacity and parsing complexity on the one hand, and graph-theoretic properties such as block-degree and well-nestedness on the other. Specifically, we started with the question,

> Which grammars induce which sets of dependency structures?

At this point, we have already come quite close to an answer to this question. Consider the class of lexicalized context-free grammars (LCFG) for example. In the previous chapter, we have seen that LCFG is linked to projectivity in the sense that every LCFG can induce only projective structures, and every such structure can be induced by some LCFG. However, this result does not yet provide a full answer to our question, which asks about classes of *languages*, not classes of *structures*. The step from structures to languages is non-trivial: it is not true that every *set* of projective dependency structures

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 85–102, 2010.

can be induced by an LCFG; in particular, the set of LCFGs is denumerable, the set of all subsets of $\mathcal{D}_1$ is not. In this chapter, we identify a class of dependency languages that *can* be induced by the grammar formalisms that we have discussed. We call this class the *regular* dependency languages. As we will see, the condition of regularity provides the missing link between grammar formalisms and dependency structures.

7.1.1 Algebraic Recognizability

We define regularity by referring to the concept of *algebraic recognizability.* This notion was introduced by Mezei and Wright [78], following previous work by Richard Büchi, John Doner, Calvin Elgot, and James Thatcher. It generalizes the definitions of regular string languages to arbitrary abstract algebras and provides a canonical way to characterize regular sets of objects. Recognizability was originally defined for single-sorted algebras. Here we use a many-sorted version, due to Courcelle [12].

Definition 7.1.1. Let $\mathfrak{A}$ be a Σ-algebra, and let $s \in \mathcal{S}_\Sigma$ be a sort. A set $L \subseteq \mathrm{dom}(\mathfrak{A})_s$ is called *recognizable*, if there exists a finite Σ-algebra $\mathfrak{B}$, a homomorphism h: $\mathfrak{A} \to \mathfrak{B}$, and a set $F \subseteq \mathrm{dom}(\mathfrak{B})_s$ such that $L = h^{-1}(F)$. □

We want to call a set of dependency structures 'regular', if it is recognizable in some dependency algebra $\mathfrak{D}$. In this case, since $\mathfrak{D}$ is initial, we do not have a choice about the homomorphism h in the above definition: it is the uniquely determined evaluation homomorphism $[\![\cdot]\!]_\mathfrak{B}$. This leads to the following definition of regularity:

Definition 7.1.2. Let $\mathfrak{D}$ be a dependency algebra with signature Σ, and let $i \in \mathcal{S}_\Sigma$ be a sort. A set $L \subseteq \mathrm{dom}(\mathfrak{D})_i$ is called *regular*, if there exists a finite Σ-algebra $\mathfrak{B}$ and a set $F \subseteq \mathrm{dom}(\mathfrak{B})_i$ such that $L = [\![F]\!]_\mathfrak{B}^{-1}$. □

The pair $M = (\mathfrak{B}, F)$ is called a (deterministic, complete) *automaton* for L. We can understand the signature of $\mathfrak{B}$ as an *input alphabet*, the domains $\mathrm{dom}(\mathfrak{B})$ as sets of *states*, the (finitely many) possible combinations of input and output values for the composition operations of $\mathfrak{B}$ as a *transition function*, and the set F as a set of *final states* of M. The behaviour of M can be described as follows: A *run* of M is a bottom-up traversal of a dependency structure D during which each node u gets labelled with a state $q \in \mathrm{dom}(\mathfrak{B})$. The label for u is chosen conditional on both the local order at u (represented by an order annotation $\sigma \in \Sigma$), and the state labels at the children of u. More specifically, when M visits a node u that is annotated with a symbol σ, and the children of the node u have previously been labelled with states $q_1, \dots, q_m$, then the automaton labels u with the state $f_\sigma(q_1, \dots, q_m)$. The automaton M *recognizes* D if, at the end of the run, the root node of D is labelled with a state $q \in F$.

Example 7.1.1. To illustrate the definition of regularity, we show that, for every dependency algebra $\mathfrak{D}$ with signature Σ, every $k \in \mathcal{S}_\Sigma$, and every set $G \subseteq \{\, f_\sigma \mid \sigma \in \Sigma \,\}$, the set L of those structures of sort k that are composed using only operations from the set G is regular. We do so by constructing an automaton $M = (\mathfrak{B}, F)$ for L as follows. For each sort $i \in \mathcal{S}_\Sigma$, the state set $\mathrm{dom}(\mathfrak{B})_i$ is the set $\{1, 0\}$. For each order annotation σ: $s_1 \times \cdots \times s_m \to s$ in Σ and each tuple $\langle b_1, \dots, b_m \rangle$ of values, we put $f_\sigma(b_1, \dots, b_m) := (\bigwedge_{i=1}^{m} b_i) \wedge b$, where $b = 1$ if and only if $f_\sigma \in G$. As the set of final states F, we choose $\{1\} \subseteq \mathrm{dom}(\mathfrak{B})_k$. For each dependency structure $D \in \mathrm{dom}(\mathfrak{D})$, the evaluation of D in $\mathfrak{B}$ returns 1 if and only if it was composed using only composition operations from the set G. Thus, the set $[\![F]\!]_{\mathfrak{D}}^{-1}$ contains exactly those dependency structures with this property that have sort k. □

We write REGD for the class of all regular dependency languages. For a given class $\mathcal{D}$ of dependency structures, we write REGD($\mathcal{D}$) for the class of all regular dependency languages that are subsets of $\mathcal{D}$.

7.1.2 Elementary Properties

We now review some of the elementary formal properties of regular dependency languages. All of these properties are immediate consequences of our definitions and general results about recognizable subsets.

Lemma 7.1.1. *The empty set is a regular dependency language. Furthermore, REGD is closed under union, intersection, difference, and inverse homomorphisms.* □

Proof. See e.g. Courcelle [12], Proposition 4.6. ■

Lemma 7.1.2. *The following relations hold for all $k \in \mathbb{N}$:*

- $\mathit{REGD}(\mathcal{D}_k) \subsetneq \mathit{REGD}(\mathcal{D}_{k+1})$
- $\mathit{REGD}(\mathcal{D}_k \cap \mathcal{D}_{wn}) \subsetneq \mathit{REGD}(\mathcal{D}_{k+1} \cap \mathcal{D}_{wn})$
- $\mathit{REGD}(\mathcal{D}_1) = \mathit{REGD}(\mathcal{D}_1 \cap \mathcal{D}_{wn})$
- $k \neq 1 \implies \mathit{REGD}(\mathcal{D}_k \cap \mathcal{D}_{wn}) \subsetneq \mathit{REGD}(\mathcal{D}_k)$ □

Proof. Each of the classes of dependency structures mentioned in this lemma coincides with a specific sub-signature of the set Ω of all order annotations. All relations therefore can be reduced to the corresponding relations on the signatures. ■

This lemma is visualized in Figure 7.1. It shows that the structural restrictions imposed by the block-degree measure and the well-nestedness condition induce two infinite, related hierarchies of ever more expressive regular dependency languages. The only point where these two hierarchies coincide is the case $k = 1$.

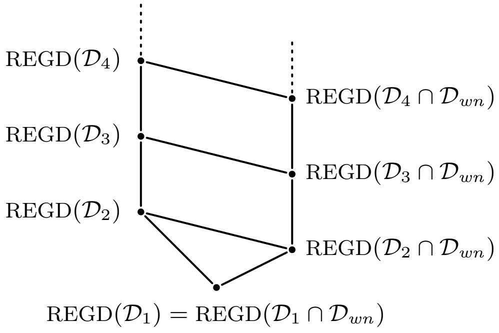

REGD($\mathcal{D}_4$)
REGD($\mathcal{D}_4 \cap \mathcal{D}_{wn}$)
REGD($\mathcal{D}_3$)
REGD($\mathcal{D}_3 \cap \mathcal{D}_{wn}$)
REGD($\mathcal{D}_2$)
REGD($\mathcal{D}_2 \cap \mathcal{D}_{wn}$)
REGD($\mathcal{D}_1$) = REGD($\mathcal{D}_1 \cap \mathcal{D}_{wn}$)

Fig. 7.1. The hierarchy of regular dependency languages

Finite Degree

Since each regular dependency language is built using a finite set of composition operations (a finite signature), and since there is a direct correspondence between the type of a composition operation and the measures of out-degree and block-degree, no regular dependency language can be unbounded in either measure.

Definition 7.1.3. Let $L \subseteq \mathcal{D}$ be a dependency language. We say that L is of *finite degree*, if there are a numbers $m \in \mathbb{N}$, $k \in \mathbb{N}$ such that no structure in L has an out-degree of more than m or a block-degree of more than k. □

Lemma 7.1.3. *Every regular dependency language is of finite degree.* □

The property that regular dependency languages are of finite out-degree separates them from dependency frameworks that allow an arbitrary number of children per node (see e.g. [19]). The restriction to finite block-degree formalizes the rather informal notion of 'limited cross-serial dependencies' that is characteristic for mildly context-sensitive language [53]. At the same time, this restriction implies that regular dependency languages are not able to account for linguistic phenomena that require arbitrary degrees of non-projectivity, such as the phenomenon of scrambling in German subordinate clauses [2].

Connection with Regular Term Languages

Recall from Theorem 4.2.1 that every dependency algebra is isomorphic to its corresponding term algebra. Therefore the following lemma applies:

Lemma 7.1.4. *Let $\mathfrak{D}$ be a dependency algebra with signature Σ, and let $i \in \mathcal{S}_\Sigma$ be a sort. Then a set $L \subseteq dom(\mathfrak{D})_i$ is regular if and only if $term(L)$ is T_Σ-recognizable.* □

The major benefit of this connection is that it allows us to study dependency languages using the tools and results of the well-developed formal theory of recognizable term languages, which is more widely known as the class of *regular term languages*. This is a very natural and robust class with many different characterizations: it is recognized by finite tree automata, generated by regular term grammars, and definable in monadic second-order logic [28]. For our purposes, the characterization in terms of grammars is the most convenient. It allows us to characterize regular dependency languages as the images of the term languages generated by regular term grammars over the signature Ω of order annotations.

7.1.3 Regular Term Grammars

A *term grammar* specifies a rewriting system for terms over an alphabet of terminal and non-terminal symbols. In each step of the rewriting process, a non-terminal symbol is replaced by a term; this yields a new term. Regular term grammars are the generative correspondents of the algebraic automata from Definition 7.1.2: Essentially, they are obtained by reading the composition functions of an automaton as a directed rewrite system—whenever an automaton defines $f_\sigma(q_1, \dots, q_m) = q$, the corresponding term grammar contains a rule $q \to f_\sigma(q_1, \dots, q_m)$; the states of the automaton become the non-terminal symbols of the grammar. Regular term grammars are distinguished from other term grammars by the restriction that non-terminal symbols may occur only at the leaves of a term, which implies that derivations correspond to sequences of simple substitution operations, just as in context-free grammars.[1] To formalize this restriction, we introduce the following notation:

Definition 7.1.4. Let $\mathcal{S}$ be a set of sorts. Let Σ be an $\mathcal{S}$-sorted set, and let A be an $\mathcal{S}$-indexed family of sets. The set of terms over Σ indexed by A, denoted by $T_\Sigma(A)$, is the $\mathcal{S}$-indexed set of all terms over $\Sigma \cup A$. □

Regular term grammars are usually defined for single-sorted algebras [16, 28]; here we adapt their definition to the many-sorted case. This extension is straightforward: instead of one non-terminal alphabet and set of productions, we need one such alphabet and set of productions per sort. The set of productions is set up such that a given non-terminal symbol can only be rewritten by a term of the same sort.

Definition 7.1.5. Let $\mathcal{S}$ be a finite set of sorts. A *regular term grammar* (over $\mathcal{S}$) is a construct $G = (N, \Sigma, S, P)$, where N is an $\mathcal{S}$-indexed family of *non-terminal alphabets*, Σ is an $\mathcal{S}$-sorted *terminal alphabet*, $S \in N$ is a distinguished *start symbol*, and $P \subseteq N \times T_\Sigma(N)$ is a finite, $\mathcal{S}$-indexed family of sets of *productions*. □

[1] In fact, one can show that a language is context-free if and only if it is the frontier of a set of terms generated by a regular term grammar [28, p. 33].

We use indexed symbols (N_G, Σ_G, S_G, P_G) to refer to the components of specific regular term grammars G. For a production $p = (A, t)$, we call A the *left-hand side* and t the *right-hand side* of p. Just as in conventional string grammars, we usually write $A \to t$ instead of (A, t). The *derivation relation* associated to a regular term grammar $G = (N, \Sigma, S, P)$ is the binary relation $\Rightarrow_G$ on $T_\Sigma(N)$ that is defined by the following inference rule:

$$\frac{t \in T_\Sigma(N) \qquad t/u = A \qquad (A \to t') \in P}{t \Rightarrow_G t[u \leftarrow t']}$$

Using this relation, the definition of the *term language* generated by G is completely analogous to the definition of the language generated by a string grammar—it is the set of all terms without non-terminals that can eventually be derived from the trivial term formed by the start symbol of the grammar: $L(G) = \{\, t \in T_\Sigma \mid S \Rightarrow_G^* t \,\}$. Two grammars are *equivalent*, if they generate the same language. Notice that all terms in the language generated by a regular term grammar are of one and the same sort; this is the sort of the start symbol S.

Definition 7.1.6. A regular term grammar $G = (N, \Sigma, S, P)$ over $\mathcal{S}$ is called *normalized*, if every production has the form $A \to \sigma(A_1, \dots, A_m)$, where $A \in N_s$, $\sigma\colon s_1 \times \cdots \times s_m \to s$, and $A_i \in N_{s_i}$, for every $i \in [m]$ and some sort $s \in \mathcal{S}$. □

Lemma 7.1.5. *For every regular term grammar, there exists an equivalent regular term grammar that is normalized.* □

Proof. A proof of this result can be found in Gécseg and Steinby [28]. The proof is a standard grammar transformation, as is it also known from context-free grammars: we delete rules of the form $A \to B$ and rules where the right-hand side is a term with a depth greater than 1, and replace them by new rules and non-terminal symbols that jointly simulate the old rules. ■

7.1.4 Regular Dependency Grammars

We now define *regular dependency grammars* as regular term grammars that generate term languages over the signature Ω of order annotations. This restriction ensures that terms manipulated by regular dependency grammars can be interpreted as (segmented) dependency structures.

Definition 7.1.7. Let $k \in \mathbb{N}$. A *regular dependency grammar* of degree k is a construct $G = (N, S, P)$, where N is a $[k]$-indexed family of *non-terminal alphabets*, $S \in N$ is a distinguished *start symbol*, and $P \subseteq N \times T_{\Omega(k)}(N)$ is a finite, $[k]$-indexed family of sets of productions. □

For a given regular dependency grammar G, let Σ be the finite subset of order annotations that occurs in the productions of G; then the construct

(N, Σ, S, P) forms a regular term grammar. Based on this observation, we make free use of all terminology for regular term grammars when talking about regular dependency grammars. We will only work with regular dependency grammars in which $S \in N_1$. This restriction ensures that the languages generated by regular dependency grammars can be interpreted as sets of proper dependency structures.

Definition 7.1.8. Let G be a regular dependency grammar. The *dependency language* generated by G is the set $\mathcal{D}(G) := \mathrm{dep}(L(G))$. □

Example 7.1.2. To illustrate the definitions, we give two examples of regular dependency grammars. The dependency languages generated by these grammars mimic the verb-argument relations found in German and Dutch subordinate clauses, respectively [52, 95, 107]: grammar G_1 generates structures with nested dependencies, grammar G_2 generates structures with crossing dependencies.

Grammar $G_1 := (N_1, S, P_1)$ (degree 1)

$$\begin{aligned} N_1 &:= \{1 \mapsto \{S, N, V\}\} \\ P_1 &:= \{S \to \langle 120\rangle(N, V),\ V \to \langle 120\rangle(N, V),\ V \to \langle 10\rangle(N),\ N \to \langle 0\rangle\} \end{aligned}$$

Grammar $G_2 := (N_2, S, P_2)$ (degree 2)

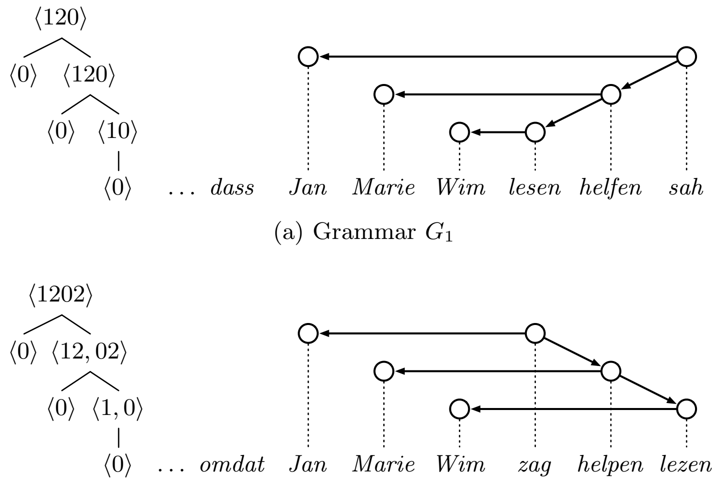
$$\begin{aligned} N_2 &:= \{1 \mapsto \{S, N\}, 2 \mapsto \{V\}\} \\ P_2 &:= \{S \to \langle 1202\rangle(N, V),\ V \to \langle 12, 02\rangle(N, V),\ V \to \langle 1, 0\rangle(N),\ N \to \langle 0\rangle\} \end{aligned}$$

Figure 7.2 shows terms generated by these grammars, and the corresponding structures. □

We now state the main result of this section:

Theorem 7.1.1. *A dependency language is regular if and only if it is generated by a regular dependency grammar.* □

Proof. This is a direct consequence of the isomorphism between regular dependency languages and regular term languages (Lemma 7.1.4) and the standard result that a term language is regular if and only if it is generated by a regular term grammar. A proof of this result can be found in Denecke and Wismath [16]; it proceeds by translating every regular term grammar into an equivalent algebraic automaton and vice versa. The major difference between grammars and automata is that automata are complete and deterministic (exactly one value per function-argument pairing), while grammars may be incomplete and indeterministic (more than one rule per non-terminal). These differences can be removed by grammar normalizations on the one hand, and a standard subset construction for automata on the other. ■


⟨120⟩
⟨0⟩ ⟨120⟩
⟨0⟩ ⟨10⟩
⟨0⟩
... dass Jan Marie Wim lesen helfen sah
⟨1202⟩
⟨0⟩ ⟨12, 02⟩
⟨0⟩ ⟨1, 0⟩
⟨0⟩
... omdat Jan Marie Wim zag helpen lezen

(a) Grammar G_1

(b) Grammar G_2

Fig. 7.2. Terms and structures generated by two regular dependency grammars

7.1.5 Dependency Languages and Lexicalized Grammars

With the concept of regularity at hand, we can now lift our results from the previous chapter to the level of languages. Given a class $\mathcal{G}$ of grammars, let us write $\mathcal{D}(\mathcal{G})$ for the class of all dependency languages induced by grammars in $\mathcal{G}$. The following theorem was first mentioned by Kuhlmann and Möhl [66]:

Theorem 7.1.2. *The following statements hold for all $k \in \mathbb{N}$:*

1. $\mathit{REGD}(\mathcal{D}_k) = \mathcal{D}(\mathit{LCFRS}(k))$
2. $\mathit{REGD}(\mathcal{D}_k \cap \mathcal{D}_{wn}) = \mathcal{D}(\mathit{CCFG}(k))$ □

To put it into words: The dependency languages induced by the class of LCFRS with fan-out at most k are exactly the regular dependency languages over dependency structures with block-degree at most k. Similarly, the dependency languages induced by the class of CCFGs with rank at most k are exactly the regular dependency languages over well-nested dependency structures with block-degree at most k.

Proof. The proof falls into two parts:

$\supseteq$ Let G be a lexicalized LCFRS. The set of derivation trees of G forms a regular term language over some finite signature of concatenation functions. By Lemma 6.2.1, we can transform this language into an equivalent (modulo relabelling) term language L that only uses essential concatenation functions. Crucially, the elimination of non-essential functions uses bottom-up and top-down relabellings, which preserve regularity [22]; therefore, the transformed language L still is a regular term language, say

$L \subseteq T_\Sigma$. We have furthermore seen (on page 75) how to define a bijective function relab: $\Sigma \to \Omega$ from the set of essential concatenation functions to the set of order annotations such that a derivation $t \in T_\Sigma$ induces the dependency structure dep(relab(t)), for all $t \in L$. Since the mapping relab is injective, we can translate L into a regular term language L' over some finite signature $\Delta \subseteq \Omega$, and hence, modulo decoding, into a regular dependency language.

$\subseteq$ Let G be a regular dependency grammar. We can construct an LCFRS G' such that the derivations of G' induce the dependency language generated by G by reversing the above relabelling on a per-rule basis.

In both directions, the relabelling maintains the signature restrictions: essential concatenation functions of fan-out k are translated into order annotations of degree k, and vice versa. The relabelling also maintains the well-nestedness restriction. ■

7.2 Pumping Lemmata

Since even infinite regular term languages (and hence: regular dependency languages) can be represented by finite grammars, these languages, very much like regular or context-free string languages, must have a periodic structure. In this section, we prove a number of novel pumping lemmata for regular term languages that make this observation precise. These lemmata provide the keys to our results about the growth of regular dependency languages (Section 7.3) and their string-generative capacity (Section 8.2).

7.2.1 The Pumping Lemma for Regular Term Languages

Recall the standard pumping lemma for context-free string languages:

> For every context-free language $L \subseteq A^*$, there is a number $p \in \mathbb{N}$ such that any string $z \in L$ of length at least p can be written as $z = uvwxy$ such that $1 \leq |vx|$, $|vwx| \leq p$, and $uv^nwx^ny \in L$, for every $n \in \mathbb{N}$.

This result is usually proved using a combinatorial argument about the derivations of a grammar that generates the language L. Essentially the same argument can be used to show a pumping lemma for regular term grammars (see e.g. Proposition 5.2 in [28]):[2]

Lemma 7.2.1. *For every regular term language $L \subseteq T_\Sigma$, there is a number $p \in \mathbb{N}$ such that any term $t \in L$ of size at least p can be written as $t = c' \cdot c \cdot t'$ such that $1 \leq |c| \leq p$, $|c \cdot t'| \leq p$, and $c' \cdot c^n \cdot t' \in L$, for every $n \in \mathbb{N}$.* □

[2] The lemma given here is in fact slightly stronger than the one given by Gécseg and Steinby [28] (Proposition 5.2), and makes pumpability dependent on the size of a tree, rather than on its height.

The number p in this lemma is called *pumping number*. The phrase 'the term t can be written as $t = c' \cdot c \cdot t'$' is an abbreviation for the formal assertion that 'there exist contexts $c' \in C_\Sigma$ and $c \in C_\Sigma$ and a term $t' \in T_\Sigma$ such that $t = c' \cdot c \cdot t'$'.

Just as the pumping lemma for context-free string languages, Lemma 7.2.1 is most often used in its contrapositive formulation, which specifies a strategy for proofs that a language $L \subseteq T_\Sigma$ is *not* regular: show that, for all $p \geq 1$, there exists a term $t \in L$ of size at least p such that for any decomposition $c' \cdot c \cdot t'$ of t in which $|c| \geq 1$ and $|c \cdot t'| \leq p$, there is a number $n \in \mathbb{N}$ such that $c' \cdot c^n \cdot t' \notin L$. It is helpful to think of a proof according to this strategy as a game against an imagined ADVERSARY, where our objective is to prove that L is non-regular, and ADVERSARY's objective is to foil this proof. The game consists of four alternating turns: In the first turn, ADVERSARY must choose a number $p \geq 1$. In the second turn, we must respond to this choice by providing a term $t \in L$ of size at least p. In the third turn, ADVERSARY must choose a decomposition of t into fragments $c' \cdot c \cdot t'$ such that $|c| \geq 1$ and $|c \cdot t'| \leq p$. In the fourth and final turn, we must provide a number $n \in \mathbb{N}$ such that $c' \cdot c^n \cdot t' \notin L$. If we are able to do so, we win the game; otherwise, ADVERSARY wins. We can prove that L is non-regular, if we have a winning strategy for the game.

Example 7.2.1. Consider the following term language (see Figure 7.3a for a schema):

$$L_1 \quad := \quad \{\, f(g^n \cdot a, g^n \cdot a) \mid n \in \mathbb{N} \,\}$$

We show that L_1 is non-regular by stating a winning strategy for the game associated with L_1. Assume that ADVERSARY has chosen the number $p \in \mathbb{N}$. Then we can always win the game by responding with the term $t = f(g^p \cdot a, g^p \cdot a)$. It is clear that t is a valid term in L_1, and that $|t| \geq p$. In whatever way ADVERSARY decomposes t into segments $c' \cdot c \cdot t'$, the term $s := c' \cdot c^2 \cdot t'$ does not belong to L_1. In particular, if c is rooted at a node that is labelled with g, then the pumped term violates the constraint that the two branches have the same length. Thus we deduce that L_1 is non-regular. □

Unfortunately, Lemma 7.2.1 sometimes is too blunt a tool to show the non-regularity of a term language, as the following example demonstrates.

Example 7.2.2. Consider the following term language (see Figure 7.3b for a schema):

$$L_2 \quad = \quad \{\, f(g^n \cdot h^{m_1} \cdot a, g^n \cdot h^{m_2} \cdot a) \mid n, m_1, m_2 \geq 1 \,\}$$

It is not unreasonable to believe that L_2, like L_1, is non-regular, but it is impossible to prove this using Lemma 7.2.1. To see this, notice that ADVERSARY has a winning strategy for $p \geq 2$: for every term $t \in L_2$ that we can provide in the second turn of the game, ADVERSARY can choose any decomposition $c' \cdot c \cdot t'$ in which $c = h(\circ)$ and $t' = a$. In this case, $|c| \geq 1$, $|c \cdot t'| \leq p$, and

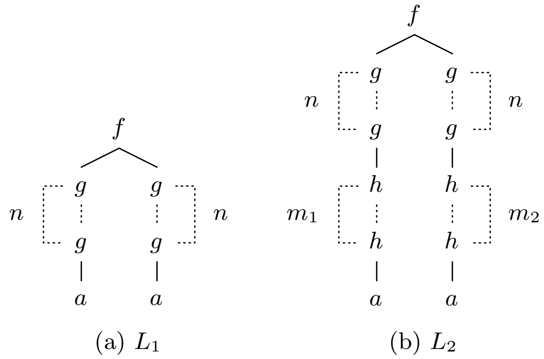


Fig. 7.3. Two term languages that are not regular

both deleting and pumping c yield only valid trees in L_2. Intuitively, we would like to force ADVERSARY to choose a decomposition that contains a g-labelled node, thus transferring our winning strategy for L_1—but this is not warranted by Lemma 7.2.1, which merely asserts that a pumpable context does exist *somewhere* in the tree, but does not allow us to delimit the exact region. □

7.2.2 Ogden's Lemma for Regular Term Languages

The pumping lemma that we prove in this section is more powerful than Lemma 7.2.1:

Lemma 7.2.2. *For every regular term language $L \subseteq T_\Sigma$, there is a number $p \geq 1$ such that every term $t \in L$ in which at least p nodes are marked as distinguished can be written as $t = c' \cdot c \cdot t'$ such that at least one node in c is marked, at most p nodes in $c \cdot t'$ are marked, and $c' \cdot c^n \cdot t' \in L$, for all $n \in \mathbb{N}$.* □

Note that, in the special case where all nodes are marked, this lemma reduces to Lemma 7.2.1.

Lemma 7.2.2 can be seen as the natural correspondent of Ogden's Lemma for context-free string languages [90]. Its contrapositive corresponds to the following modified game for term languages L: In the first turn, ADVERSARY has to choose a number $p \geq 1$. In the second turn, we have to choose a term $t \in L$ and mark at least p nodes in t. In the third turn, ADVERSARY has to choose a decomposition $c' \cdot c \cdot t'$ of t *in such a way that at least one node in c and at most p nodes in $c \cdot t'$ are marked.* In the fourth and final turn, we have to choose a number $n \in \mathbb{N}$ such that $c' \cdot c^n \cdot t' \notin L$.

Example 7.2.3 (continued). In the modified game, we can implement our idea from above to prove that the language L_2 from Example 7.2.2 is non-regular: we can always win the game by presenting ADVERSARY with the term

$$t \;=\; f(g^p \cdot h(a), g^p \cdot h(a))$$

and marking all nodes that are labelled with g as distinguished. Then, in whatever way Adversary decomposes t into segments $c' \cdot c \cdot t'$, the context c contains at least one node labelled with g, and the term $c' \cdot c^2 \cdot t'$ does not belong to L_2. □

Our proof of Lemma 7.2.2 builds on the following technical lemma:

Lemma 7.2.3. *Let Σ be a ranked alphabet. For every tree language $L \subseteq T_\Sigma$ and every $k \geq 1$, there exists a number $p \geq 1$ such that every term $t \in L$ in which at least p nodes have been marked as distinguished can be written as*

$$t \;=\; c' \cdot c_1 \cdots c_k \cdot t'$$

in such a way that for each $i \in [k]$, the context c_i contains at least one marked node, and the tree $c_1 \cdots c_k \cdot t'$ contains at most p marked nodes. □

Proof. Let m be the maximal rank of any symbol in Σ. Note that if m is zero, then each term over Σ has size one, and the lemma trivially holds with $p = 2$. For the remainder of the proof, assume that $m \geq 1$. Put $g_\Sigma(n) = \sum_{i=0}^{n} m^i$, and note that $g_\Sigma(n) < g_\Sigma(n+1)$, for all $n \in \mathbb{N}$. We will show that we can choose $p = g_\Sigma(k)$.

Let $t \in L$ be a term in which at least one node has been marked as distinguished. We call a node u of t *interesting*, if it either is marked, or has at least two children from which there is a path to an interesting node. It is easy to see from this definition that from every interesting node, there is a path to a marked node. Let $d(u)$ denote the number of interesting nodes on the path from the root node of t to u, excluding u itself. We make two observations about the function $d(u)$:

First, there is exactly one interesting node u with $d(u) = 0$. To see that there is at most one such node, let u_1 and u_2 be distinct interesting nodes with $d(u_1) = d(u_2) = n$; then the least common ancestor u of u_1 and u_2 is an interesting node with $d(u) = n - 1$. To see that there is at least one such node, recall that every marked node is interesting.

For the second observation, let u be an interesting node with $d(u) = n$. The number of interesting descendants v of u with $d(v) = n + 1$ is at most m. To see this, notice that each path from u to v starts with u, continues with some child u' of u, and then visits only non-interesting nodes w until reaching v. From each of these non-interesting nodes w, there is at most one path that leads to v. Therefore, the path from u to v is uniquely determined except for the choice of the child u', which is a choice among at most m alternatives.

Taken together, these observations imply that the number of interesting nodes u with $d(u) \leq k - 1$ is bounded by the value $g_\Sigma(k - 1)$.

Now, let t be a term in which at least $g_\Sigma(k)$ nodes have been marked as distinguished. Then there is at least one interesting node u with $d(u) = k$, and hence, at least one path that visits at least $k + 1$ interesting nodes. Choose any path that visits the maximal number of interesting nodes, and let $\vec{u}$ be a suffix of that path that visits exactly $k+1$ interesting nodes, call them $v_1, \dots, v_{k+1}$. We use $\vec{u}$ to identify a decomposition $c_1 \cdots c_k \cdot t'$ of t as follows: for each $i \in [k]$, choose v_i as the root node of c_i, choose v_{i+1} as the hole of c_i, and choose v_{k+1} as the root node of t'. This decomposition satisfies the required properties: To see that the term $c_1 \cdots c_k \cdot t'$ contains at most p marked nodes, notice that, by the choice of $\vec{u}$, no path in t that starts at v_1 contains more than $k+1$ interesting nodes, and hence the total number of interesting nodes in the subtree rooted at v_1 is bounded by $g_\Sigma(k) = p$. To see that every context c_i, $i \in [k]$, contains at least one marked node, let v be one of the interesting nodes in c_i, and assume that v is not itself marked. Then v has at least two children from which there is a path to an interesting, and, ultimately, to a marked node. At most one of these paths visits v_{i+1}; the marked node at the end of the other path is a node of c_i. ■

With Lemma 7.2.3 at hand, the proof of Lemma 7.2.2 is straightforward, and essentially identical to the proof given for the standard pumping lemma [28]:

Proof (of Lemma 7.2.2). Let $L \subseteq T_\Sigma$ be a regular term language, and let $G = (N, \Sigma, S, P)$ be a regular tree grammar in normal form that generates L. We will apply Lemma 7.2.3 with $k = |N|$. Let $t \in L$ be a term in which at least p nodes are marked as distinguished, where p is the number from Lemma 7.2.3. Then t can be written as $c' \cdot c_1 \cdots c_k \cdot t'$ such that for each index $i \in [k]$, the context c_i contains at least one marked node, and the term $c_1 \cdots c_k \cdot t'$ contains at most p marked nodes. Note that each context c_i, $i \in [k]$, is necessarily non-empty. Since G has only k nonterminals, there must be a nonterminal A and an index $i \in [k]$ such that $S \Rightarrow_G^* c' \cdot c_1 \cdots c_{i-1} \cdot A$ and either $A \Rightarrow_G^* c_i \cdot c_{i+1} \cdots c_{j-1} \cdot A$ and $A \Rightarrow_G^* c_j \cdot c_{j+1} \cdots c_k \cdot t'$, for some $i < j \leq [k]$, or $A \Rightarrow_G^* c_i \cdots c_k \cdot A$ and $A \Rightarrow_G^* t'$. Iterating the middle sub-derivations n times (where n may be zero), we obtain a new valid derivation. ■

Note that by choosing $k = m \cdot |N|$ in this proof, where $m \geq 1$, it is easy to generalize Lemma 7.2.2 as follows:

Lemma 7.2.4. *For every regular term language $L \subseteq T_\Sigma$ and every $m \geq 1$, there is a number $p \geq 1$ such that every term $t \in L$ in which at least p nodes are marked as distinguished can be written as $t = c' \cdot c_1 \cdots c_m \cdot t'$ such that for each $i \in [m]$, at least one node in c_i is marked, at most p nodes in $c_1 \cdots c_m \cdot t'$ are marked, and $c' \cdot c_1^n \cdots c_m^n \cdot t' \in L$, for all $n \in \mathbb{N}$.* □

7.3 Constant Growth

We have claimed (in our discussion of Lemma 7.1.3) that the block-degree restriction inherent to regular dependency languages formalizes the notion of 'limited cross-serial dependencies' that is characteristic for the class of mildly context-sensitive languages. In this section, we show that regular dependency languages also have another characteristic property of this class, *constant growth*. This property is usually attributed to string languages [53]; here, we define it for term languages.

7.3.1 Constant Growth and Semilinearity

Informally, a language has the constant growth property, if there are no arbitrarily long leaps in its size progression. More formally, let L be a term language, and let $\vec{n}$ be the sequence of distinct sizes of terms in L, sorted in ascending order. If L is of constant growth, then adjacent elements of $\vec{n}$ differ by at most a constant (see [62], Definition 5.1).

Definition 7.3.1. A term language $L \subseteq T_\Sigma$ is of *constant growth*, if either L is finite, or there is a number $c \in \mathbb{N}$ such that for each term $t \in L$, there exists a term $t' \in L$ such that $|t| < |t'| \leq |t| + c$. □

Example 7.3.1. We look at an example for a term language that does *not* have the constant growth property. Let Σ be a signature, and let L be the set of all complete binary terms over Σ. Given a term $t_i \in L$ with size $|t_i|$, a 'next larger' term $t_{i+1} \in L$ is obtained by adding two children to every leaf in t_i. This procedure results in a linear size progression: we see that $|t_{i+1}| = 2 \cdot |t_i| + 1$. In particular, there is no number c such that $|t_{i+1}| \leq |t_i| + c$ holds for all indices $i \in \mathbb{N}$. □

Constant growth is closely related to a property known as *semilinearity* [91]. This property is concerned with the interpretation of the elements of a language as multisets of labels. For the following definition, given a term $t \in T_\Sigma$ and a symbol $\sigma \in \Sigma$, we write $\#_\sigma(t)$ for the number of occurrences of σ in t.

Definition 7.3.2. Let Σ be a signature, and put $n := |\Sigma|$. We fix an (arbitrary) order on Σ and write σ_i for the ith symbol with respect to this order, for $i \in [n]$. The *Parikh mapping* for terms over Σ (with respect to this order) is the function $\psi_\Sigma \colon T_\Sigma \to \mathbb{N}^n$ defined as

$$\psi_\Sigma(t) \quad := \quad \langle \#_{\sigma_1}(t), \dots, \#_{\sigma_n}(t) \rangle \,.$$

We extend ψ_Σ to languages $L \subseteq T_\Sigma$ by putting $\psi_\Sigma(L) := \{\, \psi_\Sigma(t) \mid t \in L \,\}$. □

The order on Σ with respect to which a Parikh mapping is defined is convenient for our formal treatment of semilinearity, but irrelevant for our results. Therefore, we refer to *the* Parikh mapping for terms over Σ.

The Parikh mapping reduces a term to the multiset of its labels. A measure that is preserved under this reduction is the size of a term. More specifically, define the following norm on $\mathbb{N}^n$: $\|\vec{x}\| := \sum_{i=1}^{n} x_i$. Then for all terms $t \in T_\Sigma$, it holds that $\|\psi_\Sigma(t)\| = |t|$. The relevance of this observation is that it allows us to recast constant growth as a property of the image of a term language under its Parikh mapping.

Lemma 7.3.1. *A term language $L \subseteq T_\Sigma$ is of constant growth if and only if either $\psi_\Sigma(L)$ is finite, or there is a number $c \in \mathbb{N}$ such that for each term $t \in L$, there exists a term $t' \in L$ such that $\|\psi_\Sigma(t)\| < \|\psi_\Sigma(t')\| \leq \|\psi_\Sigma(t)\| + c$.* □

We now give a formal definition of semilinearity. To do so, we equip each set $\mathbb{N}^n$ with two operations: component-wise addition of two vectors ($\vec{x} + \vec{y}$), and multiplication of a vector by a scalar $a \in \mathbb{N}$ ($a \cdot \vec{x}$).

Definition 7.3.3. Let $n \in \mathbb{N}$. A set $S \subseteq \mathbb{N}^n$ is called *linear*, if there exists a vector $\vec{x}_0 \in \mathbb{N}^n$, a number $k \in \mathbb{N}$, and an indexed set $\langle\, \vec{x}_i \in \mathbb{N}^n \mid i \in [k] \,\rangle$ of vectors such that

$$S \;=\; \{\, \vec{x}_0 + \textstyle\sum_{i=1}^{k} c_i \cdot \vec{x}_i \mid c_i \in \mathbb{N} \,\} \,.$$

A set is called *semilinear*, if it is a finite union of linear sets. A language $L \subseteq T_\Sigma$ is called *linear* (*semilinear*), if $\psi_\Sigma(L)$ is a linear (semilinear) set of vectors. □

Each element of a semilinear language is the outcome of one of a finite number of generative processes. Such a process is specified by a single 'base structure' and a finite set of 'additives'. Its outcome is the set of all structures that can be obtained by combining the base structure with any number (including zero) of one or more additives. In this way, semilinearity is closely related to pumpability.

Lemma 7.3.2. *Each semilinear term language has the constant growth property.* □

Proof. Let $L \subseteq T_\Sigma$ be a term language. Put $n := |\Sigma|$, and $P := \psi_\Sigma(L)$. We show that, if P is linear or semilinear, then it satisfies the conditions of Lemma 7.3.1.

Assume that P is linear. In this case, there exists a vector $\vec{x}_0 \in \mathbb{N}^n$, a number $k \in \mathbb{N}$, and an indexed set $\langle\, \vec{x}_i \in \mathbb{N}^n \mid i \in [k] \,\rangle$ of base vectors such that

$$P \;=\; \{\, \vec{x}_0 + \textstyle\sum_{i=1}^{k} a_i \cdot \vec{x}_i \mid a_i \in \mathbb{N} \,\} \,.$$

A vector $\vec{x} \in \mathbb{N}^n$ is called *null*, if $\|\vec{x}\| = 0$. Distinguish two cases: If all base vectors are null, then P is finite. Otherwise, let $\vec{x}$ be a base vector that is not null and for which $\|\vec{x}\|$ is minimal among all non-null base vectors. Put $c := \|\vec{x}\|$, and let $\vec{y} \in P$. Since P is linear, the vector $\vec{z} := \vec{y} + \vec{x}$ is an element of P. Since $\vec{x}$ is not null, we have $\|\vec{y}\| < \|\vec{z}\| \leq \|\vec{y}\| + c$.

Now, assume that P is semilinear. In this case, there exists a number $m \in \mathbb{N}$ and an indexed family $\langle\, P_i \mid i \in [m] \,\rangle$ of linear sets such that $P = \bigcup_{i \in [m]} P_i$. Assume that P is non-finite. For every $i \in [m]$ for which the set P_i is non-finite, let c_i be the number constructed in the previous item. Put $c := \max_{i \in [m]} c_i$, and let $\vec{y} \in P$. Then there exists an index $i \in [m]$ such that $\vec{y} \in P_i$, and by the previous item, there exists a vector $\vec{z} \in P_i \subseteq P$ such that $\|\vec{y}\| < \|\vec{z}\| \leq \|\vec{y}\| + c_i \leq \|\vec{y}\| + c$.

Using Lemma 7.3.1, we conclude that L has the constant-growth property. ■

7.3.2 Regular Term Languages Are Semilinear

We now show that regular term languages are semilinear. Semilinearity of a language is routinely proven by providing an encoding of that language into a context-free language with the same Parikh image, and referring to Parikh's theorem [91]:

Proposition 1. *Every context-free language is semilinear.* □

Unfortunately, the standard proof of this theorem is rather opaque. In particular, it does not elucidate the close connection between semilinearity and pumpability. Therefore, we give a direct proof of the semilinearity of regular term languages, following a similar proof for context-free languages [32].

Theorem 7.3.1. *Every regular term language is semilinear.* □

Proof. Let L be a regular term language, and let G be a normalized regular term grammar with $L(G) = L$. For each set $M \subseteq N_G$ of non-terminals that contains S_G, let $L_M \subseteq L(G)$ be the subset of $L(G)$ that consists of all terms $t \in L(G)$ for which there is at least one derivation $S_G \Rightarrow_G^* t$ that uses exactly the non-terminals in M. Since there are only finitely many such sets L_M, and since their union is $L(G)$, it suffices to show that every set L_M is semilinear. Therefore, let us fix a set $M \subseteq N_G$, and put $m := |M|$ and $p := p(m)$, where the latter value is the constant from Lemma 7.2.4. We write $\Rrightarrow$ for the restriction of the derivation relation $\Rightarrow_G$ to derivations that use only rules of the form $A \to t$, where $A \in M$ and $t \in T_\Sigma(M)$. By the definition of L_M, it then holds that $t \in L_M$ if and only if $S \Rrightarrow^* t$. Put

$$\begin{aligned} T &:= \{\, t \in L_M \mid |t| < p \,\}\,, \\ C &:= \{\, c \in C_\Sigma \mid 1 \leq |c| \leq p \wedge \exists A \in M.\ A \Rrightarrow^* c \cdot A \,\}\,, \quad \text{and} \\ X &:= \{\, \psi_\Sigma(t) + \textstyle\sum_{c \in C} a_c \cdot \psi_\Sigma(c) \mid t \in T \wedge a_c \in \mathbb{N} \,\}\,. \end{aligned}$$

We start by noticing that the set X is semilinear: it is a finite union of linear sets, one for each term $t \in T$. To prove that the set L_M is semilinear, we show that $\psi_\Sigma(L_M) = X$.

$\subseteq$ Let $t \in L_M$ be a term. We show that $\psi_\Sigma(t) \in X$ by induction on the size $|t|$ of t. First assume that $|t| < p$. In this case, we see that $t \in T$, and $\psi_\Sigma(t) \in X$ by the definition of X. Now assume that $|t| \geq p$. In this case, if we mark all nodes in t as distinguished, then by Lemma 7.2.4, each derivation $S \Rrightarrow^* t$ can be written as

$$S \Rrightarrow^* c_0 \cdot A \Rrightarrow^* c_0 \cdot c_1 \cdot A \Rrightarrow^* \cdots \Rrightarrow^* c_0 \cdot c_1 \cdots c_m \cdot A \Rrightarrow^* c_0 \cdot c_1 \cdots c_m \cdot t'\,,$$

where $A \in M$, $c_0 \in C_\Sigma$, $c_i \in C$ for all $i \in [m]$, and $t' \in T_\Sigma$. (See the left half of Figure 7.4.) Let us write d_i for the sub-derivation $A \Rrightarrow^* c_i \cdot A$, and let $M' \subseteq M$ be the set of those non-terminals in $M - \{A\}$ that are used in some sub-derivation d_i, $i \in [m]$. For each $B \in M'$, choose some $i \in [m]$

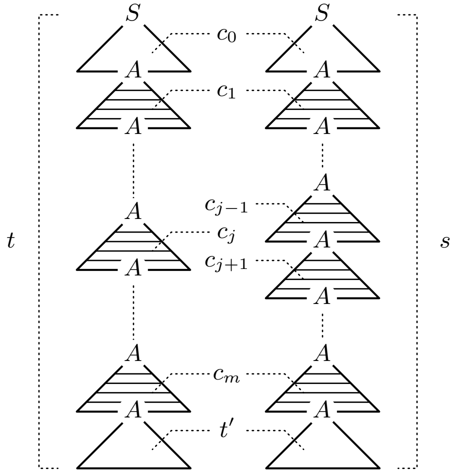


Fig. 7.4. Semilinearity

such that B is used in d_i. Then, since $|M'| < m$, some $j \in [m]$ is not chosen at all. Therefore, if the corresponding sub-derivation d_j is deleted, every non-terminal in M (including A) is still present in the resulting derivation. In this way, we obtain a new valid derivation for a term $s \in L_M$ with $|s| < |t|$. (See the right half of Figure 7.4.) By the induction hypothesis, we may assume that $\psi_\Sigma(s) \in X$. We see that $\psi_\Sigma(t) = \psi_\Sigma(s) + \psi_\Sigma(c_j)$, and so, $\psi_\Sigma(t) \in L_M$. Thus, in all cases, we have shown that $\psi_\Sigma(L_M) \subseteq X$.

$\supseteq$ Let $\vec{x} \in X$ be a vector. By the definition of X, there exists a term $t \in T$ and an indexed set $\langle\, a_c \in \mathbb{N} \mid c \in C\,\rangle$ such that $\vec{x} = \psi_\Sigma(t) + \sum_{c \in C} a_c \cdot \psi_\Sigma(c)$. We show that there exists a term $s \in L_M$ with $\psi_\Sigma(s) = \vec{x}$ by induction on $n := \sum_{c \in C} a_c$. First, assume that $n = 0$. In this case, we have $\vec{x} = \psi_\Sigma(t)$, and since $t \in L_M$, we deduce that $\psi_\Sigma(t) \in X$. Now, assume that $n > 0$. In this case, there exists a context $c \in C$ and a vector $\vec{x}' \in X$ such that $\vec{x} = \vec{x}' + \psi_\Sigma(c)$, and by the induction hypothesis, we may assume that there exists a term $t' \in L_M$ with $\vec{x}' = \psi_\Sigma(t')$. From the definition of C, we see that there is a non-terminal $A \in M$ and a derivation $A \Rrightarrow^* c \cdot A$. Since the derivation $S \Rrightarrow^* t'$ uses every non-terminal $B \in M$ (including A), it can be written as $S \Rrightarrow^* c' \cdot A \Rrightarrow^* c' \cdot t'' = t'$, for some context $c' \in C_\Sigma$ and term $t'' \in T_\Sigma(M)$. In particular, we have $A \Rrightarrow^* t''$. Plugging everything together, we can construct a valid derivation for a new term $s \in L_M$:

$$S \Rrightarrow^* c' \cdot A \Rrightarrow^* c' \cdot c \cdot A \Rrightarrow^* c' \cdot c \cdot t'' = s\,.$$

We see that $\vec{x} = \vec{x}' + \psi_\Sigma(c) = \psi_\Sigma(t') + \psi_\Sigma(c) = \psi_\Sigma(s)$. Thus, in all cases, we have shown that $X \subseteq \psi_\Sigma(L_M)$. ■

Corollary 1. *Every regular term language is of constant growth.* □

As an immediate consequence of this corollary and the isomorphism between regular dependency languages and regular term languages (Lemma 7.1.4), we obtain the main result of this section:

Theorem 7.3.2. *Every regular dependency language is of constant growth.* □

7.3.3 Related Work

To claim that every formal language that adequately models natural language should have the constant growth property is not claiming very much: Kracht [62] remarks that 'it seems that for every natural language [L] there is a number d_L such that for every $n \geq d_L$ there is a string of length n in L'. Semilinearity, on the other hand, may be too strong a restriction to impose on mathematical models of natural language: Michaelis and Kracht [79] show that an infinite progression of 'case stacking' in Old Georgian[3] would mean that this language is not semilinear. However, since there are no speakers of Old Georgian, there is no possibility to test whether this theoretical progression is actually possible.

The class of semilinear subsets of $\mathbb{N}^n$ is interesting in its own right. Among other things, it is closed under union, intersection, and complement. More generally, Ginsburg and Spanier [30] show that a subset of $\mathbb{N}^n$ is semilinear if and only if it is definable in Presburger arithmetic[4]. The class of languages with semilinear Parikh images forms an abstract family of languages, except that it is not closed under intersection with regular string languages [62, Theorem 2.93].

[3] Old Georgian is an extinct Caucasian language that was spoken roughly between the 4th and 11th century AD. It has a rich literary tradition.

[4] Presburger arithmetic is the first-order theory of the natural numbers with addition. It was named in honour of Mojżesz Presburger (1904–1943), who proved its decidability in 1929.

8

Generative Capacity and Parsing Complexity

In this chapter, we complete our study of regular dependency languages by investigating their string-generative capacity and parsing complexity. Specifically, we study the connection between these two measures and the structural constraints discussed in the first part of this book.

We start by explaining how regular dependency grammars can be extended to generators of sets of strings (Section 8.1). We then show that, for the string languages generated by these extended grammars, the block-degree measure induces an infinite hierarchy of expressiveness, and that the well-nestedness restriction properly decreases expressiveness on nearly all levels of this hierarchy (Section 8.2). Finally, we discuss the complexity of the parsing problem of the string languages generated by regular dependency grammars. In particular, we show that the well-nestedness condition can make the change between tractable and intractable parsing (Section 8.3).

8.1 Projection of String Languages

Up to this point, dependency structures were defined solely in terms of their governance and precedence relations. However, for many practical applications we are interested in *labelled* structures, where apart from the nodes and the edges, we also have ways to encode non-structural information such as word forms, grammatical functions, or edge probabilities. In this section, we extend our notion of dependency structures and dependency languages to accommodate such information. In particular, we show how dependency grammars can be understood as generators of string languages.

8.1.1 Labelled Dependency Structures

The extension to labelled structures is straightforward:

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 103–120, 2010.

Definition 8.1.1. Let A be some alphabet. An *A-labelled dependency structure* is a pair (D, lab), where D is a dependency structure, and lab: $\mathrm{dom}(D) \to A$ is a total function on the nodes of D, called the *labelling function.* □

Just as unlabelled dependency structures can be represented as terms over the alphabet Ω of order-annotations, A-labelled dependency structures can be represented as terms over the product alphabet $\langle \Omega, A\rangle$ in which each constructor $\langle \omega, a\rangle$ inherits the type of ω. For terms over this alphabet, we can extend the function dep in the natural way: the first component of a term constructor $\langle \omega, a\rangle$ carries the information about the dependency structure as such, the second component determines the label for the root node of the structure. In this way, each term over the signature $\langle \Omega, A\rangle$ denotes an A-labelled dependency structure. It is straightforward to extend our notion of dependency algebra accordingly. We can also define a string semantics for labelled dependency structures as follows. Recall that we use the notation $i\#j$ to refer to the jth occurrence of a symbol i in an order annotation.

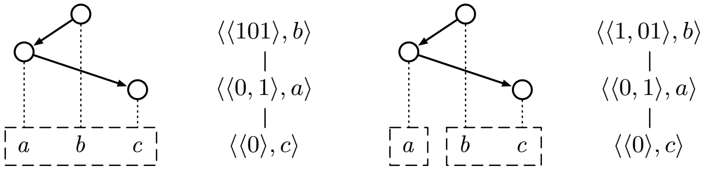
Definition 8.1.2. Let $\Sigma \subseteq \Omega$ be a finite set of order annotations, and let A be an alphabet. The *string algebra* over Σ and A is the $\langle \Sigma, A\rangle$-algebra in which $\mathrm{dom}(\mathfrak{A})_i = (A^+)^i$, for every $1 \leq i \leq \deg(\Sigma)$, and

$$f_{\langle \omega, a\rangle}(\vec{\alpha}_1, \dots, \vec{\alpha}_m) \quad = \quad \omega[0 \leftarrow a][\, i\#j \leftarrow \alpha_{i,j} \mid i \in [m] \wedge j \in [k_i]\,]\,,$$

for each constructor $\langle \omega, a\rangle$: $k_1 \times \cdots \times k_m \to k$ in $\langle \Omega, A\rangle$. □

Let $\langle \Sigma, A\rangle$ be some finite signature, where $\Sigma \subseteq \Omega$. Given a term d over this signature, we write $[\![d]\!]_{\mathbf{S}}$ for the evaluation of d in the string algebra over Σ and A and say that the labelled dependency structure that is denoted by d *projects* $[\![d]\!]_{\mathbf{S}}$. Notice that, if d denotes a dependency structure of sort k, then the projection of d is a k-tuple of (non-empty) strings over the alphabet A. For the case $k = 1$, we identify the set of one-component tuples of strings with the set of strings.

Example 8.1.1. Figure 8.1 shows two examples for labelled (segmented) dependency structures and their corresponding terms. Note that, in pictures of labelled structures, we annotate labels at the end of the corresponding projection lines. □



(a) D_1 (b) D_2

Fig. 8.1. Two labelled dependency structures and their terms

8.1.2 String-Generating Regular Dependency Grammars

With our algebraic framework in mind, it is straightforward to extend regular dependency grammars into generators of string languages (via projection). The only thing that we need to add to the existing signature is the alphabet of labels, now called *(surface) terminal symbols.*

Definition 8.1.3. Let $k \in \mathbb{N}$. A *string-generating regular dependency grammar* of degree k is a construct $G = (N, T, S, P)$, where N is a $[k]$-indexed family of *non-terminal alphabets*, T is an alphabet of *terminal symbols*, $S \in N$ is a distinguished *start symbol*, and $P \subseteq N \times T_{\langle \Omega(k), A \rangle}(N)$ is a k-indexed family of finite sets of productions. □

The derivation relation and the notion of the dependency language generated by a grammar are defined as usual, except that we are now dealing with labelled structures. The string algebra corresponding to a string-generating regular dependency grammar G is the string algebra over Σ_G and T_G, where Σ_G is the collection of those order annotations $\omega \in \Omega(k)$ that occur in the productions of G.

Definition 8.1.4. Let G be a string-generating regular dependency grammar. The *string language* projected by G is defined as $[\![G]\!]_{\mathbf{S}} := \{\, [\![d]\!]_{\mathbf{S}} \mid d \in L(G) \,\}$. □

Example 8.1.2. We give examples for two regular dependency grammars that generate the string language $\{\, a^n b^n \mid n \in \mathbb{N} \,\}$. The dependency structures generated by the first grammar are projective ('nested dependencies'), the structures generated by the second grammar may have block-degree 2 ('cross-serial dependencies'). Both grammars use the same terminal-alphabet $\{a, b\}$. We only state the productions of the grammars; sample terms and generated dependency structures are shown in Figure 8.2.

G_1 (projective dependency structures):

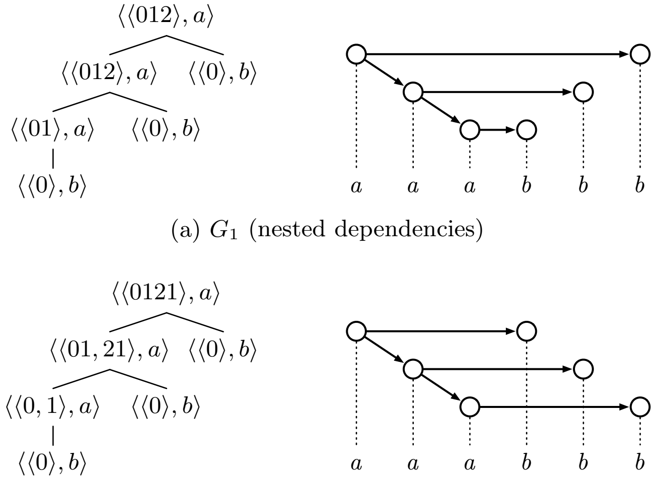
$$S \to \langle\langle 012\rangle, a\rangle(S, \langle\langle 0\rangle, b\rangle) \qquad S \to \langle\langle 01\rangle, a\rangle(\langle\langle 0\rangle, b\rangle)$$

G_2 (dependency structures with block-degree 2):

$$S \to \langle\langle 0121\rangle, a\rangle(R, \langle\langle 0\rangle, b\rangle) \qquad S \to \langle\langle 01\rangle, a\rangle(\langle\langle 0\rangle, b\rangle)$$
$$R \to \langle\langle 01, 21\rangle, a\rangle(R, \langle\langle 0\rangle, b\rangle) \qquad R \to \langle\langle 0, 1\rangle, a\rangle(\langle\langle 0\rangle, b\rangle)$$

8.1.3 String-Generative Capacity

It is apparent that our results on the equivalences between the dependency languages induced by various lexicalized grammar formalisms on the one hand



(a) G_1 (nested dependencies)

(b) G_2 (cross-serial dependencies)

Fig. 8.2. Derivations in two string-generating grammars

and classes of regular dependency languages over mildly non-projective dependency structures on the other hand can be transferred to string languages without any problems: the linearization semantics that we used for the unlabelled structures is fully compatible with the string semantics that we now use for labelled structures. However, one thing to note is, that all our results crucially depend on the grammars being lexicalized, meaning that each production in these grammars contributes an overt lexical item to the derived string—without this restriction, the notion of 'induced dependency structure' as we have used it here is ill-defined. Nevertheless, for some of the formalisms that we have studied, lexicalization is not really a restriction after all:

Lemma 8.1.1. *The string languages projected by REGD($\mathcal{D}_1$) are exactly the context-free languages.*[1] □

Proof. This follows from our previous results in combination with the result that every context-free grammar can be put into a lexicalized normal form, such as Greibach normal form [36] or Rosenkrantz normal form [101]. One caveat is that this transformation changes the structure of the derivation trees, and thus the dependency structures that we get out from these lexicalized grammars do not necessarily encode the same syntactic dependencies as the original grammar. ■

Lemma 8.1.2. *The string languages projected by REGD($\mathcal{D}_2 \cap \mathcal{D}_{wn}$) are exactly the string languages generated by TAGs.* □

[1] There is one minor difference: The context-free language that contains only the empty word cannot be projected by any regular dependency language.

Proof. This follows from our previous results in combination with the results that every TAG can be put into a lexicalized normal form [103]. ■

For LCFRS and CCFG, the problems whether every grammar can be put into some lexicalized normal form are open. These problems make an interesting topic for research for themselves, but are beyond the scope of this book.

8.2 String Languages and Structural Properties

In this section, we study the impact of structural constraints on the string-generative capacity of regular dependency languages. We present two results: first, that the string-language hierarchy known for LCFRS can be recovered in our framework by controlling the block-degree parameter; second, that additionally requiring well-nestedness leads to a proper decrease in generative capacity on nearly all levels of this hierarchy.

String-language hierarchies are usually proven using formalism-specific pumping lemmata. For more powerful formalisms, pumping arguments tend to become rather difficult and technical [see 105 or 38] because they need to reason about the combinatorial structure of the derivation and the order of the derived material at the same time. Our hierarchy proofs are novel in that they clearly separate these two issues: for the combinatorial aspect of the argument, we use only one powerful pumping lemma (Lemma 7.2.2); to reason about the order of the derived material, we use our knowledge about structural properties. With this proof technique, we can show that certain string languages 'enforce' certain structural properties in regular dependency languages that project them. The usefulness of this approach is witnessed by our result about the language hierarchy for well-nested languages, which solves an open problem concerning the relation between LCFRS and CCFG.

8.2.1 Masked Strings

To prepare our proofs, we first show two elementary results about congruence relations on strings. Recall (from Definition 4.1.1) that a congruence relation on a chain $\mathfrak{C}$ is an equivalence relation in which all blocks are convex with respect to $\mathfrak{C}$. Congruences on strings can be represented as lists of pairwise disjoint intervals of positions.

Definition 8.2.1. Let $s \in A^*$ be a string, and let $n \in \mathbb{N}$. A *mask* for s of length n is a list $M = [i_1, j_1] \cdots [i_n, j_n]$ of pairwise disjoint intervals of positions in s such that $j_k < i_{k+1}$, for all $k \in [n-1]$. It is understood that $i_k \leq j_k$, for all $k \in [n]$. □

We call the intervals $[i, j]$ the *blocks* of the mask M, and write $|M|$ to denote their number. In slight abuse of notation, we write $B \in M$, if B is a block of

M. Given a string s and a mask M for s, the set of *positions* corresponding to M is defined as

$$\mathrm{pos}([i_1, j_1] \cdots [i_n, j_n]) \;:=\; \{\, i \in \mathrm{pos}(s) \mid \exists k \in [n].\; i \in [i_k, j_k] \,\}\,.$$

For a set P of positions in a given string s, we put $\bar{P} := \mathrm{pos}(s) - P$, and write $[P]$ for the smallest mask for s such that $\mathrm{pos}(M) = P$. We say that P *contributes* to a block B of some mask, if $P \cap B \neq \emptyset$. Finally, for masks M with an even number of blocks, we define the *fusion* of M as

$$F([i_1, j_1][i'_1, j'_1] \cdots [i_n, j_n][i'_n, j'_n]) \;:=\; [i_1, j'_1] \cdots [i_n, j'_n]\,.$$

Lemma 8.2.1. *Let $s \in A^*$ be a string, let M be a mask for s with an even number of blocks, and let P be a set of positions in s such that both P and $\bar{P}$ contribute to every block of M. Then $|[P]| \geq |M|/2$. Furthermore, if $|[P]| \leq |M|/2$, then $P \subseteq pos(F(M))$.* □

Proof. For every block $B \in [P]$, let $n(B)$ be the number of blocks in M that B contributes to. We make two observations: First, since P contributes to each block of M, $|M| \leq \sum_{B \in [P]} n(B)$. Second, since $\bar{P}$ contributes to each block of M, no block $B \in [P]$ can fully contain a block of M; therefore, $n(B) \leq 2$ holds for all blocks $B \in [P]$. Putting these two observations together, we deduce that

$$|M| \;\leq\; \textstyle\sum_{B \in [P]} n(B) \;\leq\; \sum_{B \in [P]} 2 \;=\; 2 \cdot |[P]|\,.$$

For the second part of the lemma, let

$$M = [i_1, j_1][i'_1, j'_1] \cdots [i_n, j_n][i'_n, j'_n] \qquad \text{and} \qquad [P] = [k_1, l_1] \cdots [k_n, l_n]\,.$$

Then, each block of $[P]$ contributes to exactly two blocks of M. More precisely, for each $h \in [n]$, the block $[k_h, l_h]$ of $[P]$ contributes to the blocks $[i_h, j_h]$ and $[i'_h, j'_h]$ of M. (This situation is depicted in Figure 8.3.) Because $\bar{P}$ also contributes to $[i_h, j_h]$ and $[i'_h, j'_h]$, the interval $[k_h, l_h]$ is a proper subset of $[i_h, j'_h]$, which is a block of the fusion $F(M)$. Hence, $P \subseteq \mathrm{pos}(F(M))$. ■

8.2.2 Enforcing a Given Block-Degree

We now show our first result: for every natural number $k \in \mathbb{N}$, there exists a string language $L(k)$ that forces every regular dependency language that projects $L(k)$ to contain structures of block-degree k. For our proof, we use the string languages from the infinite family

$$\mathrm{COUNT}(k) \;:=\; \{\, a_1^n b_1^n \cdots a_k^n b_k^n \mid n \in \mathbb{N} \,\}\,.$$

We note that the language COUNT(1) is homomorphic to the context-free language $\{\, a^n b^n \mid n \in \mathbb{N} \,\}$ for which we have seen regular dependency grammars in Example 8.1.2, and that for every $k > 1$, the language COUNT(k) is not context-free; this can be easily proved using the standard pumping lemma for context-free languages.

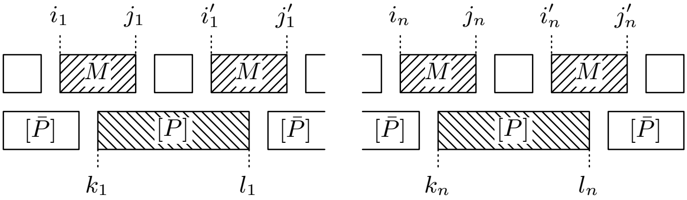


Fig. 8.3. The situation in the proof of Lemma 8.2.1

Example 8.2.1. The following grammar generates a dependency language that projects the string language COUNT(2); it is not hard to see how to modify the grammar so that it generates languages COUNT(k), for $k > 2$. The grammar is essentially identical to the TAG grammar that we gave in Figure 6.6. We only list the productions.

$$\begin{aligned}
S &\to \langle\langle 012314\rangle, a_1\rangle(R, \langle\langle 0\rangle, b_1\rangle, \langle\langle 0\rangle, a_2\rangle, \langle\langle 0\rangle, b_2\rangle)\\
S &\to \langle\langle 0123\rangle, a_1\rangle(\langle\langle 0\rangle, b_1\rangle, \langle\langle 0\rangle, a_2\rangle, \langle\langle 0\rangle, b_2\rangle)\\
R &\to \langle\langle 012, 314\rangle, a_1\rangle(R, \langle\langle 0\rangle, b_1\rangle, \langle\langle 0\rangle, a_2\rangle, \langle\langle 0\rangle, b_2\rangle)\\
R &\to \langle\langle 01, 23\rangle, a_1\rangle(\langle\langle 0\rangle, b_1\rangle, \langle\langle 0\rangle, a_2\rangle, \langle\langle 0\rangle, b_2\rangle)
\end{aligned}$$

Figure 8.4 shows a dependency structure generated by this grammar. We note that the structure is well-nested. □

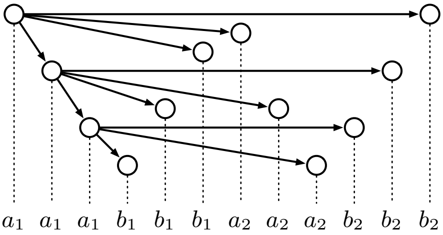

a_1 a_1 a_1 b_1 b_1 b_1 a_2 a_2 a_2 b_2 b_2 b_2

Fig. 8.4. A dependency structure for the language COUNT(2)

In the following proofs, we freely identify (segmented) labelled dependency structures with their corresponding terms. Given a term $d \in T_{\langle \Omega, A\rangle}$, we use the notation alph(d) to refer to the set of all labels from the alphabet A in d.

Lemma 8.2.2. *Let $k \in \mathbb{N}$. Every regular dependency language that projects COUNT(k) contains structures with a block-degree of at least k.* □

Proof. Let $L \in$ REGD be a regular dependency language that projects COUNT(k). For notational convenience, put $X := \{\, x_i \mid x \in \{a, b\} \wedge i \in [k] \,\}$.

We start with a simple auxiliary observation: Let s_1 and s_2 be two strings in $[\![L]\!]_{\mathbf{S}}$. If $|s_1| < |s_2|$, then every symbol from X occurs more often in s_2 than in s_1.

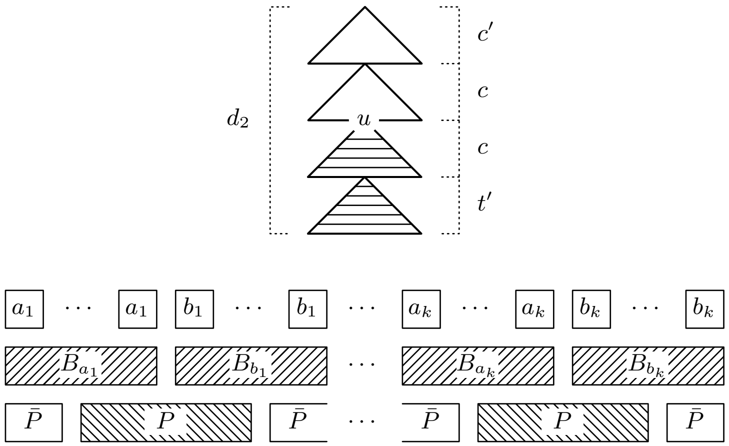

c'
c
d_2
u
c
t'
a_1 ⋯ a_1 b_1 ⋯ b_1 ⋯ a_k ⋯ a_k b_k ⋯ b_k
B_{a_1} B_{b_1} ⋯ B_{a_k} B_{b_k}
$\bar{P}$ P $\bar{P}$ ⋯ $\bar{P}$ P $\bar{P}$

Fig. 8.5. The situation in the proof of Lemma 8.2.2

Let p be the pumping number from Lemma 7.2.1, and let $d_1 \in L$ be a dependency structure with $[\![d_1]\!]_{\mathbf{S}} = a_1^n b_1^n \cdots a_k^n b_k^n$, where $n = \lceil p/2k \rceil$. Due to the isomorphism between $[\![d_1]\!]_{\mathbf{S}}$ and the precedence relation of d_1, we have $|d_1| = 2k \cdot n \geq p$. In this case, Lemma 7.2.1 asserts that d_1 can be written as $d_1 = c' \cdot c \cdot t'$ such that c contains at least one node, and $d_2 := c' \cdot c \cdot c \cdot t'$ belongs to L (see the upper part of Figure 8.5). Now, let u be the uniquely determined node in d_2 for which $d_2/u = c \cdot t'$ holds. As a consequence of the first item and the construction of d_2, we deduce that every symbol from X occurs in c. Hence, $X \subseteq \operatorname{alph}(c) \subseteq \operatorname{alph}(d_2/u)$.

We now show that u has block-degree k. Let $M = B_{a_1} B_{b_1} \cdots B_{a_k} B_{b_k}$ be the uniquely determined mask for $[\![d_2]\!]_{\mathbf{S}}$ in which each block B_{x_i} contains exactly those positions that correspond to occurrences of the symbol x_i (see the lower part of Figure 8.5), and let P be the set of those positions that correspond to the yield $\lfloor u \rfloor$. Since every symbol from X occurs in both P and its complement, both sets contribute to every block of M. With the first part of Lemma 8.2.1, we deduce that $|[P]| \geq k$. Due to the isomorphism between $[\![d_2]\!]_{\mathbf{S}}$ and the precedence relation of d_2, this means that the yield $\lfloor u \rfloor$ is distributed over at least k blocks in d_2. ∎

8.2.3 Enforcing Ill-Nestedness

We now show that even the well-nestedness constraint has an impact on the string-generative capacity of regular dependency languages. More specifically, for every natural number $k \in \mathbb{N}$, there exists a string language $L(k)$ that forces every regular dependency language over structures with a block-degree of at most k that projects $L(k)$ to contain ill-nested structures. For our proof, we use the languages from the family

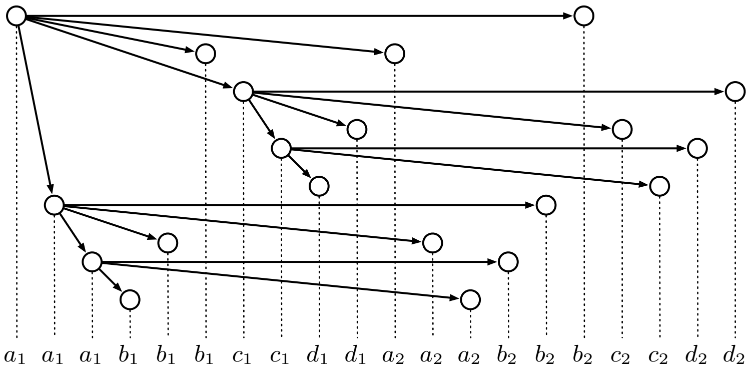

a_1 a_1 a_1 b_1 b_1 b_1 c_1 c_1 d_1 d_1 a_2 a_2 a_2 b_2 b_2 b_2 c_2 c_2 d_2 d_2

Fig. 8.6. A dependency structure for the language RESP(2)

$$\mathrm{RESP}(k) \;:=\; \{\, a_1^m b_1^m c_1^n d_1^n \cdots a_k^m b_k^m c_k^n d_k^n \mid m, n \in \mathbb{N} \,\}\,.$$

Similar to COUNT(k), the language RESP(k) is projected by a regular dependency language over the class $\mathcal{D}_k$ of dependency structures with block-degree at most k.

Example 8.2.2. The following grammar generates a dependency language that projects the string language RESP(2).

$$\begin{aligned}
S &\to \langle\langle 01234153\rangle, a_1\rangle(R_1, \langle\langle 0\rangle, b_1\rangle, R_2, \langle\langle 0\rangle, a_2\rangle, \langle\langle 0\rangle, b_2\rangle)\\
S &\to \langle\langle 012342\rangle, a_1\rangle(\langle\langle 0\rangle, b_1\rangle, R_2, \langle\langle 0\rangle, a_2\rangle, \langle\langle 0\rangle, b_2\rangle)\\
R_1 &\to \langle\langle 012, 314\rangle, a_1\rangle(R_1, \langle\langle 0\rangle, b_1\rangle, \langle\langle 0\rangle, a_2\rangle, \langle\langle 0\rangle, b_2\rangle)\\
R_1 &\to \langle\langle 01, 23\rangle, a_1\rangle(\langle\langle 0\rangle, b_1\rangle, \langle\langle 0\rangle, a_2\rangle, \langle\langle 0\rangle, b_2\rangle)\\
R_2 &\to \langle\langle 012, 314\rangle, c_1\rangle(R_2, \langle\langle 0\rangle, d_1\rangle, \langle\langle 0\rangle, c_2\rangle, \langle\langle 0\rangle, d_2\rangle)\\
R_2 &\to \langle\langle 01, 23\rangle, c_1\rangle(\langle\langle 0\rangle, d_1\rangle, \langle\langle 0\rangle, c_2\rangle, \langle\langle 0\rangle, d_2\rangle)
\end{aligned}$$

Figure 8.6 shows a dependency structure generated by this grammar. Note that this structure is ill-nested. This is mirrored in the grammar by the fact that the first order annotation contains the forbidden substring 1313. □

Lemma 8.2.3. *Let $k > 1$. Every regular dependency language $L \in$ REGD($\mathcal{D}_k$) that projects RESP(k) contains ill-nested structures.* □

Proof. Let $L \in \mathrm{REGD}(\mathcal{D}_k)$ be a regular dependency language that projects RESP(k). Define the following two sets of symbols:

$$X \;:=\; \{\, x_i \mid x \in \{a, b\} \wedge i \in [k] \,\}\,, \quad \text{and} \quad Y \;:=\; \{\, y_i \mid y \in \{c, d\} \wedge i \in [k] \,\}\,.$$

We start with a simple observation: Let d_1 and d_2 be dependency structures contained in L. If at least one symbol from X occurs more often in d_2 than in d_1, then every symbol from X does. A symmetric argument holds for Y.

Let p be the pumping number from Lemma 7.2.2, and let $d \in L$ be a dependency structure with $[\![d]\!]_{\mathbf{S}} = a_1^m b_1^m c_1^n d_1^n \cdots a_k^m b_k^m c_k^n d_k^n$, where $m = n = \lceil p/2k \rceil$. By this choice, the structure d contains $2k \cdot m \geq p$ occurrences of symbols from X, and equally many occurrences of symbols from Y.

Lemma 7.2.2 asserts that the structure d can be written as $d = c' \cdot c \cdot t'$ such that the context c contains at least one occurrence of a symbol from X, and the 'pumped' structure $d_X := c' \cdot c \cdot c \cdot t'$ is contained in L. Let u_X be the uniquely determined node in d for which $d/u_X = c \cdot t'$. We want to show that $\mathrm{alph}(d/u_X) = X$.

$\supseteq$ Since c contains at least one occurrence of a symbol from X, at least one symbol from X occurs more often in d_X than in d. Then, by our observation above, *every* symbol from X occurs more often in d_X than in d. By the construction of d_X, this implies that $X \subseteq \mathrm{alph}(c) \subseteq \mathrm{alph}(d/u_X)$.

$\subseteq$ Let $M_X = B_{a_1} B_{b_1} \cdots B_{a_k} B_{b_k}$ be the uniquely determined mask for $[\![d_X]\!]_{\mathbf{S}}$ in which each block B_{x_i} contains exactly those positions that are labelled with the symbol x_i. Furthermore, let u be the uniquely determined node in d_X for which $d_X/u = c \cdot t'$, and let P be the set of those positions in $[\![d_X]\!]_{\mathbf{S}}$ that correspond to the yield $\lfloor u \rfloor$. We now apply Lemma 8.2.1: given that $X \subseteq \mathrm{alph}(d_X/u)$, both the set P and its complement contribute to every block of M; given that $d_2 \in \mathcal{D}_k$, we have $|[P]| \leq k$. From this, we deduce that $P \subseteq \mathrm{pos}(F(M_X))$. Since every position in the set $\mathrm{pos}(F(M_X))$ is labelled with a symbol from X (see Figure 8.7), we conclude that $\mathrm{alph}(d_X/u) = \mathrm{alph}(d/u_X) \subseteq X$.

Put $Y := \{\, x_i \mid x \in \{c, d\} \wedge i \in [k] \,\}$. In symmetry to the argument above, we can show the existence of a node u_Y in d_1 for which $\mathrm{alph}(d_1/u_Y) = Y$.

Due to the isomorphism between $[\![d_1]\!]_{\mathbf{S}}$ and the precedence relation of d_1, the yields $\lfloor u_X \rfloor$ and $\lfloor u_Y \rfloor$ interleave. Since the sets X and Y are disjoint, neither $u_X \trianglelefteq u_Y$ nor $u_Y \trianglelefteq u_X$ holds. We conclude that d_1 is ill-nested. ■

8.2.4 Hierarchies of String Languages

We are now ready to present the main result of this section: the hierarchy on regular dependency languages from Lemma 7.1.2 carries over to string languages.

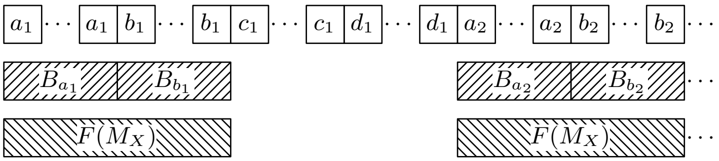

a_1 ··· a_1 b_1 ··· b_1 c_1 ··· c_1 d_1 ··· d_1 a_2 ··· a_2 b_2 ··· b_2 ···
B_{a_1}
B_{b_1}
B_{a_2}
B_{b_2} ···
$F(M_X)$
$F(M_X)$ ···

Fig. 8.7. Enforcing ill-nestedness

Theorem 8.2.1. *The following relations hold for all* $k \in \mathbb{N}$*:*

- $[\![\mathit{REGD}(\mathcal{D}_k)]\!]_S \subsetneq [\![\mathit{REGD}(\mathcal{D}_{k+1})]\!]_S$
- $[\![\mathit{REGD}(\mathcal{D}_k \cap \mathcal{D}_{wn})]\!]_S \subsetneq [\![\mathit{REGD}(\mathcal{D}_{k+1} \cap \mathcal{D}_{wn})]\!]_S$
- $[\![\mathit{REGD}(\mathcal{D}_1)]\!]_S = [\![\mathit{REGD}(\mathcal{D}_1 \cap \mathcal{D}_{wn})]\!]_S$
- $k \neq 1 \implies [\![\mathit{REGD}(\mathcal{D}_k \cap \mathcal{D}_{wn})]\!]_S \subsetneq [\![\mathit{REGD}(\mathcal{D}_k)]\!]_S$

Proof. The inclusions in the first two items as well as the third item are immediate consequences of Lemma 7.1.2. The properness of the inclusions and the last item follow from Lemmata 8.2.2 and 8.2.3 and the facts that $\mathrm{COUNT}(k) \in [\![\mathrm{REGD}(\mathcal{D}_k \cap \mathcal{D}_{wn})]\!]_{\mathrm{S}}$ and $\mathrm{RESP}(k) \in [\![\mathrm{REGD}(\mathcal{D}_k)]\!]_{\mathrm{S}}$, as witnessed by the grammars we gave above. ■

The hierarchy established by the first item corresponds to the string-language hierarchy known for LCFRS [37, 119] and other formalisms that generate the same string languages (see e.g. 24, 97, 105, 116).

8.2.5 Related Work

The language RESP(2) was first considered by Weir [119], who speculated that it separates the string-languages generated by LCFRS with fan-out 2 from the languages generated by TAG. This was subsequently proved by Seki et al. [105].

Gramatovici and Plátek [35] study a string-language hierarchy on a dependency formalism in which derivations can be controlled by the node-gaps complexity parameter that we discussed in Section 4.1.3.

8.3 Parsing Complexity

The parsing problem of regular dependency languages is the problem to find, given a grammar and a string of terminal symbols, (a compact description of) the set of all dependency structures generated by the grammar that project the string. In this section, we show that regular dependency languages can be parsed in time polynomial in the length of the input string, but that the parsing problem in which the grammar is part of the input is NP-complete even for a fixed block-degree. However, we also show that the same problem becomes polynomial when grammars are restricted to well-nested order annotations, and hence, to well-nested dependency languages. Together with the treebank evaluation that we presented in Chapter 5, this result provides strong evidence that our interest in the well-nestedness condition is justified.

8.3.1 Membership Problems

Instead of looking at the parsing problem of regular dependency languages directly, we restrict ourselves to a slightly simpler problem: the problem to decide, given a grammar and a string, whether the grammar generates any dependency structure at all that projects the string. This problem is the *membership problem* of the projected string language. For the vast majority of the algorithms that solve membership problems for generative grammars, including the ones that we discuss here, there are standard ways to extend them into full parsers, so the restriction to the membership problem is minor. The membership problem comes in two variants, depending on whether we consider the grammar to be part of the input to the problem or not:

Definition 8.3.1. The *(standard) membership problem* for a regular dependency grammar G is the following decision problem: given a string $\vec{a}$, is $\vec{a} \in [\![L(G)]\!]_{\mathbf{S}}$? The *uniform membership problem* for a class $\mathcal{G}$ of regular dependency grammars is the following decision problem: given a grammar $G \in \mathcal{G}$ and a string $\vec{a}$, is $\vec{a} \in [\![L(G)]\!]_{\mathbf{S}}$? □

The uniform membership problem is at least as difficult as the standard membership problem, but it may be more difficult. In particular, every polynomial-time algorithm that solves the uniform membership problem also solves the standard membership problem in polynomial time. On the other hand, an algorithm for the standard membership problem may take an amount of time that is exponential in size factors that depend on the grammar. In this case, it does not yield a polynomial-time algorithm for the universal membership problem.

In the computer science literature, the run-time of parsing algorithms is usually given as a function of the length of the input string, which is informative only for the standard membership problem. One of the reasons for the disinterest in the size of the grammar may be that, in many applications, grammars are small, and the candidate string is long—consider the grammar of a programming language for example, which usually only fills a few pages, but may be used in compilers that process ten thousands lines of code. This situation does not apply to computational linguistics, where rather the opposite is true: sentences are short, not more than a hundred words, while grammars are huge, with several hundreds of thousands of entries. Thus, for the parsing of natural language, the important measure in the analysis of parsing algorithms is not the length of the input string, but the size of the grammar (cf. [70]). This holds true in particular when we consider lexicalized grammars, where all productions are specialized for individual words. At the same time, these grammars have the advantage that parsing needs to consider only those productions that are associated with the words in the input string [104]. While

$$\begin{aligned}
\mathcal{A} &:= \{\, [a, [i, i]] \in T \times B \mid a_i = a \,\} \\
\mathcal{I} &:= \{\, [A, M] \in N \times B^* \mid |M| = \deg(A) \,\} \\
\mathcal{G} &:= \{[S, [1, n]]\}
\end{aligned}$$

Fig. 8.8. Axioms, items and goals for the grammatical deduction system

this strategy reduces the parsing time in all practical cases, it also introduces an additional factor into the complexity analysis of parsing algorithms that depends on the length of the input string (cf. [21]).

8.3.2 The Standard Membership Problem

As our first technical result of this section, we now show that the standard membership problem for regular dependency grammars is polynomial. To prove this, we construct a generic recognition algorithm for a regular dependency grammar G in the framework of deductive parsing [108]. Let us write k_G for the degree of G (which corresponds to the maximal block-degree among the structures in the language generated by G), and m_G for the maximal rank of G (which corresponds to the maximal out-degree of the language generated by G).

Lemma 8.3.1. *The membership problem of string languages that are projected by regular dependency languages is in time $O(|P| \cdot n^e)$, where $e = k_G \cdot (m_G + 1)$.* □

Proof. Let L be a regular dependency language, and let $\vec{a}$ a string over some alphabet. Furthermore, let $G = (N, T, S, P)$ be a normalized regular dependency grammar that generates L. To decide whether $\vec{a} \in [\![L]\!]_{\mathbf{S}}$, we construct a *grammatical deduction system* for G, and use a generic implementation of this system in the framework of deductive parsing [108].

Put $n := |\vec{a}|$, and let $B \subseteq [n] \times [n]$ be the set of all intervals of positions in the string $\vec{a}$. A grammatical deduction system consists of four components: a set $\mathcal{A}$ of *axioms*, a set $\mathcal{I}$ of *items*, a set $\mathcal{G} \subseteq \mathcal{I}$ of *goal items*, and a finite collection of *inference rules* over $\mathcal{A}$ and $\mathcal{I}$. The sets of axioms, items and goal items of our deduction system are defined in Figure 8.8.

The axioms represent the information about which position in $\vec{a}$ is labelled by which terminal symbol. An item $[A, [i_1, j_1] \cdots [i_k, j_k]]$ asserts that there is a dependency structure $d \in L(G)$ such that $[\![d]\!]_{\mathbf{S}} = \langle a_{i_1} \cdots a_{j_1}, \dots, a_{i_k} \cdots a_{j_k} \rangle$; in particular, the goal item asserts that $\vec{a} \in [\![L]\!]_{\mathbf{S}}$. The set of inference rules is constructed as follows. For each production $A \to \langle \omega, a \rangle (A_1, \dots, A_m)$ with $\omega \colon k_1 \times \cdots \times k_m \to k$, we use an inference rule of the form

$$\frac{[a, b_{0,1}] \qquad [A_1, b_{1,1} \cdots b_{1,k_1}] \qquad \cdots \qquad [A_m, b_{m,1} \cdots b_{m,k_m}]}{[A, b_1 \cdots b_k]}$$

This rule is subject to the following side conditions, which reflect the semantics of the order annotation ω. Assume that $\omega = \langle \vec{\imath}_1, \dots, \vec{\imath}_k \rangle$. We write ℓ_x for the left endpoint of the interval b_x, and r_x for the corresponding right endpoint.

$$r_{0,1} = \ell_{0,1} \quad \Longleftarrow \quad \tag{8.1}$$

$$\ell_{i_2,j_2} = r_{i_1,j_1} + 1 \quad \Longleftarrow \quad \exists h \in [k].\ \vec{\imath}_h = \vec{x} \cdot i_1\#j_1 \cdot i_2\#j_2 \cdot \vec{y} \tag{8.2}$$

$$\ell_h = \ell_{i,j} \quad \Longleftarrow \quad \vec{\imath}_h = i\#j \cdot \vec{x} \tag{8.3}$$

$$r_h = r_{i,j} \quad \Longleftarrow \quad \vec{\imath}_h = \vec{x} \cdot i\#j \tag{8.4}$$

The first condition reflects the semantics of the axioms. The second condition ensures that blocks that are adjacent in ω correspond to intervals of positions that are adjacent in $\vec{a}$. The third and fourth condition ensure that blocks that are extremal in ω correspond to extremal intervals in $\vec{a}$. Taken together, the conditions ensure that each inference rule is sound with respect to the intended semantics. Their completeness is obvious. Thus, we have $\vec{a} \in [\![L]\!]_{\mathbf{S}}$ if and only if starting from the axioms, we can deduce the goal item.

The asymptotic runtime of the generic, chart-based implementation of the grammatical deduction system for G is $O(|P| \cdot n^e)$, where e is the maximal number of free variables per inference rule that range over the domain $[n]$ (see [73]). To determine e, we inspect the schema for the inference rules above. The total number of variables over $[n]$ in this schema is $2+2k+\sum_{i=1}^{m} 2k_i$. Each non-free variable is determined by exactly one of the side conditions. Therefore, to determine the number of free variables in the rule schema, it suffices to count the instantiations of the schemata for the side conditions, and to subtract this number from the total number of variables. Schema 8.1 has 1 instantiation. Schemata 8.3 and 8.4 each have k instantiations; this is the number of leftmost and rightmost positions in the blocks of ω, respectively. Finally, schema 8.2 has $1 - k + \sum_{i=1}^{m} k_i$ instantiations: the string $\vec{\imath}$ has $1 + \sum_{i=1}^{m} k_i$ positions; k of these mark the end of a block and thus do not have a neighbouring symbol. Then the number of free variables is

$$\left(2 + 2k + \sum_{i=1}^{m} 2k_i\right) - \left(1 + 2k + 1 - k + \sum_{i=1}^{m} k_i\right) \;=\; k + \sum_{i=1}^{m} k_i \,.$$

Thus, $e \leq k_G \cdot (m_G + 1)$. ■

Theorem 8.3.1. *The membership problem of* $[\![\mathit{REGD}]\!]_{\mathbf{S}}$ *is in PTIME.* □

This result, together with our previous result about the constant growth of the languages in REGD (Theorem 7.3.2), allows us to call regular dependency languages mildly context-sensitive, according to Joshi's [1985] characterization.

8.3.3 The Uniform Membership Problem

The complexity of the generic parsing algorithm for regular dependency languages that we gave in the previous section is exponential both in the degree and in the rank of the grammar that is being processed. This means that we are punished both for languages with a high degree of non-projectivity, and for languages with a high number of dependents per node. A natural question to ask is, whether we can get substantially better than this. Unfortunately, at least in the general case, the answer to this question is probably negative: in this section, we show that the uniform membership problem for the class of regular dependency grammars is NP-complete. Given the close connection between regular dependency grammars and LCFRS, this result does not come entirely unexpected: Satta [102] showed that the uniform membership problem of both LCFRS with restricted fan-out (our block-degree) and restricted rank is NP-hard. Unfortunately, we cannot directly apply his reduction (of the 3SAT problem) to the membership problem of regular dependency languages, as this reduction makes essential use of concatenation functions with empty components, which we have excluded (see Section 6.2.3).

Instead, we provide a polynomial reduction of the EXACT COVER problem to the uniform membership problem of regular dependency grammars, UNIFORM-REGD. An instance of EXACT COVER is given by a finite set U and a finite collection $\mathcal{F}$ of subsets of U. The decision to make is, whether there is a subset $\mathcal{C} \subseteq \mathcal{F}$ such that the sets in $\mathcal{C}$ are disjoint, and their union is U.

Lemma 8.3.2. EXACT COVER $\leq_p$ UNIFORM-REGD □

Proof. Let $I = (U, \mathcal{F})$ be any instance of the EXACT COVER problem. Put $n := |U|$, and $m := |\mathcal{F}|$, and assume that the elements of U and $\mathcal{F}$ are numbered from 1 to n and 1 to m, respectively. We write u_i to refer to the ith element of U, and S_i to refer to the ith element of $\mathcal{F}$ with respect to these numberings. The main idea behind the following reduction is to construct a regular dependency grammar $G = (N, T, S, P)$ and a string $\vec{a}$ such that each dependency structure that is generated by G and projects $\vec{a}$ represents a partition $\mathcal{C} \subseteq \mathcal{F}$ of U. The string $\vec{a}$ has the form $\$ \cdot \vec{x}_1 \cdots \vec{x}_m \cdot \vec{x}$, where the substring $\vec{x}$ is a representation of the set U, and each substring $\vec{x}_i$, $i \in [m]$, controls whether the set S_i is included in $\mathcal{C}$. The grammar G is designed such that each substring $\vec{x}_i$ can be derived in only two possible ways and only as the projection of the first block of a dependency structure with block-degree 2; the second block of this structure projects material in the string $\vec{x}$. In this way, each derivation corresponds to a guess which sets of $\mathcal{F}$ to include into $\mathcal{C}$. The string $\vec{x}$ is set up to ensure that this guess is consistent.

We first describe the construction of the string $\vec{a}$. Each string $\vec{x}_i$, $i \in [m]$, has the form $\$y_1\# \cdots \#y_n\$$, where for all $j \in [n]$, $y_j = u_j$ if $u_j \in S_i$, and $y_j = \bar{u}_j$ otherwise. The string $\vec{a}$ then has the form $\$ \cdot \vec{x}_1 \cdots \vec{x}_m \cdot \bar{u}_1^m u_1 \bar{u}_1^m \cdots \bar{u}_n^m u_n \bar{u}_n^m$.

Next, we describe the construction of the grammar. The non-terminal and terminal alphabets are defined as follows:

$$\begin{aligned} N &:= \{1 \mapsto \{S\},\ 2 \mapsto \{\, [S_i, u_j] \mid i \in [m] \wedge j \in [n] \,\}\} \\ T &:= \{\$, \#\} \cup \{\, u_i \mid i \in [n] \,\} \cup \{\, \bar{u}_i \mid i \in [n] \,\} \end{aligned}$$

The start symbol is S. Before we give the set of production, we introduce the following abbreviating notation: for every terminal symbol $a \in T$, put

$$\begin{aligned} \langle a\vec{a}, \vec{a}' \rangle &:= \langle \langle 01, 1 \rangle, a \rangle (\langle \vec{a}, \vec{a}' \rangle) & \qquad \langle a, \vec{a} \rangle &:= \langle \langle 0, 1 \rangle, a \rangle (\langle \vec{a} \rangle) \\ \langle a\vec{a} \rangle &:= \langle \langle 01 \rangle, a \rangle (\langle \vec{a} \rangle) & \langle a \rangle &:= \langle \langle 0 \rangle, a \rangle \end{aligned}$$

Now for each set $S \in \mathcal{F}$, each element $u \in U$, and all $i, j \in [m]$, we introduce the following productions:

$$\begin{array}{lcllcll}
[S,u] & \to & \langle \$u, \bar{u}^i u \bar{u}^j \rangle & [S,u] & \to & \langle \$\bar{u}, \bar{u} \rangle & \text{(first selected/not selected)} \\
[S,u] & \to & \langle \# u, \bar{u}^i u \bar{u}^j \rangle & [S,u] & \to & \langle \# \bar{u}, \bar{u} \rangle & \text{(selected/not selected)} \\
[S,u] & \to & \langle \# u\$, \bar{u}^i u \bar{u}^j \rangle & [S,u] & \to & \langle \# \bar{u}\$, \bar{u} \rangle & \text{(last selected/not selected)}
\end{array}$$

We also need the production $S \to \langle\langle 0\vec{x}\vec{y}\rangle, \$\rangle([S_1,u_1],\dots,[S_m,u_n])$, where $\vec{x}$ is the row-wise reading of the $n \times m$-matrix in which each cell (i,j), $i \in [n]$, $j \in [m]$, contains the value $i + n \cdot (j-1)$, and $\vec{y}$ is the column-wise reading of this matrix.

We now claim that each substring $\vec{x}_i$, $i \in [m]$, can be derived in only two possible ways: either by rules from the group 'selected', or by rules from the group 'not selected'. Within such a group, each terminal can only be generated by exactly one rule, depending on the position of the terminal in the sub-string (first, inner, last) and the form of the terminal $(u, \bar{u})$. In this way, each derivation of $\vec{x}_i$ corresponds to a choice whether S_k should be part of $\mathcal{C}$ or not. If it is chosen, the second components of the rules consume the single terminal $\vec{u}$ in the right half of the string, along with all 'garbage' (in the form of superfluous symbols $\bar{u}$) adjacent to it. No terminal u in the right half of the string can be consumed twice; this reflects the fact that the S_k must be disjoint. If the derivation is complete, all terminals on the right side have been consumed; this reflects the fact that the union of the S_k makes the complete set. ■

Note that the grammar constructed in the proof of this lemma has degree 2, but that its maximal rank grows with the input.

Example 8.3.1. Figure 8.9 shows an example for the encoding in the proof of Lemma 8.3.2 for the instance $U = \{u_1, u_2\}$, $\mathcal{F} = \{\{u_1\}, \{u_2\}\}$. □

Theorem 8.3.2. *The uniform string membership problem for the class of normalized regular dependency grammars is NP-complete.* □

Proof. Lemma 8.3.2 establishes the NP-hardness of the problem; the grammar that we used for the reduction is normalized. To see that the problem is in NP, we notice that the length of a derivation in a normalized regular dependency grammar directly corresponds to the length of the input sentence. Therefore, we can check whether a given candidate derivation is valid in polynomial time: if the derivation is longer then the sentence, we reject it; otherwise, we

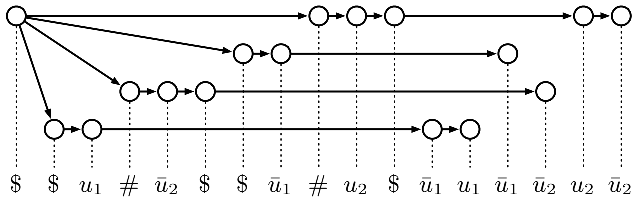

$ $ u_1 # $\bar{u}_2$ $ $ $\bar{u}_1$ # u_2 $ $\bar{u}_1$ u_1 $\bar{u}_1$ $\bar{u}_2$ u_2 $\bar{u}_2$

Fig. 8.9. The encoding in the proof of Lemma 8.3.2

compute the string value of the derivation using a variant of the tree traversal algorithm that we presented in Table 4.1. ■

8.3.4 Recognition of Well-Nested Languages

The NP-completeness of the uniform membership problem of regular dependency grammars makes it unlikely that we can find parsing algorithms that are considerably more efficient than the generic algorithm that we gave in the proof of Lemma 8.3.1, not even for grammars of some fixed degree. In this respect, regular dependency grammars are fundamentally different from CFG or even TAG, where the maximal rank of the productions of the grammar does not enter into the runtime as an exponential factor. Satta [102], who made the same observation about LCFRS, argued that the fundamental reason for the leap between polynomial parsing for CFG and TAG and the NP-hardness result for LCFRS(2) could be due to the presence of what he called 'crossing configurations' in the derivations of LCFRS(2). He concluded that to bridge the gap in parsing complexity, 'the addition of restrictions on crossing configurations should be seriously considered for the class LCFRS'. We now show that the well-nestedness condition could well be such a restriction: the uniform membership problem for well-nested regular dependency grammars is in PTIME.

Theorem 8.3.3. *Let $k \in \mathbb{N}$. The uniform string membership problem for the class of well-nested regular dependency grammars of degree k can be solved in time $O(|G|^2 \cdot n^{2k+2})$.* □

Proof. Every string-generating regular dependency grammar G of degree k that makes use of well-nested order annotations only can be transformed into a CCFG G' of rank k that is equivalent to G with respect to the generated dependency language, and hence, with respect to the generated string language. This transformation can be done in time linear in the size of the grammar, and without an increase in size: essentially, we replace all productions of the dependency grammar by their equivalents under the relabelling that we presented in Section 6.3. The membership problem of G' can be decided in time $O(|G'|^2 \cdot n^{2k+2})$ [49], where $|G'|$ is the size of the grammar. Consequently, the membership problem of G can be decided in time $O(|G|^2 \cdot n^{2k+2})$. ■

Together with the treebank evaluation that we presented in Chapter 5, this result is strong evidence that the well-nestedness condition is a very relevant condition on dependency structures indeed. A promising objective of future research is to understand the fundamental differences between well-nested and ill-nested grammars in more detail. In particular, it would be interesting to see how the generic algorithm for the parsing of regular dependency grammars can be modified to make use of the well-nestedness condition, and to give a more fine-grained complexity analysis that reverences the lexicalized nature of dependency grammars.

8.3.5 Related Work

Besides the result that we present here, there are several other results that show that grammar-driven dependency parsing can be intractable. Neuhaus and Bröker [83] use a reduction of the VERTEX COVER problem to prove that the uniform membership problem for a class of linguistically adequate, minimal dependency grammars is NP-complete. Koller and Striegnitz [61] show a corresponding result for the formalism Topological Dependency Grammar [19] using a reduction of HAMILTON CYCLE.

Gómez-Rodríguez et al. [33] present parsing algorithms for well-nested dependency trees under various restrictions, obtaining a parsing complexity of $O(n^{2k+5})$. Their algorithms are based (implicitly) on a different algebraic structure for well-nested dependency trees, in which the dependents of a given node are not attached all at once, but one one-by-one, very similar to the parser for projective dependency trees presented by Eisner [20]. This allows them to uncouple parsing complexity from the rank parameter m_G, which in the proof of Theorem 8.3.3 disappears during the normal form-transformation described by Hotz and Pitsch [49]. Gómez-Rodríguez et al. [34] provide a different such normal form for well-nested LCFRS.

9

Conclusion

In this book, we have shown that the generative capacity and the parsing complexity of lexicalized grammar formalisms can be systematically related to structural properties of the dependency graphs that these formalisms can induce. In this way, we have generalized Gaifman's [1965] equivalence result from context-free generative capacity and projective dependency structures on the one hand to mildly context-sensitive generative capacity and 'mildly' non-projective dependency structures on the other.

In this final chapter of the book, we summarize our main contributions, and identify some avenues for future research.

9.1 Main Contributions

In the first part of the book, we studied three classes of dependency structures: projective, block-restricted, and well-nested structures. Each of these classes were originally defined in terms of a structural constraint on dependency graphs. Bridging the gap between dependency representations and grammar formalisms, we complemented this graph-based perspective with an algebraic framework that encapsulates the structural constraints in the operations by which dependency graphs can be composed. An important technical result in this context was the encoding of dependency structures into terms over a signature of *order annotations* in such a way that the three classes that we considered could be characterized through certain subsets of this signature. With the one-to-one correspondence between dependency structures and terms, we were able to define the concept of a *dependency algebra* and show that these algebras are isomorphic to their corresponding term algebras. The relevance of this result is that composition operations on dependency structures can be simulated by corresponding operations on terms, which provide us with a well-studied and convenient data structure.

At the end of the first part of the book, we put our algebraic framework to use and classified several lexicalized grammar formalisms with respect to

M. Kuhlmann: Dependency Structures and Lexicalized Grammars, LNAI 6270, pp. 121–125, 2010.

the dependency structures that are induced by their derivations. Taking the algebraic approach, we formalized our particular notion of induction as the evaluation of a formalism's derivations trees in a dependency algebra. We showed the following results: The class of dependency structures that is induced by Context-Free Grammar is the class of projective dependency structures. Linear Context-Free Rewriting Systems (LCFRS) induce the class of block-restricted dependency structures; more specifically, the maximal block-degree of a dependency structure induced by an LCFRS is directly correlated to the measure of 'fan-out' that is used to sub-classify these systems. Adding well-nestedness to the block-degree restriction corresponds to the step from LCFRS to Coupled Context-Free Grammar (CCFG). As a special case, the class of well-nested dependency structures with a block-degree of at most 2 is exactly the class of dependency structures that is induced by Tree Adjoining Grammar (TAG). With these connections, we have effectively quantified the generative capacity of lexicalized grammar formalisms along an infinite hierarchy of ever more non-projective dependency structures. This measure is attractive as an alternative to string-generative capacity and tree-generative capacity because dependency representations are more informative than strings, but less formalism-specific than parse trees. In recent work, we have applied this measure also to other grammar formalisms, such as Combinatory Categorial Grammar [60] and Minimalist Grammar [4].

The algebraic perspective on dependency structures also led to efficient algorithms for deciding whether a given structure is projective or well-nested, or has a certain block-degree. We used these algorithms to evaluate the empirical relevance of the three structural constraints on data from three widely-used dependency treebanks. The outcome of these experiments was, that while the class of projective dependency structures is clearly insufficient for many practical applications, one only needs to go a small step beyond projectivity in order to cover virtually all of the data. In particular, the class of TAG-inducible dependency structures covers close to 99.5% of all the analyses in both the Danish Dependency Treebank [63] and two versions of the Prague Dependency Treebank [40, 41], the largest dependency corpus.

In the second part of the book, we developed the theory of regular sets of dependency structures, or *regular dependency languages*. Our approach to define the notion of 'regularity' for dependency structures was completely canonical: the regular dependency languages are the recognizable subsets of dependency algebras in the sense of Mezei and Wright [78]. By this definition, we obtained natural notions of automata and grammars for dependency structures on the basis of which we were able to study language-theoretic properties[1] We also proved a powerful pumping lemma for regular dependency languages, and used it to show that they are semilinear, a property also characteristic for mildly context-sensitive languages. As another application of

[1] In recent work, we have also characterized regular sets of dependency structures in terms of monadic second-order logic [67].

our pumping lemma, we showed that, under the constraint of regularity, there is a direct correspondence between block-degree and well-nestedness on the one hand, and string-generative capacity on the other. More specifically, the block-degree parameter induces an infinite hierarchy of string languages, and on almost every level of this hierarchy, the string languages that correspond to well-nested dependency structures form a strict subclass.

Finally, we investigated the parsing complexity of regular dependency languages. While the restriction to a fixed block-degree is sufficient to render the parsing problem polynomial in the length of the input sentence, we found that it does not suffice for practical applications: the parsing problem where the size of the grammar is taken into account is NP-complete for unrestricted regular dependency grammars. Interestingly, the corresponding problem for well-nested grammars is polynomial. Together with our treebank studies, this results provides further evidence that the well-nestedness constraint has relevance for practical applications.

9.2 Future Directions

There are several aspects of the work reported in this book, both theoretical and practical, that can be elaborated in future research.

9.2.1 Development of the Formalism

A major limitation in the practical applicability of regular dependency grammars is the fact that every set of dependency trees generated by such a grammar has bounded out-degree (Lemma 7.1.3). This implies that regular dependency languages cannot account for phenomena in which a given word accepts an arbitrary number of modifiers, such as nouns accept chains of adjectives. It would therefore be interesting to extend our concept of regularity to sets of dependency trees with an unbounded number of dependents. This would require us to find a different algebraic structure for dependency trees. One such structure is implicit in the parsing algorithms presented by Eisner [20] (for projective dependency trees) and Gómez-Rodríguez et al. [33] (for well-nested dependency trees). In these algorithms, the dependents of a node are not added all at once, but one-by-one in a piecemeal fashion.

9.2.2 Linguistic Relevance

The treebank evaluation that we reported in Chapters 3–5 was a purely quantitative one: we simply counted the structures that satisfied or did not satisfy a given structural constraint. While this was helpful in getting an estimate of the practical relevance of the structural constraints that we discussed in this book, it would be highly desirable to complement our results with a qualitative analysis of the treebank data. In particular, it would be very interesting to see whether there is any systematic connection between structural constraints and specific linguistic phenomena or typological properties.

An important question related to this concerns the practical relevance of the notion of induction that we have used in this book. The dependency tree induced by a derivation of Tree Adjoining Grammar (TAG), for example, may not be the 'right' dependency tree from a practical perspective: Several authors [7, 96, 98] have pointed out that for most TAG grammars, the directionality of some of the edges in a derivation tree need to be reversed in order to reflect the intended syntactic dependencies. Modelling this in our formal framework would require a more complex notion of induction as the one that we have used in this book.

9.2.3 Applications to Parsing

It appears promising to investigate the usefulness of our results for parsing. Recent work on data-driven dependency parsing has shown that parsing with unrestricted non-projective dependency graphs is intractable under all but the simplest probability models [74, 75]. On the other hand, projective dependency parsers combine favourably with more complicated models. It would be interesting to see whether the structural constraints that we have discussed in this book can be exploited to obtain efficient yet well-informed parsers even for certain classes of non-projective dependency graphs.

An attractive such class to look at is the class of well-nested dependency structures with a block-degree of at most 2. As we mentioned above, this class has close to perfect coverage on the data in the three treebanks that we evaluated it on. At the same time, we showed that this class corresponds to the dependency structures induced by TAG (Section 6.4). Gómez-Rodríguez et al. [33] show how to tap the literature on efficient parsing algorithms for lexicalized TAG to develop exhaustive parsing algorithms for the class of well-nested dependency structures—but the question whether the use of these more expressive representations can also lead to improvements in parsing accuracy is still unresolved. It would also be interesting to see how block-degree and well-nestedness can be captured not only by exhaustive parsers, but also in the framework of deterministic dependency parsing [87].

9.2.4 An Algebraic Perspective on Grammar Formalisms

The main question that we asked in this book,

> Which grammars induce which sets of dependency structures?

is only one instantiation of a more general approach to the comparative study of grammar formalisms. Formally, our notion of induction can be understood as the interpretation of grammatical derivations (represented as terms) in an algebra whose domain is a set of dependency structures. Choosing different algebras, we can give different interpretations to derivations; for example, the standard perspectives on CFG and TAG can be recovered by choosing suitable algebras on strings and trees, respectively. This suggests a dissection of a grammar into two parts: a ‘generative’ component that produces abstract descriptions of grammatical derivations, and an ‘interpretative’ component that assigns linguistic meaning to derivations. Such an analysis is the underlying idea of Pollard’s [1984] ‘generalized context-free grammars’ and the original conception of Linear Context-Free Rewriting Systems [118]. It gives us a way to compare grammar formalisms based the question whether they differ with respect to their generative components, their interpretative components, or both, which can lead to a better understanding of the generative capacity of a grammar formalism. It is interesting also because it may simplify the reuse of both formal results, algorithmic techniques, and linguistic resources, if these can be factorized in the same way. This was the case for our results about the string-generative capacity of regular dependency grammars (Lemmata 8.2.2 and 8.2.3), which we derived by combining a pumping lemma for regular tree languages (Lemma 7.2.1) with an observation about the combinatorics of strings (Lemma 8.2.1). Similarly, the parsing algorithm that we outlined in Section 8.3 can be reconstructed as the combination of a parsing algorithm for regular tree grammars and a bookkeeping mechanism for the string segments projected by subderivations. A thorough analysis of grammar formalisms along these lines makes an interesting enterprise for future research.

References

1. Abe, N.: Feasible learnability of formal grammars and the theory of natural language acquisition. In: 36th Annual Meeting of the Association for Computational Linguistics and 18th International Conference on Computational Linguistics (COLING-ACL), Montréal, Canada, pp. 1–6 (1998)
2. Becker, T., Rambow, O., Niv, M.: The derivational generative power of formal systems, or: Scrambling is beyond LCFRS. IRCS Report 92-38, University of Pennsylvania, Philadelphia, USA (1992)
3. Bodirsky, M., Kuhlmann, M., Möhl, M.: Well-nested drawings as models of syntactic structure. In: Tenth Conference on Formal Grammar and Ninth Meeting on Mathematics of Language, Edinburgh, UK, pp. 195–203 (2005)
4. Boston, M.F., Hale, J., Kuhlmann, M.: Dependency structures derived from Minimalist Grammars. In: Eleventh Meeting on Mathematics of Language, Pre-Proceedings, Bielefeld, Germany, pp. 11–20 (2009)
5. Boullier, P.: On TAG parsing. In: Traitement Automatique des Langues Naturelles (TALN), Cargèse, France, pp. 75–84 (1999)
6. Buchholz, S., Marsi, E.: CoNLL-X shared task on multilingual dependency parsing. In: Tenth Conference on Computational Natural Language Learning (CoNLL), New York, USA, pp. 149–164 (2006)
7. Candito, M.-H., Kahane, S.: Can the TAG derivation tree represent a semantic graph? An answer in the light of Meaning-Text Theory. In: Fourth Workshop on Tree-Adjoining Grammars and Related Formalisms (TAG+4), Philadelphia, USA, pp. 21–24 (1998)
8. Carme, J., Niehren, J., Tommasi, M.: Querying unranked trees with stepwise tree automata. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 105–118. Springer, Heidelberg (2004)
9. Chen-Main, J., Joshi, A.K.: Multi-component tree adjoining grammars, dependency graph models, and linguistic analyses. In: ACL 2007 Workshop on Deep Linguistic Processing, Prague, Czech Republic, pp. 1–8 (2007)
10. Chen-Main, J., Joshi, A.K.: Some observations on a 'graphical' model-theoretical approach and generative models. In: Model-Theoretic Syntax at 10, Dublin, Ireland, pp. 55–64 (2007)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)

12. Courcelle, B.: Basic notions of universal algebra for language theory and graph grammars. Theoretical Computer Science 163(1-2), 1–54 (1996)
13. Covington, M.A.: A fundamental algorithm for dependency parsing. In: 39th Annual ACM Southeast Conference, Athens, GA, USA, pp. 95–102 (2001)
14. Culotta, A., Sorensen, J.: Dependency tree kernels for relation extraction. In: 42nd Annual Meeting of the Association for Computational Linguistics (ACL), Barcelona, Spain, pp. 423–429 (2004)
15. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order, 2nd edn. Cambridge University Press, Cambridge (2001)
16. Denecke, K., Wismath, S.L.: Universal Algebra and Applications in Theoretical Computer Science. Chapman and Hall/CRC, Boca Raton (2001)
17. Diestel, R.: Graph Theory, 3rd edn. Graduate Texts in Mathematics, vol. 173. Springer, Heidelberg (2005)
18. Dikovsky, A., Modina, L.: Dependencies on the other side of the curtain. Traitement Automatique Des Langues 41(1), 67–96 (2000)
19. Duchier, D., Debusmann, R.: Topological dependency trees: A constraint-based account of linear precedence. In: 39th Annual Meeting of the Association for Computational Linguistics (ACL), Toulouse, France, pp. 180–187 (2001)
20. Eisner, J.: Three new probabilistic models for dependency parsing: An exploration. In: 16th International Conference on Computational Linguistics (COLING), Copenhagen, Denmark, pp. 340–345 (1996)
21. Eisner, J., Satta, G.: Efficient parsing for bilexical context-free grammars and Head Automaton Grammars. In: 37th Annual Meeting of the Association for Computational Linguistics (ACL), College Park, MD, USA, pp. 457–464 (1999)
22. Engelfriet, J.: Bottom-up and top-down tree transformations – a comparison. Theory of Computing Systems 9(2), 198–231 (1975)
23. Engelfriet, J., Maneth, S.: Output string languages of compositions of deterministic macro tree transducers. Journal of Computer and System Sciences 64(2), 350–395 (2002)
24. Engelfriet, J., Rozenberg, G., Slutzki, G.: Tree transducers, L systems, and two-way machines. Journal of Computer and System Sciences 20(2), 150–202 (1980)
25. Fischer, M.J.: Grammars with macro-like productions. In: Ninth Annual Symposium on Switching and Automata Theory, Schenectady, New York, USA, pp. 131–142 (1968)
26. Fujiyoshi, A., Kasai, T.: Spinal-formed context-free tree grammars. Theory of Computing Systems 33(1), 59–83 (2000)
27. Gaifman, H.: Dependency systems and phrase-structure systems. Information and Control 8(3), 304–337 (1965)
28. Gécseg, F., Steinby, M.: Tree languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, pp. 1–68. Springer, Heidelberg (1997)
29. Gerdes, K., Kahane, S.: Word order in German: A formal dependency grammar using a topological hierarchy. In: 39th Annual Meeting of the Association for Computational Linguistics (ACL), Toulouse, France, pp. 220–227 (2001)
30. Ginsburg, S., Spanier, E.H.: Semigroups, Presburger formulas, and languages. Pacific Journal of Mathematics 16(2), 285–296 (1966)
31. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. Journal of the Association for Computing Machinery 24(1), 68–95 (1977)

32. Goldstine, J.: A simplified proof of Parikh's theorem. Discrete Mathematics 19(3), 235–239 (1977)
33. Gómez-Rodríguez, C., Weir, D.J., Carroll, J.: Parsing mildly non-projective dependency structures. In: Twelfth Conference of the European Chapter of the Association for Computational Linguistics (EACL), Athens, Greece, pp. 291–299 (2009)
34. Gómez-Rodríguez, C., Kuhlmann, M., Satta, G.: Efficient parsing of well-nested linear context-free rewriting systems. In: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Los Angeles, USA, pp. 276–284 (2010)
35. Gramatovici, R., Plátek, M.: A natural infinite hierarchy by free-order dependency grammars. In: Information Technologies – Applications and Theory, Bystrá dolina, Slovak Republic, pp. 51–56 (2006)
36. Greibach, S.A.: A new normal-form theorem for context-free phrase structure grammars. Journal of the Association for Computing Machinery 12(1), 42–52 (1965)
37. Groenink, A.V.: Surface without Structure. Word Order and Tractability Issues in Natural Language Analysis. PhD thesis, Utrecht University, Utrecht, The Netherlands (1997)
38. Guan, Y.: Klammergrammatiken, Netzgrammatiken und Interpretationen von Netzen. PhD thesis, Saarland University, Saarbrücken, Germany (1992)
39. Haghighi, A.D., Ng, A.Y., Manning, C.D.: Robust textual inference via graph matching. In: Human Language Technology Conference (HLT) and Conference on Empirical Methods in Natural Language Processing (EMNLP), Vancouver, Canada, pp. 387–394 (2005)
40. Hajič, J., Hajičová, E., Pajas, P., Panevová, J., Sgall, P.: Prague Dependency Treebank. 1.0. Linguistic Data Consortium, 2001T10 (2001)
41. Hajič, J., Panevová, J., Hajičová, E., Sgall, P., Pajas, P., Štěpánek, J., Havelka, J., Mikulová, M.: Prague Dependency Treebank 2.0. Linguistic Data Consortium, 2006T01 (2006)
42. Hajičová, E., Havelka, J., Sgall, P., Veselá, K., Zeman, D.: Issues of projectivity in the Prague Dependency Treebank. Prague Bulletin of Mathematical Linguistics 81, 5–22 (2004)
43. Havelka, J.: Projectivity in totally ordered rooted trees. Prague Bulletin of Mathematical Linguistics 84, 13–30 (2005)
44. Havelka, J.: Beyond projectivity: Multilingual evaluation of constraints and measures on non-projective structures. In: 45th Annual Meeting of the Association for Computational Linguistics (ACL), Prague, Czech Republic, pp. 608–615 (2007)
45. Havelka, J.: Relationship between non-projective edges, their level types, and well-nestedness. In: Human Language Technologies: The Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL), Rochester, NY, USA, pp. 61–64 (2007)
46. Hays, D.G.: Grouping and dependency theory. In: National Symposium on Machine Translation, Englewood Cliffs, NY, USA, pp. 258–266 (1960)
47. Hays, D.G.: Dependency theory: A formalism and some observations. Language 40(4), 511–525 (1964)

48. Holan, T., Kuboň, V., Oliva, K., Plátek, M.: Two useful measures of word order complexity. In: Workshop on Processing of Dependency-Based Grammars, Montréal, Canada, pp. 21–29 (1998)
49. Hotz, G., Pitsch, G.: On parsing coupled-context-free languages. Theoretical Computer Science 161(1-2), 205–233 (1996)
50. Hudson, R.: Word Grammar. Basil Blackwell, Oxford (1984)
51. Hudson, R.: Language Networks. The New Word Grammar. Oxford University Press, Oxford (2007)
52. Huybregts, R.: The weak inadequacy of context-free phrase structure grammars. In: de Haan, G., Trommelen, M., Zonneveld, W. (eds.) Van Periferie Naar Kern, Foris, Dordrecht, The Netherlands, pp. 81–99 (1984)
53. Joshi, A.K.: Tree Adjoining Grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In: Natural Language Parsing, pp. 206–250. Cambridge University Press, Cambridge (1985)
54. Joshi, A.K., Schabes, Y.: Tree-Adjoining Grammars. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, pp. 69–123. Springer, Heidelberg (1997)
55. Joshi, A.K., Levy, L.S., Takahashi, M.: Tree Adjunct Grammars. Journal of Computer and System Sciences 10(2), 136–163 (1975)
56. Kahane, S., Nasr, A., Rambow, O.: Pseudo-projectivity: A polynomially parsable non-projective dependency grammar. In: 36th Annual Meeting of the Association for Computational Linguistics and 18th International Conference on Computational Linguistics (COLING-ACL), Montréal, Canada, pp. 646–652 (1998)
57. Kallmeyer, L.: Comparing lexicalized grammar formalisms in an empirically adequate way: The notion of generative attachment capacity. In: International Conference on Linguistic Evidence, Tübingen, Germany, pp. 154–156 (2006)
58. Kanazawa, M.: The pumping lemma for well-nested multiple context-free languages. In: DLT 2009. LNCS, vol. 5583, pp. 312–325. Springer, Heidelberg (2009)
59. Kepser, S., Mönnich, U.: Closure properties of linear context-free tree languages with an application to optimality theory. Theoretical Computer Science 354(1), 82–97 (2006)
60. Koller, A., Kuhlmann, M.: Dependency trees and the strong generative capacity of CCG. In: Twelfth Conference of the European Chapter of the Association for Computational Linguistics (EACL), Athens, Greece, pp. 460–468 (2009)
61. Koller, A., Striegnitz, K.: Generation as dependency parsing. In: 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, USA, pp. 17–24 (2002)
62. Kracht, M.: The Mathematics of Language. Studies in Generative Grammar, vol. 63. Mouton de Gruyter, Berlin (2003)
63. Kromann, M.T.: The Danish Dependency Treebank and the underlying linguistic theory. In: Second Workshop on Treebanks and Linguistic Theories (TLT), Växjö, Sweden, pp. 217–220 (2003)
64. Kromann, M.T.: Discontinuous Grammar. A Model of Human Parsing and Language Acquisition. Dr. Ling. Merc. Dissertation, Copenhagen Business School, Copenhagen, Denmark (2005)

65. Kuhlmann, M., Möhl, M.: Extended cross-serial dependencies in Tree Adjoining Grammars. In: Eighth International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+8), Sydney, Australia, pp. 121–126 (2006)
66. Kuhlmann, M., Möhl, M.: Mildly context-sensitive dependency languages. In: 45th Annual Meeting of the Association for Computational Linguistics (ACL), Prague, Czech Republic, pp. 160–167 (2007)
67. Kuhlmann, M., Niehren, J.: Logics and automata for totally ordered trees. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 217–231. Springer, Heidelberg (2008)
68. Kuhlmann, M., Satta, G.: Treebank grammar techniques for non-projective dependency parsing. In: Twelfth Conference of the European Chapter of the Association for Computational Linguistics (EACL), Athens, Greece, pp. 478–486 (2009)
69. Kunze, J.: Die Auslassbarkeit von Satzteilen bei koordinativen Verbindungen im Deutschen. Akademie-Verlag, Berlin (1972)
70. Lee, L.: Fast context-free grammar parsing requires fast Boolean matrix multiplication. Journal of the Association for Computing Machinery 49(1), 1–15 (2002)
71. Maier, W., Lichte, T.: Characterizing discontinuity in constituent treebanks. In: 14th Conference on Formal Grammar, Bordeaux, France (2009)
72. Marcus, S.: Algebraic Linguistics: Analytical Models. Mathematics in Science and Engineering, vol. 29. Academic Press, New York (1967)
73. McAllester, D.: On the complexity analysis of static analyses. Journal of the Association for Computing Machinery 49(4), 512–537 (2002)
74. McDonald, R., Pereira, F.: Online learning of approximate dependency parsing algorithms. In: Eleventh Conference of the European Chapter of the Association for Computational Linguistics (EACL), Trento, Italy, pp. 81–88 (2006)
75. McDonald, R., Satta, G.: On the complexity of non-projective data-driven dependency parsing. In: Tenth International Conference on Parsing Technologies (IWPT), Prague, Czech Republic, pp. 121–132 (2007)
76. McDonald, R., Pereira, F., Ribarov, K., Hajič, J.: Non-projective dependency parsing using spanning tree algorithms. In: Human Language Technology Conference (HLT) and Conference on Empirical Methods in Natural Language Processing (EMNLP), Vancouver, Canada, pp. 523–530 (2005)
77. Mel'čuk, I.: Dependency Syntax: Theory and Practice. State University of New York Press, Albany (1988)
78. Mezei, J.E., Wright, J.B.: Algebraic automata and context-free sets. Information and Control 11(1-2), 3–29 (1967)
79. Michaelis, J., Kracht, M.: Semilinearity as a syntactic invariant. In: Retoré, C. (ed.) LACL 1996. LNCS (LNAI), vol. 1328, pp. 329–345. Springer, Heidelberg (1997)
80. Möhl, M.: Drawings as models of syntactic structure: Theory and algorithms. Master's thesis, Saarland University, Saarbrücken, Germany (2006)
81. Mönnich, U.: Adjunction as substitution. an algebraic formulation of regular, context-free, and tree-adjoining languages. In: Third Conference on Formal Grammar, Aix-en-Provence, France, pp. 169–178 (1997)
82. Nasr, A.: A formalism and a parser for lexicalised dependency grammars. In: Fourth International Workshop on Parsing Technologies (IWPT), Prague, Czech Republic, pp. 186–195 (1995)

83. Neuhaus, P., Bröker, N.: The complexity of recognition of linguistically adequate dependency grammars. In: 35th Annual Meeting of the Association for Computational Linguistics (ACL), Madrid, Spain, pp. 337–343 (1997)
84. Niehren, J., Podelski, A.: Feature automata and recognizable sets of feature trees. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) TAPSOFT 1993. LNCS, vol. 668, pp. 356–375. Springer, Heidelberg (1993)
85. Nivre, J.: Constraints on non-projective dependency parsing. In: Eleventh Conference of the European Chapter of the Association for Computational Linguistics (EACL), Trento, Italy, pp. 73–80 (2006)
86. Nivre, J.: Inductive Dependency Parsing. Text, Speech and Language Technology, vol. 34. Springer, Heidelberg (2006)
87. Nivre, J.: Algorithms for deterministic incremental dependency parsing. Computational Linguistics 34(4), 513–553 (2008)
88. Nivre, J., Nilsson, J.: Pseudo-projective dependency parsing. In: 43rd Annual Meeting of the Association for Computational Linguistics (ACL), Ann Arbor, USA, pp. 99–106 (2005)
89. Nivre, J., Hall, J., Kübler, S., McDonald, R., Nilsson, J., Riedel, S., Yuret, D.: The CoNLL 2007 shared task on dependency parsing. In: Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), Prague, Czech Republic, pp. 915–932 (2007)
90. Ogden, W.: A helpful result for proving inherent ambiguity. Mathematical Systems Theory 2(3), 191–194 (1968)
91. Parikh, R.J.: On context-free languages. Journal of the Association for Computing Machinery 13(4), 570–581 (1966)
92. Pericliev, V., Ilarionov, I.: Testing the projectivity hypothesis. In: Sixth International Conference on Computational Linguistics (COLING), Bonn, Germany, pp. 56–58 (1986)
93. Pollard, C.J.: Generalized Phrase Structure Grammars, Head Grammars, and Natural Language. PhD thesis, Stanford University (1984)
94. Quirk, C., Menezes, A., Cherry, C.: Dependency treelet translation: Syntactically informed phrasal SMT. In: 43rd Annual Meeting of the Association for Computational Linguistics (ACL), Ann Arbor, USA, pp. 271–279 (2005)
95. Rambow, O.: Formal and Computational Aspects of Natural Language Syntax. PhD thesis, University of Pennsylvania, Philadelphia, USA (1994)
96. Rambow, O., Joshi, A.K.: A formal look at dependency grammars and phrase-structure grammars, with special consideration of word-order phenomena. In: Wanner, L. (ed.) Recent Trends in Meaning-Text Theory. Studies in Language, Companion Series, vol. 39, pp. 167–190. John Benjamins, Amsterdam (1997)
97. Rambow, O., Satta, G.: A two-dimensional hierarchy for parallel rewriting systems. Technical Report IRCS-94-02, University of Pennsylvania, Philadelphia, USA (2004)
98. Rambow, O., Vijay-Shanker, K., Weir, D.J.: D-Tree grammars. In: 33rd Annual Meeting of the Association for Computational Linguistics (ACL), Cambridge, MA, USA, pp. 151–158 (1995)
99. Raoult, J.-C.: Rational tree relations. Bulletin of the Belgian Mathematical Society 4(1), 149–176 (1997)
100. Robinson, J.J.: Dependency structures and transformational rules. Language 46(2), 259–285 (1970)

101. Rosenkrantz, D.J.: Matrix equations and normal forms for context-free grammars. Journal of the Association for Computing Machinery 14(3), 501–507 (1967)
102. Satta, G.: Recognition of Linear Context-Free Rewriting Systems. In: 30th Annual Meeting of the Association for Computational Linguistics (ACL), Newark, DE, USA, pp. 89–95 (1992)
103. Schabes, Y.: Mathematical and Computational Aspects of Lexicalized Grammars. PhD thesis, University of Pennsylvania, Philadelphia, USA (1990)
104. Schabes, Y., Abeillé, A., Joshi, A.K.: Parsing strategies with 'lexicalized' grammars: Application to Tree Adjoining Grammars. In: Twelfth International Conference on Computational Linguistics (COLING), Budapest, Hungary, pp. 578–583 (1988)
105. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On Multiple Context-Free Grammars. Theoretical Computer Science 88(2), 191–229 (1991)
106. Sgall, P., Hajičová, E., Panevová, J.: The Meaning of the Sentence in its Semantic and Pragmatic Aspects. Springer, Heidelberg (1986)
107. Shieber, S.M.: Evidence against the context-freeness of natural language. Linguistics and Philosophy 8(3), 333–343 (1985)
108. Shieber, S.M., Schabes, Y., Pereira, F.: Principles and implementation of deductive parsing. Journal of Logic Programming 24(1-2), 3–36 (1995)
109. Simion, R.: Noncrossing partitions. Discrete Mathematics 217(1-3), 367–409 (2000)
110. Sleator, D., Temperley, D.: Parsing English with a Link Grammar. Technical report, Carnegie Mellon University, Pittsburgh, USA (1991)
111. Sleator, D., Temperley, D.: Parsing English with a Link Grammar. In: Third International Workshop on Parsing Technologies (IWPT), Tilburg, The Netherlands, Durbuy, Belgium, pp. 277–292 (1993)
112. Sloane, N.J.A.: The on-line encyclopedia of integer sequences (2010), `http://www.research.att.com/~njas/sequences/`
113. Tesnière, L.: Éléments de syntaxe structurale. Klinksieck, Paris (1959)
114. Tesnière, L.: Grundzüge der strukturalen Syntax. Klett-Cotta, Stuttgart (1980)
115. Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. Mathematical Systems Theory 2(1), 57–81 (1968)
116. van Vugt, N.: Generalized context-free grammars. Master's thesis, Universiteit Leiden, The Netherlands (1996)
117. Veselá, K., Havelka, J., Hajičová, E.: Condition of projectivity in the underlying dependency structures. In: 20th International Conference on Computational Linguistics (COLING), Geneva, Switzerland, pp. 289–295 (2004)
118. Vijay-Shanker, K., Weir, D.J., Joshi, A.K.: Characterizing structural descriptions produced by various grammatical formalisms. In: 25th Annual Meeting of the Association for Computational Linguistics (ACL), Stanford, CA, USA, pp. 104–111 (1987)
119. Weir, D.J.: Characterizing Mildly Context-Sensitive Grammar Formalisms. PhD thesis, University of Pennsylvania, Philadelphia, USA (1988)
120. Weir, D.J.: Linear context-free rewriting systems and deterministic tree-walking transducers. In: 30th Annual Meeting of the Association for Computational Linguistics (ACL), Newark, DE, USA, pp. 136–143 (1992)

121. Yli-Jyrä, A.: Multiplanarity – a model for dependency structures in treebanks. In: Second Workshop on Treebanks and Linguistic Theories (TLT), Växjö, Sweden, pp. 189–200 (2003)

122. Zeman, D.: Parsing with a Statistical Dependency Model. PhD thesis, Charles University, Prague, Czech Republic (2004)

Index