Chapter 1
Dependency Grammar: Classification and Exploration

Ralph Debusmann and Marco Kuhlmann

Abstract Grammar formalisms built on the notion of word-to-word dependencies make attractive alternatives to formalisms built on phrase structure representations. However, little is known about the formal properties of dependency grammars, and few such grammars have been implemented. We present results from two strands of research that address these issues. The aims of this research were to *classify* dependency grammars in terms of their generative capacity and parsing complexity, and to systematically *explore* their expressive power in the context of a practical system for grammar development and parsing.

1.1 Introduction

Syntactic representations based on word-to-word dependencies have a long tradition in descriptive linguistics [29]. In recent years, they have also become increasingly used in computational tasks, such as information extraction [5], machine translation [43], and parsing [42]. Among the purported advantages of dependency over phrase structure representations are conciseness, intuitive appeal, and closeness to semantic representations such as predicate-argument structures. On the more practical side, dependency representations are attractive due to the increasing availability of large corpora of dependency analyses, such as the Prague Dependency Treebank [19].

The recent interest in dependency representations has revealed several gaps in the research on grammar formalisms based on these representations: First, while several linguistic theories of dependency grammars exist (examples are Functional Generative Description [48], Meaning-Text Theory [36], and Word Grammar [24]), there are few results on their formal properties—in particular, it is not clear how they can be related to the more well-known phrase structure-based formalisms. Second, few dependency grammars have been implemented in practical systems, and no tools for the development and exploration of new grammars are available.

Programming Systems Lab, Saarland University, Saarbrücken, Germany

In this chapter, we present results from two strands of research on dependency grammar that addresses the above issues. The aims of this research were to classify dependency grammars in terms of their generative capacity and parsing complexity, and to systematically explore their expressive power in the context of a practical system for grammar development and parsing. Our classificatory results provide fundamental insights into the relation between dependency grammars and phrase structure grammars. Our exploratory work shows how dependency-based representations can be used to model the complex interactions between different dimensions of linguistic description, such as word-order, quantifier scope, and information structure.

Structure of the chapter. The remainder of this chapter is structured as follows. In Sect. 1.2, we introduce dependency structures as the objects of description in dependency grammar, and identify three classes of such structures that are particularly relevant for practical applications. We then show how dependency structures can be related to phrase structure-based formalisms via the concept of lexicalization (Sect. 1.3). Section 1.4 introduces Extensible Dependency Grammar (XDG), a metagrammatical framework designed to facilitate the development of novel dependency grammars. In Sect. 1.5, we apply XDG to obtain an elegant model of complex word order phenomena, in Sect. 1.6 develop a relational syntax-semantics interface, and in Sect. 1.7 present an XDG model of *regular dependency grammars*. We apply the ideas behind this modeling in Sect. 1.8, where we introduce the grammar development environment for XDG and investigate its practical utility with an experiment on large-scale parsing. Section 1.9 concludes the chapter.

1.2 Dependency Structures

The basic assumptions behind the notion of dependency are summarized in the following sentences from the seminal work of Tesnière [51]:

> 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*). (ch. 1, §§ 2–4; ch. 2, §§ 1–2)

We can represent the dependency relations among the words of a sentence as a graph. More specifically, the *dependency structure* for a sentence $w = w_1 \cdots w_n$ is the directed graph on the set of positions of w that contains an edge $i \to j$ if and only if the word w_j depends on the word 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 Fig. 1.1. In this graph, the edge between the word *likes* and the word *Dan* encodes the syntactic information that *Dan* is the subject of *likes*. 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 [21], we use dotted lines to indicate the left-to-right ordering of the words in the sentence.

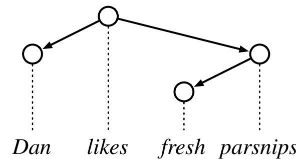
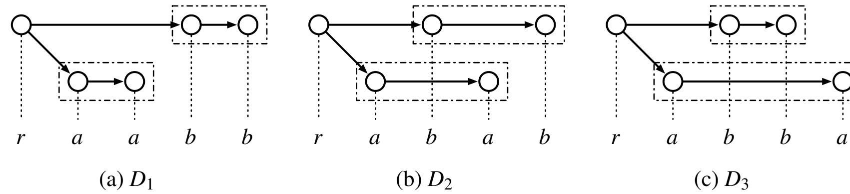


Fig. 1.1: A dependency structure

With the concept of a dependency structure at hand, we can express linguistic universals in terms of *structural constraints* on graphs. The most widely used such constraint is to require the dependency structure to form a tree. This requirement models the stipulations that no word should depend on itself, not even transitively, that each word should have at most one governor, and that a dependency analysis should cover all the words in the sentence. The dependency analysis shown in Fig. 1.1 satisfies the treeness constraint.

Another well-known constraint on dependency structures is *projectivity* [34]. In contrast to treeness, which imposes restrictions on dependency as such, projectivity concerns the relation between dependency and the left-to-right order of the words in the sentence. Specifically, it requires each dependency subtree to cover a contiguous region of the sentence. As an example, consider the dependency structure in Fig. 1.2a. Projectivity is interesting because the close relation between dependency and word order that it enforces can be exploited in parsing algorithms [17]. 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 extraction, topicalization and extraposition; they are particularly frequent in the analysis of languages with flexible word order, such as Czech [22, 52]. Unfortunately, without any further restrictions, non-projective dependency parsing is intractable [40, 35].

In search of a balance between the benefit of more expressivity 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 structures are computationally well-behaved [56, 41, 20]. Such constraints identify classes of what we may call *mildly non-projective dependency structures*. The *block-degree restriction* [22] relaxes projectivity such that dependency subtrees can be distributed over more than one interval. For example, in Fig. 1.2b, each of the marked subtrees spans two intervals. The third structural constraint that we have investigated is original to our research: *well-nestedness* [4] is the restriction that pairs of disjoint dependency subtrees must not cross, which means



(a) D_1 (b) D_2 (c) D_3

Fig. 1.2: Three dependency structures

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 Fig. 1.2c is well-nested, while the structure depicted in Fig. 1.2b is not.

To investigate the practical relevance of the three structural constraints, we did an empirical evaluation on two versions of the Prague Dependency Treebank [33] (Table 1.1). This evaluation shows that while projectivity is too strong a constraint on dependency structures (it excludes almost 23% of the analyses in both versions of the treebank), already a small step beyond projectivity covers virtually all of the data. In particular, even the rather restricted class of well-nested dependency structures with block-degree at most 2 has a coverage of almost 99.5%.

	PDT 1.0				PDT 2.0			
block-degree	unrestricted		well-nested		unrestricted		well-nested	
1 (projective)	56 168	76.85%	56 168	76.85%	52 805	77.02%	52 805	77.02%
2	16 608	22.72%	16 539	22.63%	15 467	22.56%	15 393	22.45%
3	307	0.42%	300	0.41%	288	0.42%	282	0.41%
4	4	0.01%	2	< 0.01%	2	< 0.01%	–	–
5	1	< 0.01%	1	< 0.01%	1	< 0.01%	1	< 0.01%
TOTAL	73 088	100.00%	73 010	99.89%	68 562	100.00%	68 481	99.88%

Table 1.1: Structural properties of dependency structures in the Prague Dependency Treebank

1.3 Dependency Structures and Lexicalized Grammars

One of the fundamental questions that we can ask about a grammar formalism is, whether it adequately models natural language. We can answer this question by studying the *generative capacity* of the formalism: when we interpret grammars as generators of sets of linguistic structures (such as strings, parse trees, or predicate-argument structures), then we can call a grammar adequate, if it generates exactly those structures that we consider relevant for the description of natural language.

Grammars may be adequate with respect to one type of expression, but inadequate with respect to another. Here we are interested in the generative capacity of grammars when we interpret them as generators for sets of dependency structures:

> Which grammars generate which sets of dependency structures?

An answer to this question is interesting for at least two reasons. First, dependency structures make an attractive measure of the generative capacity of a grammar: they are more informative than strings, but less formalism-specific and arguably closer to a semantic representation than parse trees. Second, an answer to the question allows us to tap the rich resource of formal results about generative grammar formalisms and to transfer them to the work on dependency grammar. Specifically, it enables us to import the expertise in developing parsing algorithms for lexicalized grammar formalisms. This can help us identify the polynomial fragments of non-projective dependency parsing.

1.3.1 Lexicalized Grammars Induce Dependency Structures

In order to relate grammar formalisms and dependency representations, we first need to specify in what sense we can consider a grammar as a generator of dependency structures. To focus our discussion, let us consider the well-known case of context-free grammars (CFGs). As our running example, Fig. 1.3 shows a small CFG together with a parse tree for a simple English sentence.

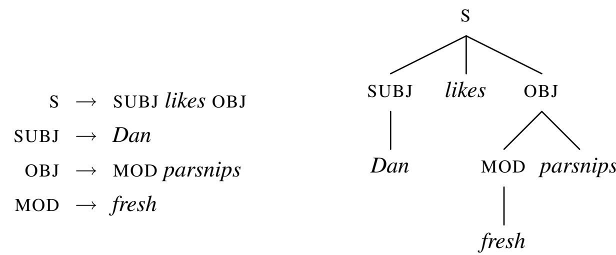


Fig. 1.3: A context-free grammar and a parse tree generated by this grammar

An interesting property of our sample grammar is that it is *lexicalized*: every rule of the grammar contains exactly one terminal symbol. Lexicalized grammars play a significant role in contemporary linguistic theories and practical applications. Crucially for us, every such grammar can be understood as a generator for sets of dependency structures, in the following sense. Consider a derivation of a terminal string by means of a context-free grammar. A *derivation tree* for this derivation 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. The

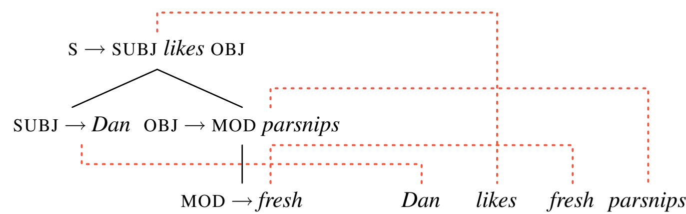


Fig. 1.4: Lexicalized derivations induce dependency structures

left half of Fig. 1.4 shows the derivation tree for the parse tree from our example. 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: each occurrence of a production participating in the derivation contributes exactly one terminal symbol to this string. If we order the nodes of the derivation tree according to the string positions of their corresponding terminal symbols, we get a dependency tree. For our example, this procedure results in the tree depicted in Fig. 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 famous counterexample is provided by the verb-argument dependencies in German and Dutch subordinate clauses: context-free grammar can only characterize the 'nested' dependencies of German, but not the 'cross-serial' assignments of Dutch. This observation goes along with arguments [25, 49] that certain constructions in Swiss German require grammar formalisms that adequately model these constructions to generate the so-called copy language, which is beyond even the string-generative capacity of CFGs. If we accept this analysis, then we must conclude that context-free grammars are not adequate for the description of natural language, and that we should look out for more powerful formalisms. This conclusion is widely accepted today. Unfortunately, the first class in Chomsky's hierarchy of formal languages that *does* contain the copy language, the class of *context-sensitive languages*, also contains many languages that are considered to be beyond human capacity. Also, while CFGs can be parsed in polynomial time, parsing of context-sensitive grammars is PSPACE-complete. 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 the best-known among these is Tree Adjoining Grammar (TAG) [27]. The class of string languages generated by TAGs contains the copy language, but unlike context-sensitive grammars, TAGs can be parsed in polynomial time. More important to us than their increased string-generative capacity however is their stronger power with respect to dependency representations: derivations in (lexicalized) TAGs can induce the 'cross-serial' dependencies of Dutch [26]. The principal goal of our classificatory work is to make the relations between grammars and the dependency structures that they can induce precise.

In spite of the apparent connection between the generative capacity of a grammar formalism and the structural properties of the dependency structures that this formalism can induce, there have been only few results that link the two research areas. A fundamental reason for the lack of such bridging results is that, while structural constraints on dependency structures 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 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 a certain set of operations. One of the central technical questions that we need to answer in order to connect grammars and structures is, how classes of dependency structures can be given an algebraic structure.

1.3.2 The Algebraic View on Dependency Structures

In order to link structural constraints to generative grammar formalisms, we need to view dependency structures as the outcomes of compositional processes. Under this view, structural constraints do not only apply to fully specified dependency trees, but already to the composition operations by which these trees are constructed. We formalized the compositional view in two steps. In the first step, we showed 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 that we have discussed above stand in one-to-one correspondence with terms over specific subsets of this signature [31]. In the second step, we defined the concept of a *dependency algebra*. In these algebras, order annotations are interpreted as composition operations on dependency structures [30, 32]. We have proved 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, Fig. 1.5 shows the terms that correspond to the dependency structures in Fig. 1.2. Each order annotation in these terms encodes node-specific information about the linear order on the nodes. As an example, the constructor $\langle 0,1 \rangle$ in Fig. 1.5b represents the information that the marked subtrees in Fig. 1.2b each consist 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 Fig. 1.5b.

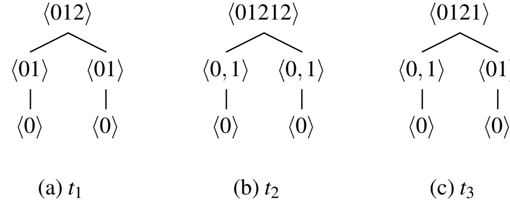


Fig. 1.5: Terms for the dependency structures in Fig. 1.2

Our algebraic framework enables us to classify the dependency structures that are induced by various lexicalized grammar formalisms. In particular, we can extend Gaifman's result [18] 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 [53, 54], the classes of well-nested block-restricted structures correspond to Coupled Context-Free Grammar [23]. 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 [27, 4]. This result is particularly interesting in the context of our treebank evaluation.

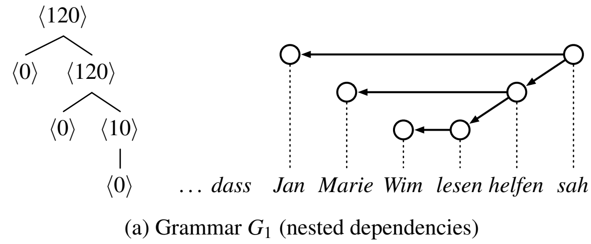
1.3.3 Regular Dependency Grammars

We can now lift our results from individual dependency structures to sets of such structures. The key to this transfer is the concept of *regular sets of dependency structures* [31], which we define as the recognizable subsets of dependency algebras in the sense of Mezei and Wright [37]. Based on the isomorphism between dependency algebras and term algebras, we obtain a natural grammar formalism for dependency structures from the concept of a *regular term grammar*.

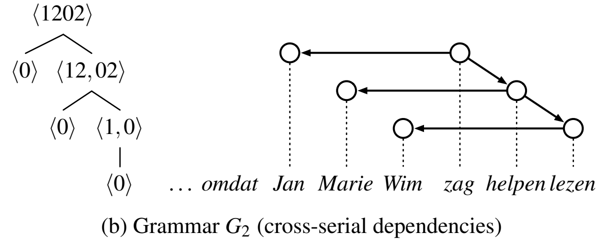
Definition 1. A *regular dependency grammar* is a construct $G = (N, \Sigma, S, P)$, where N is a ranked alphabet of non-terminal symbols, Σ is a finite set of order annotations, $S \in N_1$ is a distinguished start symbol, and P is a finite set of productions of the form $A \to t$, where $A \in N_k$ is a non-terminal symbol, and $t \in T_{\Sigma,k}$ is a well-formed term over Σ of sort k, for some $k \in \mathbb{N}$.

To illustrate the definition, we give two examples of regular dependency grammars. The sets of dependency structures generated by these grammars mimic the verb-argument relations found in German and Dutch subordinate clauses, respectively: grammar G_1 generates structures with nested dependencies, grammar G_2 generates structures with crossing dependencies. We only give the two sets of productions.

$$S \to \langle 120\rangle(N,V) \qquad V \to \langle 120\rangle(N,V) \qquad V \to \langle 10\rangle(N) \qquad N \to \langle 0\rangle \qquad (G_1)$$
$$S \to \langle 1202\rangle(N,V) \qquad V \to \langle 12,02\rangle(N,V) \qquad V \to \langle 1,0\rangle(N) \qquad N \to \langle 0\rangle \qquad (G_2)$$



(a) Grammar G_1 (nested dependencies)



(b) Grammar G_2 (cross-serial dependencies)

Fig. 1.6: Terms and structures generated by two regular dependency grammars

Figure 1.6 shows terms generated by these grammars, and the corresponding dependency structures.

The sets of dependency structures generated by regular dependency grammars have all the characteristic properties of mildly context-sensitive languages. Furthermore, it turns out that the structural constraints that we have discussed above have direct implications for their string-generative capacity and parsing complexity. First, the block-degree measure gives rise to an infinite hierarchy of ever more powerful string languages; adding the well-nestedness constraint leads to a proper decrease of string-generative power on nearly all levels of this hierarchy [32]. Certain string languages enforce structural properties in the dependency languages that project them: For every natural number k, the language

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

requires every regular set of dependency structures that projects it to contain structures with a block-degree of at most k. Similarly, the language

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

requires every regular set of dependency structures with block-degree at most k that projects it to contain structures that are not well-nested. Second, 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 still NP-complete. Interestingly, for well-nested dependency languages, parsing is polynomial even with the size of the grammar taken into account [30].

1.4 Extensible Dependency Grammar

For the *exploration* of dependency grammars, we have developed a new metagrammatical framework called Extensible Dependency Grammar (XDG) [11, 8]. The main innovation of XDG is *multi-dimensionality*: an XDG analysis consists of a tuple of dependency graphs all sharing the same set of nodes, called *dependency multigraph*. The components of the multigraph are called *dimensions*. The multidimensional metaphor was crucial for our formulations of a new, elegant model of complex word order phenomena in German, and a new, relational model of the syntax-semantics interface.

1.4.1 Dependency Multigraphs

To give an intuition, let us start with an example multigraph depicted in Fig. 1.7. The multigraph has three dimensions called DEP (for *dependency tree*), QS (for *quantifier scope analysis*) and DEP/QS (DEP/QS *syntax-semantics interface*). It is not necessary to fully understand what we intend to model with these dimensions; they just serve as an illustrative example, and are elucidated in more detail in Sect. 1.6 below.

In an XDG multigraph, each dimension is a dependency graph made up of a set of nodes associated with indices, words and node attributes. The indices and words are shared across all dimensions. For instance, the second node on the DEP dimension is associated with the index 2, the word *loves*, and the node attributes *in*, *out* and *order*. On the DEP/QS dimension, the node has the same index and word and the node attribute *dom*. Node attributes always denote sets of tuples over finite domains of atoms; their typical use is to model finite relations like functions and orders. The nodes are connected by labeled edges. On the QS dimension for example, there is an edge from node 3 to node 1 labeled sc, and another one from node 1 to node 2, also labeled sc.

In the example, the DEP dimension states that *everybody* is the subject of *loves*, and *somebody* the object. The *in* and *out* attributes represent the licensed incoming and outgoing edges. For example, node 2 must not have any incoming edges, and it must have one outgoing edge labeled subj and one labeled obj. The *order* attribute represents a total order among the head (↑) and its dependents: the subj dependents must precede the head, and head must precede the obj dependents.

The QS dimension is an analysis of the scopal relationships of the quantifiers in the sentence. It models the reading where *somebody* takes scope over *everybody*, which in turn takes scope over *loves*. The DEP/QS analysis represents the syntax-semantics interface between DEP and QS. The attribute *dom* is a set of those dependents on the DEP dimension that must dominate the head on the QS dimension. For example, the subj and obj dependents of node 2 on DEP must dominate 2 on QS.

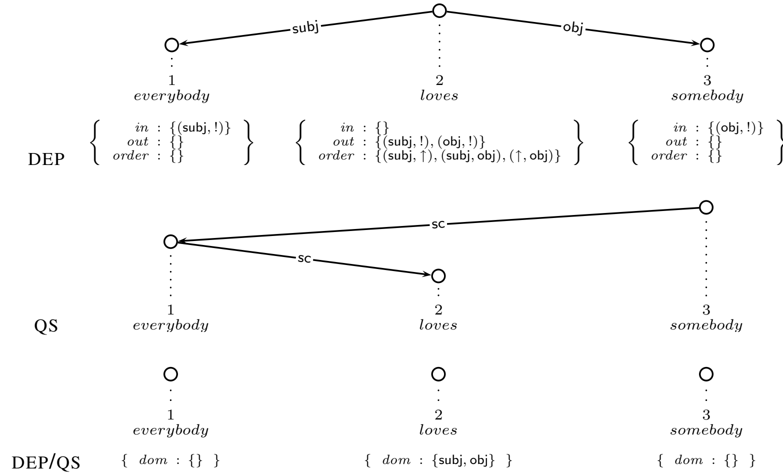


Fig. 1.7: Dependency multigraph for *Everybody loves somebody*

1.4.2 Grammars

XDG is a *model-theoretic* framework: grammars first delineate the set of all *candidate structures*, and second, all structures which are not well-formed according to a set of *constraints* are eliminated. The remaining structures are the *models* of the grammar. This contrasts with approaches such as the regular dependency grammars of Sect. 1.3.3, where the models are *generated* using a set of productions.

An XDG grammar $G = (MT, lex, P)$ has three components: a *multigraph type* MT, a *lexicon* lex, and a set of *principles* P. The multigraph type specifies the dimensions, words, edge labels and node attributes, and thus delineates the set of candidate structures of the grammar. The lexicon is a function from the words of the grammar to sets of lexical entries, which determine the node attributes of the nodes with that word. The principles are a set of formulas in first-order logic constituting the constraints of the grammar. Principles can talk about precedence, edges, dominances (transitive closure[1] of the edge relation), the words associated to the nodes, and the node attributes. Here is an example principle forbidding cycles on dimension DEP. It states that no node may dominate itself:

$$\forall v : \neg(v \rightarrow^{+}_{\text{DEP}} v) \tag{1.1}$$

The second example principle stipulates a constraint for all edges from v to v' labeled l on dimension DEP: if l is in the set denoted by the lexical attribute *dom* of v on DEP/QS, then v' must dominate v on QS:

[1] Transitive closures cannot be expressed in first-order logic. As in practice, the only transitive closure that we need is the transitive closure of the edge relation, we have decided to encode it in the multigraph model and thus stay in first-order logic [10].

$$\forall v : \forall v' : \forall l : \; v \xrightarrow{l}_{\mathrm{DEP}} v' \wedge l \in dom_{\mathrm{DEP/QS}}(v) \;\Rightarrow\; v' \rightarrow^{+}_{\mathrm{QS}} v \tag{1.2}$$

Observe that the principle is indeed satisfied in Fig. 1.7: the attribute *dom* for node 2 on DEP/QS includes subj and obj, and both the subj and the obj dependents of node 2 on DEP dominate node 2 on QS.

A multigraph is a *model* of a grammar $G = (MT, lex, P)$ iff it is one of the candidate structures delineated by MT, it selects precisely one lexical entry from *lex* for each node, and it satisfies all principles in P.

The *string language* $L(G)$ of a grammar G is the set of *yields* of its models. The *recognition problem* is the question given a grammar G and a string s, is s in $L(G)$. We have investigated the complexity of three kinds of recognition problems [9]: The *universal* recognition problem where both G and s are variable is PSPACE-complete, the *fixed* recognition problem where G is fixed and s is variable is NP-complete, and the *instance* recognition problem where the principles are fixed, and the lexicon and s are variable is also NP-complete. XDG parsing is NP-complete as well.

XDG is at least as expressive as CFG [8]. We have proven that the string languages of XDG grammars are closed under union and intersection [10]. In Sect. 1.7, we give a constructive proof that XDG is at least as expressive as the class of regular dependency grammars introduced in Sect. 1.3.3, which entails through an encoding of LCFRS in regular dependency grammars, that XDG is at least as expressive as LCFRS. As XDG is able to model scrambling (see Sect. 1.5.2), which LCFRS is not [3], it is indeed *more* expressive than LCFRS.

1.5 Modeling Complex Word Order Phenomena

The first application for the multi-dimensionality of XDG in CHORUS is the design of a new, elegant model of complex word order phenomena such as *scrambling*.

1.5.1 Scrambling

In German, the word order in subordinate sentences is such that all verbs are positioned at the right end in the so-called *verb cluster*, and are preceded by all the non-verbal dependents in the so-called *Mittelfeld*. Whereas the mutual order of the verbs is fixed, that of the non-verbal dependents in the Mittelfeld is totally free.[2] This leads to the phenomenon of scrambling. We show an example in Fig. 1.8, where the subscripts indicate the dependencies between the verbs and their arguments.

In the dependency analysis in Fig. 1.9 (top), we can see that scrambling gives rise to *non-projectivity*. In fact, scrambling even gives rise to an unbounded block-

[2] These are of course simplifications: the order of the verbs can be subject to alternations such as *Oberfeldumstellung*, and although all linearizations of the non-verbal dependents are grammatical, some of them are clearly marked.

Mittelfeld				verb cluster		
(dass)	*Nilpferde*$_3$	*Maria*$_1$	*Hans*$_2$	*füttern*$_3$	*helfen*$_2$	*soll*$_1$
(that)	*hippos*$_3$	*Maria*$_1$	*Hans*$_2$	*feed*$_3$	*help*$_2$	*should*$_1$
(that) Maria should help Hans feed hippos						

Fig. 1.8: Example for scrambling

degree (see Sect. 1.2), which means that it can neither be modeled by LCFRS, nor by regular dependency grammars.

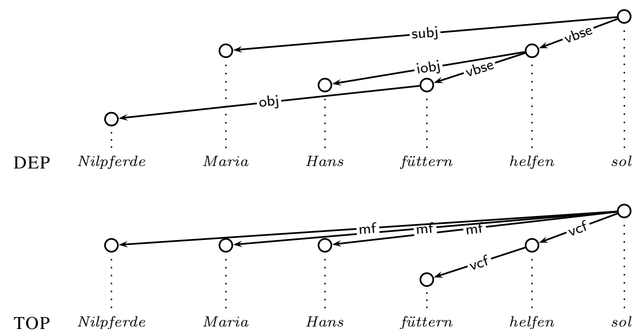


Fig. 1.9: Dependency analysis (top) and topological analysis (bottom) of the scrambling example

1.5.2 *A Topological Model of Scrambling*

As we have proven [8], scrambling *can* be modeled in XDG. But how? There is no straightforward way of articulating appropriate word order constraints on the DEP dimension directly. At this point, we can make use of the multi-dimensionality of XDG. The idea is to keep the dependency analysis on the DEP dimension as it is, and move all ordering constraints to an additional dimension called TOP. The models on TOP are *projective* trees which represent the topological structure of the sentence as in Topological Dependency Grammar (TDG) [15]. A TOP analysis of the example sentence is depicted in Fig. 1.9 (bottom). Here, the non-verbal dependents *Nilpferde*, *Maria* and *Hans* are dependents of the finite verb *soll* labeled mf for “Mittelfeld”. The verbal dependent of *soll*, *helfen*, and that of *helfen*, *füttern*, are labeled vcf for “verb cluster field”. With this additional dimension, articulating the appropriate word order constraints is straightforward: all mf dependents of the finite verb must precede its vcf dependents, and the mutual order of the mf dependents is unconstrained.

The relation between the DEP and TOP dimensions is such that the trees on TOP are a *flattening* of the corresponding trees on DEP. We can express this in XDG by requiring that the dominance relation on TOP is a subset of the dominance relation on DEP:

$$\forall v : \forall v' : \; v \rightarrow^{+}_{\text{TOP}} v' \;\Rightarrow\; v \rightarrow^{+}_{\text{DEP}} v'$$

This principle is called the *climbing principle* [15], and gets its name from the observation that the non-verbal dependents seem to “climb up” from their position on DEP to a higher position on TOP. For example, in Fig. 1.9, the noun *Nilpferde* is a dependent of *füttern* on DEP, and climbs up to become a dependent of the finite verb *soll* on TOP.

Just using the climbing principle is too permissive. For example, in German, extraction of determiners and adjectives out of noun phrases must be ruled out, whereas relative clauses *can* be extracted. To this end, we apply a principle called *barriers principle* [15], which allows each word to “block” certain dependents from climbing up. This allows us to express that nouns block their determiner and adjective dependents from climbing up, but not their relative clause dependents.

1.6 A Relational Syntax-Semantics Interface

Our second application of XDG is the realization of a new, relational syntax-semantics interface [11]. The interface is relational in the sense that it constrains the *relation* between the syntax and the semantics, as opposed to the traditional functional approach where the semantics is *derived* from syntax. In combination with the constraint-based implementation of XDG, the main advantage of this approach is *bi-directionality*: the same grammar can be “reversed” and be used for generation, and constraints and preferences can “flow back” from the semantics to disambiguate the syntax. In this section, we introduce the subset of the full relational syntax-semantics interface for XDG [11], concerned only with the relation between grammatical functions and quantifier scope. We model quantifier scope using dominance constraints, the integral part of the Constraint Language for Lambda Structures (CLLS) [16].

1.6.1 Dominance Constraints

Dominance constraints can be applied to model the *underspecification* of scopal relationships. For example, the sentence *Everybody loves somebody* has two scopal readings: one where everybody loves the same somebody, and one where everybody loves somebody else. The first reading is called the *strong reading*: here, *somebody* takes scope over *everybody*, which in turn takes scope over *loves*. In the second reading, the *weak reading*, *everybody* takes scope over *somebody* over *loves*. Using

dominance constraints, it is possible to model both readings in one underspecified representation called *dominance constraint*:

$$X_1 : everybody(X_1') \wedge X_2 : loves \wedge X_3 : somebody(X_3') \wedge X_1' \lhd^* X_2 \wedge X_3' \lhd^* X_2 \tag{1.3}$$

The dominance constraint comprises three *labeling literals* and two *dominance literals*. A labeling literal such as $X_1 : everybody(X_1')$ assigns labels to node variables, and constrains the daughters: X_1 must have the label *everybody*, and it must have one daughter, viz. X_1'. The dominance literals $X_1' \lhd^* X_2$ and $X_3' \lhd^* X_2$ stipulate that the node variables X_1' and X_3' must dominate (or be equal to) the node variable X_2, expressing that the node variables corresponding to *everybody* and *somebody* must dominate *loves*, but that their mutual dominance relationship is unknown.

The models of dominance constraints are trees called *configurations*. The example dominance constraint (1.3) represents the two configurations displayed in Fig. 1.10 (a) (strong reading) and (b) (weak reading).

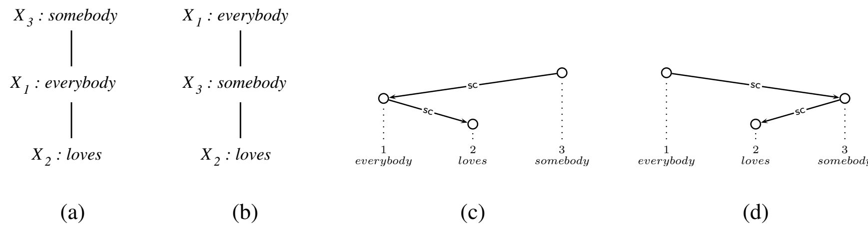


Fig. 1.10: (a) Configuration representing the strong reading, (b) the weak reading of *Everybody loves somebody*, (c) corresponding XDG dependency tree for the strong reading, (d) weak reading.

In XDG, we represent the configurations on a dimension called QS for *quantifier scope analysis*. For example, the configuration in Fig. 1.10 (a) corresponds to the XDG dependency tree in Fig. 1.10 (c), and (b) to (d). The QS dimension must satisfy only one principle: it must have tree-shape.

We model the dominance constraint itself by translating the labeling literals into constraints on the node-word mapping, and the dominance literals into dominance predicates. Hereby, we conflate the node variables participating in labeling literals such as $X_1 : everybody(X_1')$ into individual nodes.[3] We translate the dominance constraint (1.3) into the following XDG principle:

$$w(1) = everybody \wedge w(2) = loves \wedge w(3) = somebody \wedge 1 \rightarrow^*_{\text{QS}} 2 \wedge 3 \rightarrow^*_{\text{QS}} 2 \tag{1.4}$$

The set of QS tree structures which satisfy this principle corresponds precisely to the set of configurations of the dominance constraint in (1.3), i.e., the two dependency trees in Fig. 1.10 (c) and (d).

[3] In our simple example, the labeling literals have at most one daughter. In a more realistic setting [8], we distinguish the daughters of labeling literals with more than one daughter using distinct edge labels.

1.6.2 The Interface

We now apply this formalization of dominance constraints in XDG for building a relational syntax-semantics interface. The interface relates DEP, a syntactic dimension representing grammatical functions, and QS, a semantic dimension representing quantifier scope, and consists of two ingredients: the additional interface dimension DEP/QS, and an additional interface principle. The models on DEP/QS have no edges, as the sole purpose of this dimension is to carry the lexical attribute *dom* specifying how the syntactic dependencies on DEP relate to the quantifier scope dependencies on QS. The value of *dom* is a set of DEP edge labels, and for each node, all syntactic dependents with a label in *dom* must dominate the node on QS. We call the corresponding principle, already formalized in (1.2), *dominance principle*.

This kind of syntax-semantics interface is "two-way", or *bi-directional*: information does not only flow from syntax to semantics, but also vice versa. Like a functional interface, given a syntactic representation, the relational interface is able to derive the corresponding semantic representation. For example, the two syntactic dependencies from node 2 (*loves*) to node 1 (labeled subj) and to node 3 (labeled obj) in Fig. 1.7, together with the dominance principle, yield the information that both the subject and the object of *loves* must dominate it on the QS dimension.

The relational syntax-semantics interface goes beyond the functional one in its ability to let information from the semantics "flow back" to the syntax. For example, assume that we start with a partial QS structure including the information that *everybody* and *somebody* both dominate *loves*. Together with the dominance principle, this excludes any edges from *everybody* to *loves* and from *somebody* to *loves* on DEP.[4] Thus, information from the semantics has disambiguated the syntax. This bi-directionality can also be exploited for "reversing" grammars to be used for generation as well as for parsing [28, 7].

1.7 Modeling Regular Dependency Grammars

In this section, we apply XDG to obtain a multi-dimensional model of regular dependency grammars (REGDG). This not only gives us a lower bound of the expressivity of XDG, but also yields techniques for parsing TAG grammars. We demonstrate the application of the latter to large-scale parsing in Sect. 1.8.2.

Our modeling of REGDG proceeds in two steps. In the first, we examine the *structures* that they talk about: totally ordered dependency trees. To ease the transition to XDG, we replace the node labels in the REGDG dependency trees by edge labels in the XDG dependency trees. Fig. 1.11 (top) shows such a dependency tree of the string *aaabbb*. We call the XDG dependency tree dimension DEP. On the DEP dimension, the models must have tree-shape but need not be projective.

[4] For the example, we assume that the value of *dom* for *everybody* and *somebody* includes all edge labels.

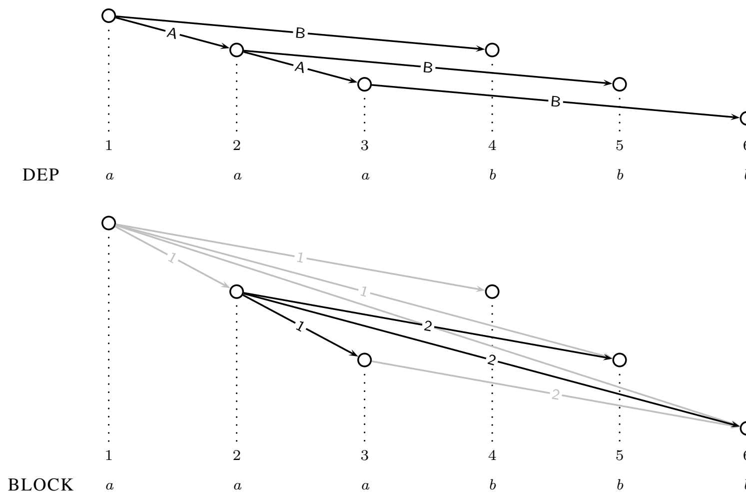


Fig. 1.11: XDG dependency tree (top) and XDG block graph (bottom) for the string *aaabbb*

In the second step, we examine the *rules* of REGDG. They can be best explained by example. Consider the rule:

$$A \rightarrow \langle a, \langle 01,21 \rangle \rangle (A,B) \tag{1.5}$$

which is expanded by the second *a* in Fig. 1.11. First, the rule stipulates that a head with incoming edge label A associated with the word *a* must have two dependents: one labeled A and one labeled B. Second, the rule stipulates the order of the yields of the dependents and the head, where the yields are divided into contiguous sets of nodes called *blocks*. In the order tuples (e.g. $\langle 01,21 \rangle$), 0 represents the head, 1 the blocks in the yield of the first dependent (here: A), and 2 the blocks in the yield of the second dependent (here: B). The tuple $\langle 01,21 \rangle$ from the example rule then states: the yield of the A dependent must consist of two blocks (two occurrences of 1 in the tuple) and that of the B dependent of one block (one occurrence of 2), the head must precede the first block of the A dependent, which must precede the first (and only) block of the B dependent, which must precede the second block of the A dependent, and the yield of the head must be divided into two blocks, where the gap is between the first block of the A dependent and the first (and only) block of the B dependent.

As REGDG do not only make statements on dependency structures but also on the yields of the nodes, we exploit the multi-dimensionality of XDG and introduce a second dimension called BLOCK. The structures on the BLOCK dimension are graphs representing the function from nodes to their yields on DEP. That is, each edge from v to v' on BLOCK corresponds to a sequence of zero or more edges from v to v' on DEP:

$$\forall v : \forall v' : \; v \rightarrow_{\mathrm{QS}} v' \; \Leftrightarrow \; v \rightarrow^{*}_{\mathrm{DEP}} v'$$

An edge from v to v' labeled i on BLOCK states that v' is in the ith block of the yield of v on DEP.

We model that the blocks are *contiguous* sets of nodes by a principle stipulating that for all pairs of edges, one from v to v', and one from v to v'', both labeled with the same label l, the set of nodes between v' and v'' must also be in the yield of v:

$$\forall v : \forall v' : \forall v'' : \forall l : (v \xrightarrow{l}_{\text{BLOCK}} v' \wedge v \xrightarrow{l}_{\text{BLOCK}} v'') \Rightarrow (\forall v''' : v' < v''' \wedge v''' < v'' \Rightarrow v \rightarrow^{*}_{\text{BLOCK}} v''')$$

Fig. 1.11 (bottom)[5] shows an example BLOCK graph complementing the DEP tree in Fig. 1.11 (top). On DEP, the yield of the second a (node 2) consists of itself and the third a (node 3) in the first block, and the second b and the third b (nodes 5 and 6) in the second block. Hence, on the BLOCK dimension, the node has four dependents: itself and the third a are dependents labeled 1, and the second b and the third b are dependents labeled 2.

We model the rules of the REGDG in XDG in four steps. First, we lexically constrain the incoming and outgoing edges of the nodes on DEP. For example, to model the example rule (1.5), we stipulate that the node associated with the word a must have precisely one incoming edge labeled A, and one A and one B dependent, as shown in Fig. 1.12 (a).

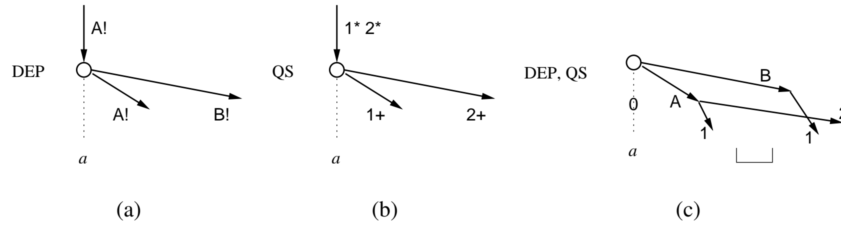


Fig. 1.12: (a) Constraints on DEP, (b) BLOCK, and (c) on both DEP and BLOCK

Second, we lexically constrain the incoming and outgoing edges on BLOCK. As each node on BLOCK can end up in any block of any other node, each node may have arbitrary many incoming edges either labeled 1 or 2. The constraint on the outgoing edges reflects the number of blocks into which the yield of the node must be divided. For the example rule (1.5), the yield must be divided into two blocks, and hence the node must have one or more dependents labeled 1, and one or more labeled 2, as depicted in Fig. 1.12 (b).

Third, we lexically constrain the order of the blocks of the DEP dependents. We do this by a constraint relating the DEP and BLOCK dimensions. For the example rule (1.5), we must order the head to the left of the first block of the A dependent. In terms of our XDG model, as illustrated in Fig. 1.12 (c), we must order the head to the left of all 1 dependents on BLOCK of the A dependent on DEP. Similarly, we

[5] For clarity, the graphical representation does not include the edges from each node to itself, and all edges except those emanating from node 2 are ghosted.

must order the 1 dependents of the A dependent to the left of the 1 dependents of the B dependent, and these in turn to the left of the 2 dependents of the A dependent.

Fourth, we lexically model the location of the gaps between the blocks. In the example rule (1.5), there is one gap between the first block of the A dependent and the first (and only) block of the B dependent, as indicated in Fig. 1.12 (c).

1.8 Grammar Development Environment

We have complemented the theoretical exploration of dependency grammars using XDG with the development of a comprehensive grammar development environment, the XDG Development Kit (XDK) [13, 8, 47]. The XDK includes a parser, a powerful grammar description language, an efficient compiler for it, various tools for testing and debugging, and a graphical user interface, all geared towards rapid prototyping and the verification of new ideas. It is written in MOZART/OZ [50, 38]. A snapshot of the XDK is depicted in Fig. 1.13.

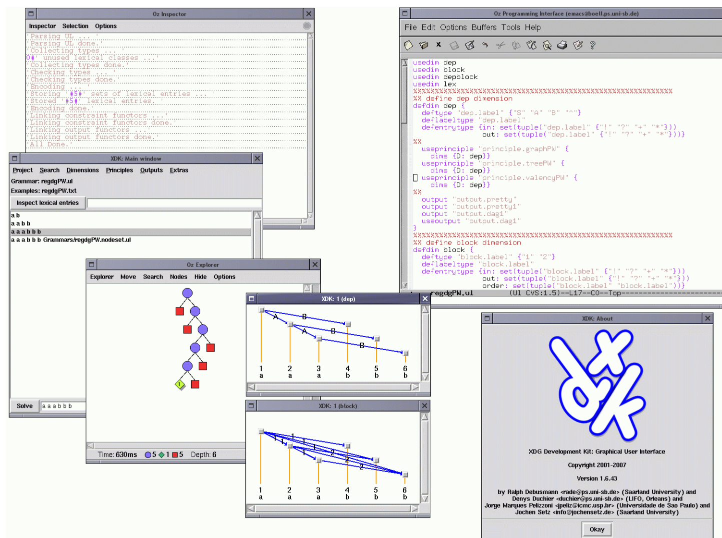


Fig. 1.13: The XDG Development Kit (XDK)

1.8.1 Parser

The included parser is based on constraint programming [45], a modern technique for solving NP-complete problems. For efficient parsing, the applied constraints implementing the XDG principles must be fine-tuned. Fine-tuned implementations of the principles of the account of word order outlined in Sect. 1.5, and the relational syntax-semantics interface outlined in Sect. 1.6 already exist, and have yielded efficiently parsable, smaller-scale grammars for German [6, 2], and English [8]. Koller and Striegnitz show that an implementation of TDG can be applied for efficient TAG generation [28].

1.8.2 Large-Scale Parsing

In this section, we answer the question whether the constraint parser of the XDK actually scales up for large-scale parsing. We find a positive answer to this question by showing that the parser can be fine-tuned for parsing the large-scale TAG grammar XTAG [55], such that most of the time, it finds the first parses of a sentence *before* a fast TAG chart parser with polynomial time complexity. This is surprising given that the XDK constraint parser has exponential time complexity in the worst case.

For our experiment, we applied the most recent version of the XTAG grammar from February 2001, which has a full form lexicon of 45171 words and 1230 elementary treyes. The average lexical ambiguity is 28 elementary trees per word, and the maximum lexical ambiguity 360 (for *get*). Verbs are typically assigned more than 100 elementary trees. We developed an encoding of the XTAG grammar into XDG based on ideas from [12] and our encoding of regular dependency grammars (Sect. 1.7), and implemented these ideas in the XDK.

We tested the XDK with this grammar on a subset of section 23 of the Penn Treebank, where we manually replaced words not in the XTAG lexicon by appropriate words from the XTAG lexicon. We compared our results with the official XTAG parser: the LEM parser [44], a chart parser implementation with polynomial complexity. For the LEM parser, we measured the time required for building up the chart, and for the XDK parser, the time required for the first solution and the first 1000 solutions. Contrary to the LEM parser, the XDK parser does not build up a chart representation for the efficient enumeration of parses. Hence one of the most interesting questions was how long the XDK parser would take to find not only the first but the first 1000 parses.

We did not use the supertagger included in the LEM package, which significantly increases its efficiency at the cost of accuracy [44]. We must also note that longer sentences are assigned up to millions of parses by the XTAG grammar, making it unlikely that the first 1000 parses found by the constraint parser also include the *best* parses. This could be remedied with sophisticated search techniques for constraint parsing [14, 39].

We parsed 596 sentences of section 23 of the Penn Treebank whose length ranged from 1 to 30 on an Athlon 64 3000+ processor with 1 GByte of RAM. The average sentence length was 12.36 words. From these 596 sentences, we first removed all those which took longer than a timeout of 30 minutes using either the LEM or the XDK parser. The LEM parser exceeded the timeout in 132 cases, and the XDK in 94 cases, where 52 of the timeouts were shared among both parsers. As a result, we had to remove 174 sentences to end up with 422 sentences where neither LEM nor the XDK had exceeded the timeout. They have an average length of 10.73 words.

The results of parsing these remaining 422 sentences is shown in Table 1.2. Here, the second column shows the time the LEM parser required for building up the chart, and the percentage of exceeded timeouts. The third and fourth column show the times required by the standard XDK parser (using the constraint engine of MOZART/OZ 1.3.2) for finding the first parse and the first 1000 parses, and the percentage of exceeded timeouts. The fourth and fifth column show the times when replacing the standard MOZART/OZ constraint engine with the new, faster GECODE 2.0.0 constraint library [46], and again the percentage of exceeded timeouts.

Interestingly, despite the polynomial complexity of the LEM parser, the XDK parser not only less often ran into the 30 minute timeout, but was also faster than LEM on the remaining sentences. Using the standard MOZART/OZ constraint engine, the XDK found the first parse 3.2 times faster, and using GECODE, 16.8 times faster. Even finding the first 1000 parses was 1.7 (MOZART/OZ) and 7.8 (GECODE) times faster. The gap between LEM and the XDK parser increased with increased sentence length. Of the sentences between 16 and 30 words, the LEM parser exceeded the timeout in 82.14% of the cases, compared to 45.54% (MOZART/OZ) and 38.39% (GECODE). Finding the first parse of the sentences between 16 and 30 words was 8.9 times faster using MOZART/OZ, and 41.1 times faster using GECODE. The XDK parser also found the first 1000 parses of the longer sentences faster than LEM: 5.2 times faster using MOZART/OZ and 19.8 times faster using GECODE.

	LEM	XDK			
		MOZART/OZ		GECODE	
		1 parse	1000 parses	1 parse	1000 parses
1 – 30 words	200.47s	62.96s	117.29s	11.90s	25.72s
timeouts	132 (22.15%)	93 (15.60%)	94 (15.78%)	60 (10.07%)	60 (10.07%)
1 – 15 words	166.03s	60.48s	113.43s	11.30s	24.52s
timeouts	40 (8.26%)	42 (8.68%)	43 (8.88%)	17 (3.51%)	17 (3.51%)
16 – 30 words	1204.10s	135.24s	229.75s	29.33s	60.71s
timeouts	92 (82.14%)	51 (45.54%)	51 (45.54%)	43 (38.39%)	43 (38.39%)

Table 1.2: Results of the XTAG parsing experiment

1.9 Conclusion

The goals of the research reported in this chapter were to classify dependency grammars in terms of their generative capacity and parsing complexity, and to explore their expressive power in the context of a practical system. To reach the first goal, we have developed the framework of *regular dependency grammars*, which provides a link between dependency structures on the one hand, and mildly context-sensitive grammar formalisms such as TAG on the other. To reach the second goal, we have designed a new meta grammar formalism, XDG, implemented a grammar development environment for it, and used this to give novel accounts of linguistic phenomena such as word order variation, and to develop a powerful syntax-semantics interface. Taken together, our research has provided fundamental insights into both the theoretical and the practical aspects of dependency grammars, and a more accurate picture of their usability.

References

1. 45th Annual Meeting of the Association for Computational Linguistics (ACL) (2007)
2. Bader, R., Foeldesi, C., Pfeiffer, U., Steigner, J.: Modellierung grammatischer Phänomene der deutschen Sprache mit Topologischer Dependenzgrammatik (2004). Softwareprojekt, Saarland University
3. Becker, T., Rambow, O., Niv, M.: The derivational generative power, or, scrambling is beyond LCFRS. Tech. rep., University of Pennsylvania (1992)
4. 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 (2005)
5. Culotta, A., Sorensen, J.: Dependency tree kernels for relation extraction. In: 42nd Annual Meeting of the Association for Computational Linguistics (ACL), pp. 423–429. Barcelona, Spain (2004). DOI 10.3115/1218955.1219009
6. Debusmann, R.: A declarative grammar formalism for dependency grammar. Diploma thesis, Saarland University (2001). Http://www.ps.uni-sb.de/Papers/abstracts/da.html
7. Debusmann, R.: Multiword expressions as dependency subgraphs. In: Proceedings of the ACL 2004 Workshop on Multiword Expressions: Integrating Processing. Barcelona/ES (2004)
8. Debusmann, R.: Extensible dependency grammar: A modular grammar formalism based on multigraph description. Ph.D. thesis, Universität des Saarlandes (2006)
9. Debusmann, R.: The complexity of First-Order Extensible Dependency Grammar. Tech. rep., Saarland University (2007)
10. Debusmann, R.: Scrambling as the intersection of relaxed context-free grammars in a model-theoretic grammar formalism. In: ESSLLI 2007 Workshop Model Theoretic Syntax at 10. Dublin/IE (2007)
11. Debusmann, R., Duchier, D., Koller, A., Kuhlmann, M., Smolka, G., Thater, S.: A relational syntax-semantics interface based on dependency grammar. In: Proceedings of COLING 2004. Geneva/CH (2004)
12. Debusmann, R., Duchier, D., Kuhlmann, M., Thater, S.: TAG as dependency grammar. In: Proceedings of TAG+7. Vancouver/CA (2004)
13. Debusmann, R., Duchier, D., Niehren, J.: The XDG grammar development kit. In: Proceedings of the MOZ04 Conference, *Lecture Notes in Computer Science*, vol. 3389, pp. 190–201. Springer, Charleroi/BE (2004)

14. Dienes, P., Koller, A., Kuhlmann, M.: Statistical A* dependency parsing. In: Prospects and Advances in the Syntax/Semantics Interface. Nancy/FR (2003)
15. Duchier, D., Debusmann, R.: Topological dependency trees: A constraint-based account of linear precedence. In: Proceedings of ACL 2001. Toulouse/FR (2001)
16. Egg, M., Koller, A., Niehren, J.: The Constraint Language for Lambda Structures. Journal of Logic, Language, and Information (2001)
17. 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), pp. 457–464. College Park, MD, USA (1999). DOI 10.3115/1034678.1034748
18. Gaifman, H.: Dependency systems and phrase-structure systems. Information and Control **8**, 304–337 (1965)
19. 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)
20. 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) [1], pp. 608–615. URL http://www.aclweb.org/anthology/P/P07/P07-1077.pdf
21. Hays, D.G.: Dependency theory: A formalism and some observations. Language **40**(4), 511–525 (1964). DOI 10.2307/411934
22. Holan, T., Kuboň, V., Oliva, K., Plátek, M.: Two useful measures of word order complexity. In: Workshop on Processing of Dependency-Based Grammars, pp. 21–29. Montréal, Canada (1998)
23. Hotz, G., Pitsch, G.: On parsing coupled-context-free languages. Theoretical Computer Science **161**(1–2), 205–233 (1996). DOI 10.1016/0304-3975(95)00114-X
24. Hudson, R.A.: English Word Grammar. B. Blackwell, Oxford/UK (1990)
25. Huybregts, R.: The weak inadequacy of context-free phrase structure grammars. In: G. de Haan, M. Trommelen, W. Zonneveld (eds.) Van periferie naar kern, pp. 81–99. Foris, Dordrecht, The Netherlands (1984)
26. 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 (1985)
27. Joshi, A.K., Schabes, Y.: Tree-Adjoining Grammars. In: G. Rozenberg, A. Salomaa (eds.) Handbook of Formal Languages, vol. 3, pp. 69–123. Springer (1997)
28. Koller, A., Striegnitz, K.: Generation as dependency parsing. In: Proceedings of ACL 2002. Philadelphia/US (2002)
29. Kruijff, G.J.M.: Dependency grammar. In: Encyclopedia of Language and Linguistics, 2nd edn., pp. 444–450. Elsevier (2005)
30. Kuhlmann, M.: Dependency structures and lexicalized grammars. Doctoral dissertation, Saarland University, Saarbrücken, Germany (2007)
31. Kuhlmann, M., Möhl, M.: Mildly context-sensitive dependency languages. In: 45th Annual Meeting of the Association for Computational Linguistics (ACL) [1], pp. 160–167. URL http://www.aclweb.org/anthology/P07-1021
32. Kuhlmann, M., Möhl, M.: The string-generative capacity of regular dependency languages. In: Twelfth Conference on Formal Grammar. Dublin, Ireland (2007)
33. Kuhlmann, M., Nivre, J.: Mildly non-projective dependency structures. In: 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING-ACL), Main Conference Poster Sessions, pp. 507–514. Sydney, Australia (2006). URL http://www.aclweb.org/anthology/P06-2000
34. Marcus, S.: Algebraic Linguistics: Analytical Models, *Mathematics in Science and Engineering*, vol. 29. Academic Press, New York, USA (1967)
35. McDonald, R., Satta, G.: On the complexity of non-projective data-driven dependency parsing. In: Tenth International Conference on Parsing Technologies (IWPT), pp. 121–132. Prague, Czech Republic (2007). URL http://www.aclweb.org/anthology/W/W07/W07-2216
36. Mel'čuk, I.: Dependency Syntax: Theory and Practice. State Univ. Press of New York, Albany/US (1988)

37. Mezei, J.E., Wright, J.B.: Algebraic automata and context-free sets. Information and Control **11**(1–2), 3–29 (1967). DOI 10.1016/S0019-9958(67)90353-1
38. Mozart Consortium: The Mozart-Oz website (2007). Http://www.mozart-oz.org/
39. Narendranath, R.: Evaluation of the stochastic extension of a constraint-based dependency parser (2004). Bachelorarbeit, Saarland University
40. Neuhaus, P., Bröker, N.: The complexity of recognition of linguistically adequate dependency grammars. In: 35th Annual Meeting of the Association for Computational Linguistics (ACL), pp. 337–343. Madrid, Spain (1997). DOI 10.3115/979617.979660
41. Nivre, J.: Constraints on non-projective dependency parsing. In: Eleventh Conference of the European Chapter of the Association for Computational Linguistics (EACL), pp. 73–80. Trento, Italy (2006)
42. 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), pp. 915–932. Prague, Czech Republic (2007). URL http://www.aclweb.org/anthology/D/D07/D07-1096
43. Quirk, C., Menezes, A., Cherry, C.: Dependency treelet translation: Syntactically informed phrasal SMT. In: 43rd Annual Meeting of the Association for Computational Linguistics (ACL), pp. 271–279. Ann Arbor, USA (2005). DOI 10.3115/1219840.1219874
44. Sarkar, A.: Complexity of Lexical Descriptions and its Relevance to Natural Language Processing: A Supertagging Approach, chap. Combining SuperTagging with Lexicalized Tree-Adjoining Grammar Parsing. MIT Press (2007)
45. Schulte, C.: Programming Constraint Services, *Lecture Notes in Artificial Intelligence*, vol. 2302. Springer-Verlag (2002)
46. Schulte, C., Lagerkvist, M., Tack, G.: GECODE—Generic Constraint Development Environment (2007). Http://www.gecode.org/
47. Setz, J.: A principle compiler for Extensible Dependency Grammar. Tech. rep., Saarland University (2007). Bachelorarbeit
48. Sgall, P., Hajicova, E., Panevova, J.: The Meaning of the Sentence in its Semantic and Pragmatic Aspects. D. Reidel, Dordrecht/NL (1986)
49. Shieber, S.M.: Evidence against the context-freeness of natural language. Linguistics and Philosophy **8**(3), 333–343 (1985). DOI 10.1007/BF00630917
50. Smolka, G.: The Oz programming model. In: J. van Leeuwen (ed.) Computer Science Today, Lecture Notes in Computer Science, vol. 1000, pp. 324–343. Springer-Verlag, Berlin/DE (1995)
51. Tesnière, L.: Éléments de syntaxe structurale. Klinksieck, Paris, France (1959)
52. Veselá, K., Havelka, J., Hajičová, E.: Condition of projectivity in the underlying dependency structures. In: 20th International Conference on Computational Linguistics (COLING), pp. 289–295. Geneva, Switzerland (2004). DOI 10.3115/1220355.1220397
53. 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), pp. 104–111. Stanford, CA, USA (1987). DOI 10.3115/981175.981190
54. Weir, D.J.: Characterizing mildly context-sensitive grammar formalisms. Ph.D. thesis, University of Pennsylvania, Philadelphia, USA (1988). URL http://wwwlib.umi.com/dissertations/fullcit/8908403
55. XTAG Research Group: A Lexicalized Tree Adjoining Grammar for English. Tech. Rep. IRCS-01-03, IRCS, University of Pennsylvania (2001)
56. Yli-Jyrä, A.: Multiplanarity – a model for dependency structures in treebanks. In: Second Workshop on Treebanks and Linguistic Theories (TLT), pp. 189–200. Växjö, Sweden (2003)