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 [27]. In recent years, they have also become increasingly used in computational tasks, such as information extraction [2], machine translation [39], and parsing [38]. 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 [17].

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 grammar exist (examples are Functional Generative Description [44], Meaning-Text Theory [34], and Word Grammar [22]), there are few results on their formal properties—in particular, it is not clear how they are 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*. In Sect. 1.8, 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 [47]:

> 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 [19], we use dotted lines to indicate the left-to-right ordering of the words in the sentence.

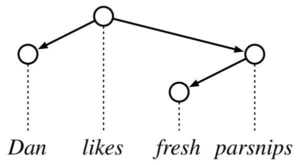
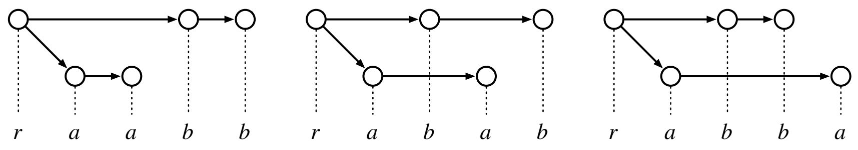


Fig. 1.1: A dependency structure

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 have only one independent word (usually the main verb). The dependency analysis shown in Fig. 1.1 satisfies the treeness constraint.

Another well-known constraint on dependency structures is *projectivity* [32]. 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 [14]. However, in recent literature, there is a growing interest in *non-projective* dependency trees, 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 [20, 48]. Unfortunately, without any further restrictions, non-projective dependency parsing is intractable [36, 33].

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 [52, 37, 18]. Such constraints identify classes of what we may call *mildly non-projective dependency structures*. One important constraint is the *block-degree restriction* [20], which relaxes projectivity such that dependency subtrees can be distributed over more than one block. For example, in Fig. 1.2b, each of the marked subtrees spans two blocks. In our own work, we have proposed the *well-nestedness condition* [1], which says that pairs of disjoint dependency subtrees must not cross—this means that there



(a) degree 1, well-nested (b) degree 2, ill-nested (c) degree 2, well-nested

Fig. 1.2: Three dependency structures

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 [31] (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. In this work we were interested in the generative capacity of grammars when interpreted 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, and more succinct and arguably closer to predicate-argument structure than parse trees. Second, an answer to the question allows us to tap the rich resource of formal results about phrase structure-based 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 phrase structure-based formalisms. This can help us identify the polynomial fragments of non-projective dependency parsing.

1.3.1 Lexicalized Grammars Induce Dependency Structures

We now explain how a grammar can be interpreted 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.

$$\begin{aligned} \text{S} &\rightarrow \text{SUBJ } \textit{likes} \text{ OBJ} \\ \text{SUBJ} &\rightarrow \textit{Dan} \\ \text{OBJ} &\rightarrow \text{MOD } \textit{parsnips} \\ \text{MOD} &\rightarrow \textit{fresh} \end{aligned}$$

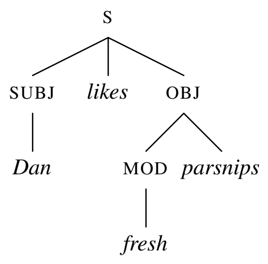


Fig. 1.3: A context-free grammar and a parse tree generated by this grammar

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 left half of Fig. 1.4 shows the derivation tree for the parse tree from our example.

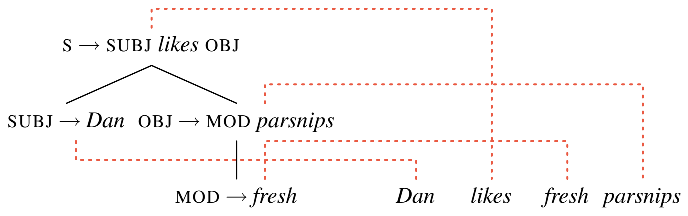


Fig. 1.4: Lexicalized derivations induce dependency structures

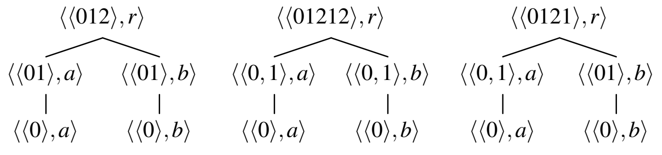
If the underlying grammar is lexicalized, then there is a one-to-one correspondence between the nodes in the derivation tree and the positions in the derived string: each occurrence of a production participating in the derivation contributes exactly one terminal symbol to this string. If we order the 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.

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 [23, 45] 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) [25]. 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 extended 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 [24]. A central result of our work is a mathematically precise classification of TAG and other mildly context-sensitive grammar formalisms with respect to the kinds of 'crossing' dependencies that these formalisms can induce.

While researchers in the field have a strong intuition that the generative capacity of a grammar formalism and the structural properties of the dependency structures that this formalism can induce are intimately related, there have been only few formal results on these matters. A fundamental reason for this is that, while constraints such as projectivity, block-degree, and well-nestedness are immediately observable in a *structure*, it is not obvious how they fit into a *grammar*: During a derivation, the induced dependency structure is determined by its constituent parts and the operations that are used to combine them, but it is not clear what these constituents or combining operations are in the case of, say, well-nested dependency structures with block-degree at most 2. One of the central technical contributions of our work on connecting grammars and structures is a compositional description of structural constraints.

1.3.2 The Compositional View on Dependency Structures

In order to link structural constraints to 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 developed such a 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 different classes of dependency structures that we have discussed above stand in one-to-one correspondence with terms over specific subsets of this signature [29]. In the second step, we showed how order annotations can be interpreted as composition operations on dependency structures [28, 30], and proved that these operations can be freely simulated by term construction on the encodings.


$\langle\langle 012\rangle, r\rangle$
$\langle\langle 01\rangle, a\rangle$
$\langle\langle 01\rangle, b\rangle$
$\langle\langle 0\rangle, a\rangle$
$\langle\langle 0\rangle, b\rangle$
$\langle\langle 01212\rangle, r\rangle$
$\langle\langle 0,1\rangle, a\rangle$
$\langle\langle 0,1\rangle, b\rangle$
$\langle\langle 0\rangle, a\rangle$
$\langle\langle 0\rangle, b\rangle$
$\langle\langle 0121\rangle, r\rangle$
$\langle\langle 0,1\rangle, a\rangle$
$\langle\langle 01\rangle, b\rangle$
$\langle\langle 0\rangle, a\rangle$
$\langle\langle 0\rangle, b\rangle$

(a) t_1 (b) t_2 (c) t_3

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

To give an intuition for our formal framework, Fig. 1.5 shows the terms that correspond to the dependency structures in Fig. 1.2. Each order annotation in these terms (the first components in the node labels) encodes local information about the linear order. As an example, the annotation $\langle 0,1\rangle$ in Fig. 1.5b represents the information that both the 'a' subtree and the 'b' subtree in Fig. 1.2b consist of two blocks (the two components of the tuple $\langle 0, 1 \rangle$), with the root node (represented by the symbol 0) situated in the left block, and the subtree rooted at the first child (represented by the symbol 1) in the right block. 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.

1.3.3 Regular Dependency Grammars

With the term encoding at hand, we can also 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* or *regular dependency languages* [29]: We call a set of structures *regular*, if its term encoding forms a regular set of terms [16]. Regular sets of terms are standard in formal language theory, and have many characterizations: they are recognized by finite tree automata, definable in monadic second-order logic, and generated by regular term grammars. These grammars are very similar to standard context-free grammars, but generate terms instead of strings. Under the correspondence between dependency structures and terms discussed above, regular term grammars can be understood as generators of dependency structures, and thus as 'regular dependency grammars'.

To illustrate the idea, 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 'cross-serial' dependencies. We only give the productions for the verbs; the arguments are generated by rules such as $N \to \langle\langle 0\rangle, \mathit{Jan}\rangle$.

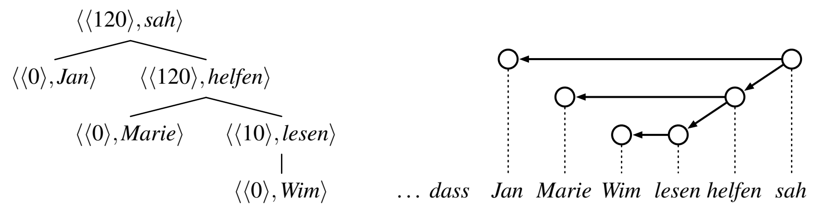
$$S \to \langle\langle 120\rangle, \mathit{sah}\rangle(N,V) \qquad V \to \langle\langle 120\rangle, \mathit{helfen}\rangle(N,V) \mid \langle\langle 10\rangle, \mathit{lesen}\rangle(N) \qquad (G_1)$$

$$S \to \langle\langle 1202\rangle, \mathit{zag}\rangle(N,V) \qquad V \to \langle\langle 12,02\rangle, \mathit{helpen}\rangle(N,V) \mid \langle\langle 1,0\rangle, \mathit{lezen}\rangle(N) \qquad (G_2)$$

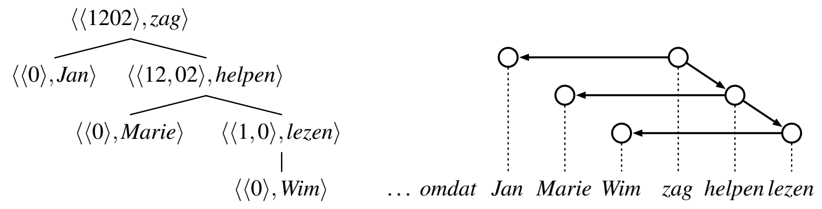
Figure 1.6 shows terms generated by these grammars, and the corresponding dependency structures.

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 the string-generative capacity and parsing complexity. First, we can show that the block-degree measure gives rise to an infinite hierarchy of ever more powerful string languages, and that adding the well-nestedness constraint leads to a proper decrease of string-generative power on nearly all levels of this hierarchy [30] (Fig. 1.7). Certain string languages enforce structural properties in the dependency languages that project them: For every natural number k, the language

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


$\langle\langle 120\rangle, sah\rangle$
$\langle\langle 0\rangle, Jan\rangle$
$\langle\langle 120\rangle, helfen\rangle$
$\langle\langle 0\rangle, Marie\rangle$
$\langle\langle 10\rangle, lesen\rangle$
$\langle\langle 0\rangle, Wim\rangle$
... dass Jan Marie Wim lesen helfen sah

(a) Grammar G_1 (nested dependencies)


$\langle\langle 1202\rangle, zag\rangle$
$\langle\langle 0\rangle, Jan\rangle$
$\langle\langle 12,02\rangle, helpen\rangle$
$\langle\langle 0\rangle, Marie\rangle$
$\langle\langle 1,0\rangle, lezen\rangle$
$\langle\langle 0\rangle, Wim\rangle$
... omdat Jan Marie Wim zag helpen lezen

(b) Grammar G_2 (cross-serial dependencies)

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

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 [28].

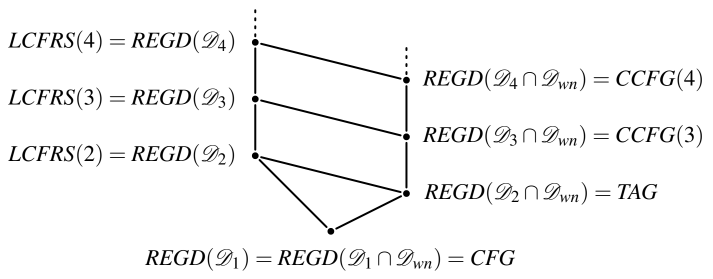

$LCFRS(4) = REGD(\mathscr{D}_4)$
$LCFRS(3) = REGD(\mathscr{D}_3)$
$LCFRS(2) = REGD(\mathscr{D}_2)$
$REGD(\mathscr{D}_4 \cap \mathscr{D}_{wn}) = CCFG(4)$
$REGD(\mathscr{D}_3 \cap \mathscr{D}_{wn}) = CCFG(3)$
$REGD(\mathscr{D}_2 \cap \mathscr{D}_{wn}) = TAG$
$REGD(\mathscr{D}_1) = REGD(\mathscr{D}_1 \cap \mathscr{D}_{wn}) = CFG$

Fig. 1.7: The hierarchy of regular dependency languages. Notation: $\mathscr{D}_k$, the class of all dependency structures with block-degree at most k; $\mathscr{D}_{wn}$, the class of well-nested structures

Our formal framework finally enables us to answer the question which grammars generate which sets of dependency structures [28]. In particular, we can extend Gaifman's result [15] that lexicalized context-free grammars generate only projective dependency structures into the realm of the mildly context-sensitive: The class of regular dependency languages over block-restricted structures ($REGD(\mathscr{D}_k)$ for some fixed k) is the class of dependency languages induced by Linear Context-Free Rewriting Systems (LCFRS) [49, 50]. The restriction to well-nested structures ($REGD(\mathscr{D}_k \cap \mathscr{D}_{wn})$) corresponds to the restriction to Coupled Context-Free Grammar (CCFG) [21]. As a special case, the regular dependency languages over well-nested structures with block-degree at most 2 ($REGD(\mathscr{D}_2 \cap \mathscr{D}_{wn})$) is exactly the class of dependency languages induced by (lexicalized) Tree Adjoining Grammar [25, 1]. The latter result is particularly interesting in the context of our treebank evaluation: It shows that, while unrestricted non-projective dependency parsing remains an NP-hard problem, for a class of structures that covers close to 99.5% of the data in the Prague Dependency Treebank, parsing is polynomial—every regular dependency grammar over well-nested structures with block-degree at most 2 can be converted into a lexicalized TAG in linear time, so that these grammars can be parsed in polynomial time using the parsing algorithms available for TAG.

Taken together, our classificatory results provide a rather complete picture of the relation between structural constraints such as block-degree and well-nestedness on the one hand, and language-theoretic properties such as generative capacity and parsing complexity on the other.

1.4 Extensible Dependency Grammar

In order to explore the power of dependency grammars, we have developed a new meta-grammatical framework called Extensible Dependency Grammar (XDG) [8, 5]. At the core of XDG is the principle of *multi-dimensionality*: an XDG analysis consists of a tuple of dependency structures, all sharing the same set of nodes, called *dependency multigraph*. The components of the multigraph are called *dimensions*. Multi-dimensionality was crucial for our formulations of a new, elegant model of complex word order phenomena, and a new, relational syntax-semantics interface.

1.4.1 Dependency Multigraphs

Dependency multigraphs are collections of dependency structures; they contain one structure for each linguistic dimension that we want to model. To give an example, consider the multigraph in Fig. 1.8. This graph has three dimensions: DEP (*dependency tree*), QS (*quantifier scope analysis*), and DEP/QS (DEP/QS *syntax-semantics interface*). The DEP dimension is used to model syntactic dependencies as before. The QS dimension represents certain aspects of the semantic structure of the sentence. Finally, the DEP/QS dimension captures information about how the syntactic and the semantic structures of the sentence interact.

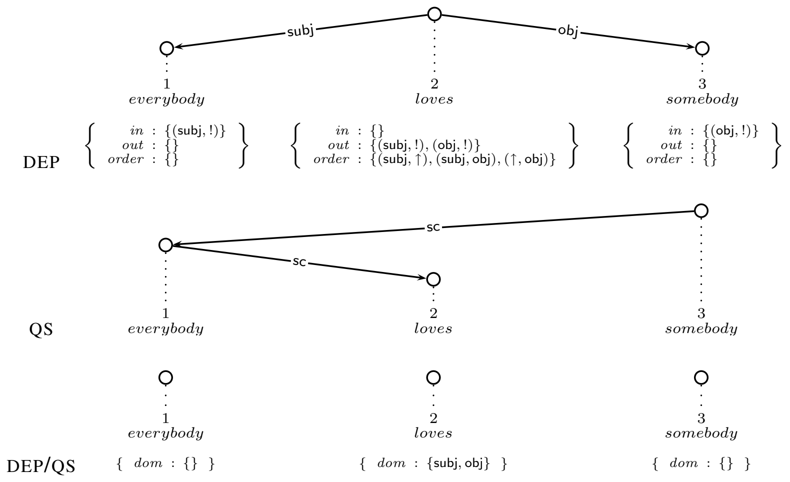


Fig. 1.8: Dependency multigraph for *Everybody loves somebody*

In an XDG multigraph, each dimension is a dependency graph made up of a set of nodes associated with *indices* (positions in the sentence), 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 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, inspired by the formalism of dominance constraints (described in another chapter of this volume). It models a reading of the sentence in which *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.

1.4.2 Grammars

Extensible Dependency Grammar 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 the perspective taken by the regular dependency grammars of Sect. 1.3.3, where the models are understood as being *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:

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

Observe that the principle is indeed satisfied in Fig. 1.8: 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 [6]: 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. The latter problem is the problem most relevant for parsing; in this sense, XDG parsing is NP-complete.

[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 [7].

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. 1.9: Example for scrambling

1.5 Modelling Complex Word Order Phenomena

To illustrate the power of multi-dimensionality, we now present two applications of XDG to different areas of linguistic description. The first application is the design of a new, elegant model of complex word order phenomena. Note that, due to space limitations, we have to simplify the description of both the linguistic data and our approach to modelling it. For details, we refer the reader to the cited publications.

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 almost totally free. This phenomenon is known as *scrambling*. We show an example in Fig. 1.9. The subscript attached to the words indicate the dependencies between the verbs and their arguments.

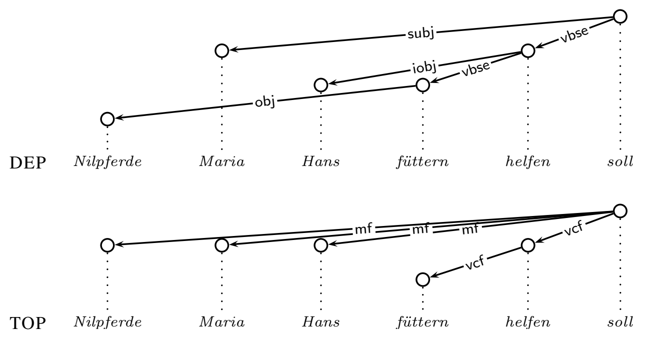


Fig. 1.10: Dependency analysis (top) and topological analysis (bottom) of the scrambling example

In the dependency analysis of the example sentence, given in Fig. 1.10 (top), we can see that scrambling gives rise to non-projectivity—in fact, it is easy to see that *unrestricted* scrambling even gives rise to an unbounded block-degree (see Sect. 1.2), which means that it cannot be modelled by the regular dependency grammars that we discussed in Sect. 1.3.3. However scrambling *can* be modeled in XDG [5]: While there is no straightforward way of articulating appropriate word order constraints on the DEP dimension directly, we can make use of multi-dimensionality. The idea is to keep the dependency analysis on the DEP dimension as is, and move all ordering constraints to an additional dimension called TOP, which models the topological structure of the sentence, as in Topological Dependency Grammar (TDG) [12]. The models on TOP are *projective* trees; one TOP analysis of the example sentence is depicted in Fig. 1.10 (bottom). Here, the non-verbal dependents *Nilpferde*, *Maria* and *Hans* are dependents of the finite verb *soll* labelled mf for 'Mittelfeld'. The verbal dependent of *soll*, *helfen*, and that of *helfen*, *füttern*, are labelled 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 by requiring the dominance relation on TOP to be 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* [12], 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. 1.10, 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* [12], 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 illustration for the power of multi-dimensionality is the realization of a new, relational syntax-semantics interface for dependency grammar [8]. The interface is relational in the sense that it constrains the *relation* between the syntax and the semantics, as opposed to the traditional 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 used for generation, and constraints and preferences can 'flow back' from the semantics to disambiguate the syntax. In this section, we introduce a subset of the full relational syntax-semantics interface for XDG [8], concerned only with the relation between grammatical functions and quantifier scope. Our modelling of scope is inspired by the formalism of dominance constraints [13].

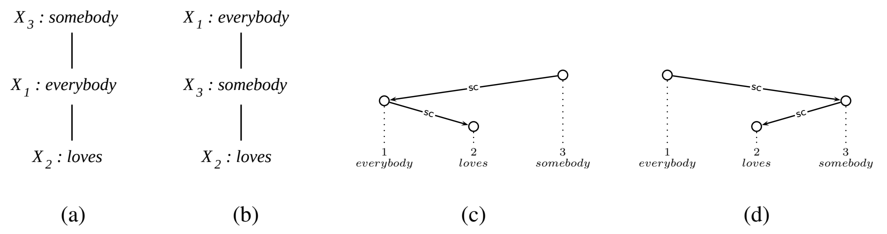


Fig. 1.11: Configurations representing (a) the strong reading, (b) the weak reading of *Everybody loves somebody*, (c) corresponding XDG dependency tree for the strong reading, (d) weak reading.

Dominance constraints can be applied to model underspecified scopal relationships. The example sentence *Everybody loves somebody* from Sect. 1.4 not only has the reading where everybody loves the same somebody (*somebody* takes scope over *everybody*), the so-called *strong reading*, but also one where everybody loves somebody else (*everybody* takes scope over *somebody*), the *weak reading*. Using dominance relations, we can model both readings in one underspecified description:

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

This formula expresses that both *everybody* and *somebody* must take scope over *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. 1.11 (a) and (b).

In XDG, we can represent the configurations on a dimension QS (*quantifier scope analysis*). For example, the configuration in Fig. 1.11 (a) corresponds to the XDG dependency tree in Fig. 1.11 (c), and (b) to (d). The QS dimension must satisfy only one principle: it must be a tree. We model the dominance constraint itself by translating the dominance literals $\lhd^*$ into constraints on the dependency graph. For example, the description (1.3) is translated into the following XDG constraint:

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

The set of QS tree structures which satisfy this constraint corresponds precisely to the set of configurations of the formula in (1.3).

We now apply the translation of dominance constraints in XDG to build a relational syntax-semantics interface. The interface relates the dimensions DEP (dependency structure) and QS, and consists of two ingredients: the additional interface dimension DEP/QS, and an interface principle. The models on DEP/QS have no edges, as the sole purpose of the multigraphs on this dimension is to carry the lexical attribute *dom* that specifies 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), the *dominance principle*.

Our syntax-semantics interface is ‘two-way’, or *bi-directional*: information does not only flow from syntax to semantics, but also vice versa. Most syntax-semantics interfaces are able to derive a semantic representation from a syntactic representation. For example, the two syntactic dependencies from node 2 (*loves*) to node 1 (labelled subj) and to node 3 (labelled obj) in Fig. 1.8, together with the dominance principle, yield the information that both the subject and the object of *loves* must dominate it on the QS dimension. Our interface goes beyond this in its ability to let information from semantics ‘flow back’ to 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 *loves* to *everybody* and from *loves* to *somebody* on DEP. In this way, information from semantics can help to disambiguate the syntax. This bi-directionality can also be exploited for ‘reversing’ grammars to be used for generation as well as for parsing [26, 4].

1.7 Translating Regular Dependency Grammars

The previous two sections have demonstrated the expressiveness of XDG using case studies. Before we address the practical implementation of the formalism, we discuss the relation between XDG and the regular dependency grammars from Sect. 1.3.3. In what follows, we use the abbreviation RDG for the class of these grammars. We have previously shown that XDG is at least as expressive as CFG [5], and that, at the same time, XDG is fundamentally different from CFG in that the string languages that can be characterized with XDG are closed under intersection [7]. In this section we will see that XDG is at least as expressive as RDG. As XDG is able to model scrambling (as we saw in Sect. 1.5), which RDG is not, this implies that XDG is indeed *more* expressive than RDG, and therefore, more expressive than all formalisms in the hierarchy from Fig. 1.7, such as TAG and LCFRS.

To translate an RDG into XDG, we use two dimensions: the first dimension, DEP (for *dependency tree*), models the tree aspect of the structures generated by the RDG; the second dimension, BLOCK, models the distribution of the nodes in a subtree over one or more blocks in the linear order. This is best explained by means of an example. We consider the following RDG rule:

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

Rule 1.5 can be used during an RDG derivation to rewrite the non-terminal symbol A. When used, it contributes the word a and introduces two new non-terminals, A and B, which need to be rewritten in subsequent steps. We model these requirements in XDG by assigning to the word a a lexical entry that requires an incoming edge labelled with the symbol A, and two outgoing edges, labelled with the symbols A and B, respectively. This is the entry that is selected for the second occurrence of the word a in the dependency structure shown in Fig. 1.12 (top). The first and third occurrence of a have entries that correspond to the rules

$$S \to \langle\langle 0121\rangle, a\rangle(A,B) \qquad \text{and} \qquad A \to \langle\langle 0,1\rangle, a\rangle(B)\,,$$

respectively. Note that the first of these rules, which rewrites the symbol S that we take as the distinguished start symbol of the grammar, is translated into a lexical entry that requires an empty set of incoming edges, and therefore can be used only to contribute the root node of the structure.

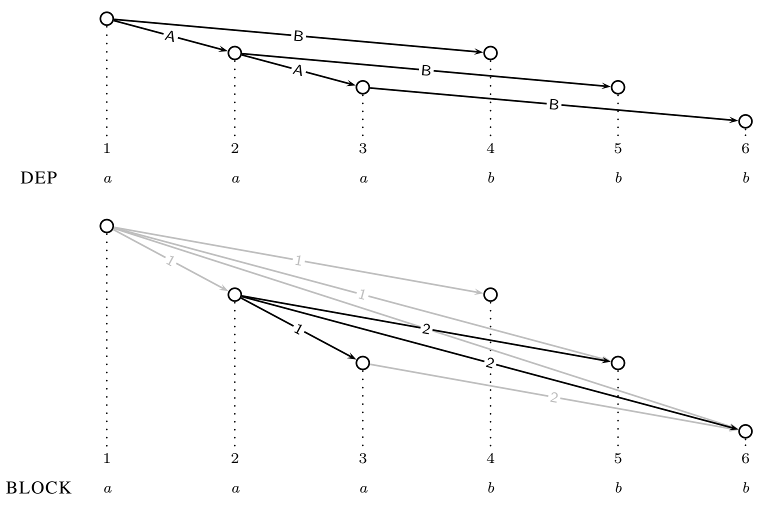


Fig. 1.12: XDG dependency tree (top) and XDG block graph (bottom) for the string *aaabbb*

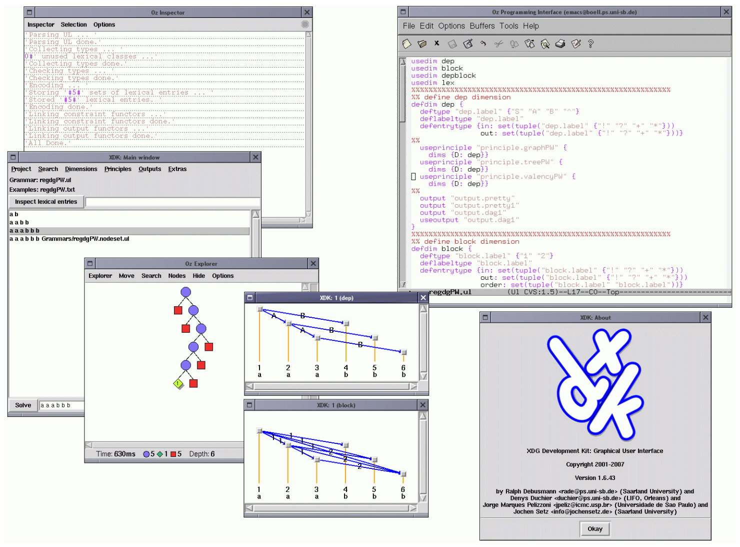
Using rule 1.5 during an RDG derivation not only determines the tree shape of the resultant dependency structure, but also its order; this is represented on the BLOCK dimension. Specifically, the order annotation $\langle 01,21\rangle$ of the word a specifies that the subtree rooted at that word is distributed over two blocks in the linear order: the first block is composed of a itself, succeeded by the first block of the subtree obtained by rewriting the non-terminal A; the second block is composed of the first (and only) block of the subtree obtained by rewriting the non-terminal B, and the second block of the A tree. Between the two blocks, there is a gap in the yield of a. We translate these requirements into a lexical entry for the second occurrence of a in Fig. 1.12 that specifies three pieces of information: First, it requires an outgoing edge labelled with 1 (‘first block’) to both the node itself and every node that is reached by an incoming edge labelled with 1 from the A-daughter of the node. Second, it requires an outgoing edge labelled with 2 (‘second block’) to every node reached by an incoming edge labelled with 1 from the B-daughter, and to every node reached by an incoming edge labelled with 2 from the A-daughter. These edges are drawn as solid edges in the lower half of Fig. 1.12. (For clarity, the Figure does not contain the edge from node 2 to itself, which would also be enforced by the translation.)

Third, the lexical entry enforces all nodes reached by an edge labelled with 1 to precede all nodes reached by an edge labelled with 2. For the structure as a whole, this guarantees that there is an i-labelled edge $v \longrightarrow_{\text{BLOCK}} v'$ if and only if v' belongs to the ith block in the yield of v. The only thing that remains to be done in order to determine the linear order is to require that the blocks are *contiguous* sets of nodes. This can be done by a principle that stipulates that for all pairs of edges, one from v to v', and one from v to v'', both labelled with the same label l, the set of nodes between v' and v'' must also be in the yield of v:

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

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) [10, 5, 43]. 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 [46, 35]. A snapshot of the XDK is depicted in Fig. 1.13.


Oz Inspector
Inspector Selection Options
XDK: Main window
Project Search Dimensions Principles Outputs Extras
Inspect lexical entries
Solve
Oz Explorer
Explorer Move Search Nodes Hide Options
Time: 630ms 5 1 5 Depth: 6
XDK: 1 (dep)
A
B
A
B
B
1 a
2 a
3 a
4 b
5 b
6 b
XDK: 1 (block)
1
2
1
2
1 a
2 a
3 a
4 b
5 b
6 b
Oz Programming Interface (emacs@boell.ps.uni-sb.de)
File Edit Options Buffers Tools Help
XDK: About
Okay

```
'Parsing UL ... '
'Parsing UL done.'
'Collecting types ... '
0#' unused lexical classes ...'
'Collecting types done.'
'Checking types ... '
'Checking types done.'
'Encoding ... '
'Storing '#5#' sets of lexical entries ... '
'Stored '#5#' lexical entries. '
'Encoding done.'
'Linking constraint functors ...'
'Linking constraint functors done.'
'Linking output functors ...'
'Linking output functors done.'
'All Done.'
```

Grammar: regdgPW.ul
Examples: regdgPW.txt

```
a b
a a b b
a a a b b b
a a a b b b Grammars/regdgPW.nodeset.ul
```

a a a b b b

```
usedim dep
usedim block
usedim depblock
usedim lex
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% define dep dimension
defdim dep {
  deftype "dep.label" {"S" "A" "B" "^"}
  deflabeltype "dep.label"
  defentrytype {in: set(tuple("dep.label" {"!" "?" "+" "*"}))
                out: set(tuple("dep.label" {"!" "?" "+" "*"}))}
%%
  useprinciple "principle.graphPW" {
    dims {D: dep}}
  useprinciple "principle.treePW" {
    dims {D: dep}}
  useprinciple "principle.valencyPW" {
    dims {D: dep}}
%%
  output "output.pretty"
  output "output.pretty1"
  output "output.dag1"
  useoutput "output.dag1"
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% define block dimension
defdim block {
  deftype "block.label" {"1" "2"}
  deflabeltype "block.label"
  defentrytype {in: set(tuple("block.label" {"!" "?" "+" "*"}))
                out: set(tuple("block.label" {"!" "?" "+" "*"}))
                order: set(tuple("block.label" "block.label"))}
    regdgPW.ul        (Ul CVS:1.5)--L17--C0--Top-------------------
```

XDG Development Kit: Graphical User Interface

Copyright 2001-2007

Version 1.6.43

by Ralph Debusmann <rade@ps.uni-sb.de> (Saarland University) and
Denys Duchier <duchier@ps.uni-sb.de> (LIFO, Orleans) and
Jorge Marques Pelizzoni <jpeliz@icmc.usp.br> (Universidade de Sao Paulo) and
Jochen Setz <info@jochensetz.de> (Saarland University)

Fig. 1.13: The XDG Development Kit (XDK)

The included parser is based on constraint programming [41], a modern technique for solving combinatorial problems. For efficient processing, 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 [3], and English [5]. Koller and Striegnitz show that an implementation of TDG can be applied for efficient surface realization using TAG [26].

Evaluating the quality of the implementation, we now want to answer the question whether the constraint parser of the XDK scales up to large-scale parsing. We find a positive answer to this question: we show that the parser can be fine-tuned for parsing the large-scale TAG grammar XTAG [51] such that most of the time, it finds the first parses of a sentence *before* the standard XTAG chart parser. This is surprising given that the XDK constraint parser has worst-case exponential runtime.

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 trees. 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 [9] and our encoding of regular dependency grammars, 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 [40], 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 interesting question was how long the XDK parser would take to find not only the first but the first 1000 parses.

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 [42], and again the percentage of exceeded timeouts.

Interestingly, the XDK parser does not only less often ran into the 30 minute timeout, but was also faster than the LEM parser 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

As a note, in our experiments we did not use the supertagger included in the LEM package, which significantly increases its efficiency at the cost of accuracy [40]. 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. One may be able to remedy this using specialized search techniques for constraint parsing [11].

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. 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)
2. 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
3. Debusmann, R.: A declarative grammar formalism for dependency grammar. Diploma thesis, Saarland University (2001). Http://www.ps.uni-sb.de/Papers/abstracts/da.html
4. Debusmann, R.: Multiword expressions as dependency subgraphs. In: Proceedings of the ACL 2004 Workshop on Multiword Expressions: Integrating Processing. Barcelona/ES (2004)
5. Debusmann, R.: Extensible dependency grammar: A modular grammar formalism based on multigraph description. Ph.D. thesis, Universität des Saarlandes (2006)
6. Debusmann, R.: The complexity of First-Order Extensible Dependency Grammar. Tech. rep., Saarland University (2007)
7. 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)
8. 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)
9. Debusmann, R., Duchier, D., Kuhlmann, M., Thater, S.: TAG as dependency grammar. In: Proceedings of TAG+7. Vancouver/CA (2004)
10. 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)
11. Dienes, P., Koller, A., Kuhlmann, M.: Statistical A* dependency parsing. In: Prospects and Advances in the Syntax/Semantics Interface. Nancy/FR (2003)
12. Duchier, D., Debusmann, R.: Topological dependency trees: A constraint-based account of linear precedence. In: Proceedings of ACL 2001. Toulouse/FR (2001)
13. Egg, M., Koller, A., Niehren, J.: The Constraint Language for Lambda Structures. Journal of Logic, Language, and Information (2001)
14. 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
15. Gaifman, H.: Dependency systems and phrase-structure systems. Information and Control **8**, 304–337 (1965)
16. Gécseg, F., Steinby, M.: Tree languages. In: G. Rozenberg, A. Salomaa (eds.) Handbook of Formal Languages, vol. 3, pp. 1–68. Springer (1997)
17. 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)

18. 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), pp. 608–615. Prague, Czech Republic (2007). URL http://www.aclweb.org/anthology/P/P07/P07-1077.pdf
19. Hays, D.G.: Dependency theory: A formalism and some observations. Language **40**(4), 511–525 (1964). DOI 10.2307/411934
20. 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)
21. 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
22. Hudson, R.A.: English Word Grammar. B. Blackwell, Oxford/UK (1990)
23. 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)
24. 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)
25. 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)
26. Koller, A., Striegnitz, K.: Generation as dependency parsing. In: Proceedings of ACL 2002. Philadelphia/US (2002)
27. Kruijff, G.J.M.: Dependency grammar. In: Encyclopedia of Language and Linguistics, 2nd edn., pp. 444–450. Elsevier (2005)
28. Kuhlmann, M.: Dependency structures and lexicalized grammars. Doctoral dissertation, Saarland University, Saarbrücken, Germany (2007)
29. Kuhlmann, M., Möhl, M.: Mildly context-sensitive dependency languages. In: 45th Annual Meeting of the Association for Computational Linguistics (ACL), pp. 160–167. Prague, Czech Republic (2007). URL http://www.aclweb.org/anthology/P07-1021
30. Kuhlmann, M., Möhl, M.: The string-generative capacity of regular dependency languages. In: Twelfth Conference on Formal Grammar. Dublin, Ireland (2007)
31. 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
32. Marcus, S.: Algebraic Linguistics: Analytical Models, *Mathematics in Science and Engineering*, vol. 29. Academic Press, New York, USA (1967)
33. 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
34. Mel'čuk, I.: Dependency Syntax: Theory and Practice. State Univ. Press of New York, Albany/US (1988)
35. Mozart Consortium: The Mozart-Oz website (2007). Http://www.mozart-oz.org/
36. 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
37. 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)
38. 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

39. 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
40. 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)
41. Schulte, C.: Programming Constraint Services, *Lecture Notes in Artificial Intelligence*, vol. 2302. Springer-Verlag (2002)
42. Schulte, C., Lagerkvist, M., Tack, G.: GECODE—Generic Constraint Development Environment (2007). Http://www.gecode.org/
43. Setz, J.: A principle compiler for Extensible Dependency Grammar. Tech. rep., Saarland University (2007). Bachelorarbeit
44. Sgall, P., Hajicova, E., Panevova, J.: The Meaning of the Sentence in its Semantic and Pragmatic Aspects. D. Reidel, Dordrecht/NL (1986)
45. Shieber, S.M.: Evidence against the context-freeness of natural language. Linguistics and Philosophy **8**(3), 333–343 (1985). DOI 10.1007/BF00630917
46. 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)
47. Tesnière, L.: Éléments de syntaxe structurale. Klinksieck, Paris, France (1959)
48. 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
49. 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
50. 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
51. XTAG Research Group: A Lexicalized Tree Adjoining Grammar for English. Tech. Rep. IRCS-01-03, IRCS, University of Pennsylvania (2001)
52. 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)