Extensible Dependency Grammar: A Modular Grammar Formalism Based On Multigraph Description

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

eingereicht von Ralph Debusmann
Saarbrücken, den 28. April 2006

Verfasser: Ralph Debusmann
Kaiserstraße 197
66133 Saarbrücken
rade@ps.uni-sb.de

Dekan: Prof. Dr. Thorsten Herfet

Prüfungsausschuss: Prof. Dr. Jörg Siekmann
Prof. Dr. Gert Smolka
Prof. Dr. Manfred Pinkal
Dr. Dominik Heckmann

Tag des Kolloquiums: 3.11.2006

Abstract

This thesis develops Extensible Dependency Grammar (XDG), a new grammar formalism combining dependency grammar, model-theoretic syntax, and Jackendoff's parallel grammar architecture. The design of XDG is strongly geared towards modularity: grammars can be modularly extended by any linguistic aspect such as grammatical functions, word order, predicate-argument structure, scope, information structure and prosody, where each aspect is modeled largely independently on a separate dimension. The intersective demands of the dimensions make many complex linguistic phenomena such as extraction in syntax, scope ambiguities in the semantics, and control and raising in the syntax-semantics interface simply fall out as by-products without further stipulation.

This thesis makes three main contributions:

1. The first formalization of XDG as a multigraph description language in higher order logic, and investigations of its expressivity and computational complexity.

2. The first implementation of XDG, the XDG Development Kit (XDK), an extensive grammar development environment built around a constraint parser for XDG.

3. The first application of XDG to natural language, modularly modeling a fragment of English.

Kurzzusammenfassung

Diese Dissertation entwickelt Extensible Dependency Grammar (XDG), einen neuen Grammatikformalismus, der Dependenzgrammatik, modelltheoretische Syntax und die Parallele Grammatik-Architektur von Jackendoff miteinander kombiniert. Das Design von XDG ist vollständig auf Modularität ausgerichtet: Grammatiken können modular durch jeden beliebigen linguistischen Aspekt erweitert werden, z.B. grammatische Funktionen, Wortstellung, Prädikat-Argument Struktur, Skopus, Informationsstruktur und Prosodie, wobei jeder Aspekt größtenteils unabhängig auf einer separaten Dimension modelliert wird. Durch das Zusammenspiel der einzelnen Dimensionen fallen viele complex linguistische Phänomene wie Extraktion in der Syntax, Skopusambiguitäten in der Semantik, und Kontrolle und Anhebung in der Syntax-Semantik-Schnittstelle einfach als Nebenprodukte heraus, ohne dass sie explizit beschrieben werden müssten.

Die Dissertation enthält drei Hauptbeiträge:

1. Die erste Formalisierung von XDG, realisiert als Multigraph-Beschreibungssprache in höherstufiger Logik, und Untersuchungen ihrer Ausdrucksstärke und ihrer computationalen Komplexität.

2. Die erste Implementierung von XDG, das XDG Development Kit (XDK), eine umfangreiche Grammatik-Entwicklungsumgebung, die um einen Constraintparser für XDG herum gebaut ist.

3. Die erste Anwendung von XDG auf natürliche Sprache, die ein Fragments des Englischen auf modulare Art und Weise beschreibt.

Ausführliche Zusammenfassung

In dieser Arbeit entwickeln wir den Grammatikformalismus Extensible Dependency Grammar (XDG) als Kombination von Dependenzgrammatik, modelltheoretischer Syntax and Jackendoffs Paralleler Grammatik-Architektur. Die Kombination ergibt ein neuartiges, radikal modulares Design, das es erlaubt, beliebige linguistische Aspekte zwar innerhalb desselben Formalismus, jedoch weitestgehend unabhängig voneinander auf sogenannten Dimensionen beschreiben zu können. Das erleichtert die Modellierung von linguistischen Phänomenen, da immer nur einzelne Aspekte wie z.B. die grammatischen Funktionen, Wortstellung oder Prädikat-Argument-Struktur, und nicht alle Aspekte gleichzeitig berücksichtigt werden müssen. Zum Beispiel ist Wortstellung im Gegensatz zu den grammatischen Funktionen für die Modellierung der Prädikat-Argument-Struktur meist unerheblich, musste jedoch in bisherigen Ansätzen oft trotzdem bei der Modellierung der Syntax-Semantik-Schnittstelle miteinbezogen werden. In XDG lassen sich beide Aspekte hingegen komplett voneinander abkoppeln. Bei dieser modularen Herangehensweise fallen viele sonst problematische linguistische Phänomene wie Extraktion, Skopusambiguitäten und Kontrolle und Raising dann einfach als Nebenprodukte heraus, ohne dass sie explizit beschrieben werden müssten.

Diese Dissertation leistet drei Beiträge, um zu zeigen, dass XDG nicht nur eine abstrakte Idee ist, sondern auch konkret realisiert werden kann: die erste Formalisierung von XDG als Beschreibungssprache für Multigraphen in höherstufiger Logik, die erste Implementierung von XDG innerhalb eines umfangreichen Grammatikentwicklungssystems, und die erste Anwendung dieses Systems auf natürliche Sprache.

Die Formalisierung von XDG entwickeln wir in Teil I, und zeigen dort, wie sich die Kernkonzepte der Dependenzgrammatik, z.B. Lexikalisierung, Valenz und Ordnung, in XDG realisieren lassen. Das ermöglicht uns dann, erste Untersuchungen der Ausdrucksstärke und der computationalen Komplexität von XDG anzustellen. Wir beweisen, dass XDG mindestens so ausdrucksstark ist wie kontextfreie Grammatik, und zeigen darüber hinaus, dass nichtkontextfreie Sprachen wie $a^n b^n c^n$ oder linguistische Benchmarks wie überkreuzende Dependenzen und Scrambling ebenfalls elegant modelliert werden können. Der Preis für diese Ausdrucksstärke wird im Folgenden sichtbar, wenn wir beweisen, dass das XDG-Erkennungsproblem NP-hart ist.

Trotz dieser hohen Komplexität erzielt der in Teil II dieser Arbeit entwickelte XDG-Constraintparser für kleinere, handgeschriebene Grammatiken erstaunlich gute Ergebnisse. Um den XDG-Parser herum bauen wir die komfortable Grammatikentwicklungsumgebung XDG Development Kit (XDK), die es erlaubt, bequem Grammatiken von Hand oder automatisch zu erstellen und zu testen. Das XDK ist eine unabdingbare Voraussetzung für die Entwicklung der XDG-Grammatiktheorie, und wurde schon mehrfach erfolgreich in der Lehre eingesetzt.

In Teil III entwickeln wir schrittweise eine Grammatik für ein Fragment des Englischen, die sowohl Syntax, Semantik und Phonologie modelliert. Wir zeigen hier konkret, wie komplizierte Phänomene wie Extraktion (u.a. Pied Piping) in der Syntax, Skopus-Ambiguitäten in der Semantik, und Kontrolle und Raising in der Syntax-Semantik-Schnittstelle als Nebenprodukte aus der modularen Beschreibung herausfallen, ohne direkt beschrieben werden zu müssen.

Acknowledgments

First of all, I would like to thank my supervisor Gert Smolka for adopting me as his PhD student in the first place, and for all his suggestions and his patience. I would also like to thank my second supervisor Manfred Pinkal for his suggestions, and for bringing me up as a researcher in the very first place at the department of computational linguistics.

From April 2002 until April 2005, the work on this thesis was funded by the International Post-Graduate College Language Technology and Cognitive Systems, a program of the Deutsche Forschungsgemeinschaft (DFG). I'd like to say thank you to the organizers Matthew Crocker, Sabine Schulte im Walde, Claudia Verburg, my supervisor in Edinburgh, Mark Steedman, and to my colleagues in the IGK and in Edinburgh, including (in alphabetical order) Colin Bannard, Markus Becker, Bettina Braun, Peter Dienes, Amit Dubey, Malte Gabsdil, Ciprian Gerstenberger, Kerstin Hadelich, Dominik Heckmann, Nikiforos Karamanis, Pia Knöferle, Michael Kruppa, Jochen Leidner, Dominika Oliver, Olga Ourioupina, Sebastian and Ulrike Pado, Oana Postolache, Andrew Smith, Tim Smith, Kristina Striegnitz, Maarika Traat and Victor Tron.

From April 2005 until April 2006, this thesis was funded by the DFG as part of CHORUS project Sonderforschungsbereich 378 (SFB 378). In the CHORUS project, I had the opportunity to work together with Alexander Koller, Marco Kuhlmann and Stefan Thater, and earlier with Manuel Bodirsky, Katrin Erk and Markus Egg.

During all these years, except for five months at the department of Informatics at the University of Edinburgh in 2003, my workplace was the Programming Systems Lab (PS-Lab) in Saarbrücken. Two former members of the PS-Lab, Denys Duchier and Joachim Niehren, were, before they left, the main advisers of my work. Denys was the one who laid the foundations for XDG in the late nineties. Joachim was always a fervent supporter of the project, and has supported me in all areas of scientific life. I would also like to thank the other members of the PS-Lab: Mathias Möhl, Andreas Rossberg, Jan Schwinghammer and Guido Tack, the former members Thorsten Brunklaus, Leif Kornstaedt, Didier Le Botlan, Tim Priesnitz, Lutz Straßburger and Gabor Szokoli, and our secretary Ann Van de Veire.

I have also profited from the collaboration with members from the Computational Linguistics Department, including (in alphabetical order) Gerd Fliedner, Valia Kordoni, Christian Korthals, Andrea Kowalski, Christopher-John Rupp and Magdalena Wolska. And Geert-Jan Kruijff, who was always a helpful adviser with all his expertise in dependency grammar.

I had the pleasure to visit Charles University in Prague, where I was invited to by Martin Platek and Vladislav Kubon. Ondrej Bojar then spent half a year in Saarbrücken in return and was a fantastic colleague.

Thanks also for interesting conversations to Cem Bozsahin, Charles Fillmore, Mark Pedersen and Gerold Schneider. And to Jorge Pelizzoni, Ray Jackendoff and Jerry Sadock for illuminating email exchanges about XDG and the parallel grammar architecture.

Finally, meine liebste Simone, without your love, patience and support, without all the nights that you stayed up while I sat writing and re-writing, writing and re-writing again, and without your encouragement as I lost belief in my abilities, I would have never made it. I promise I will never ever write a thesis again.

Contents

1. Introduction

We begin with introducing the background of this thesis, comprising dependency grammar, model-theoretic syntax, the parallel grammar architecture and constraint parsing. Against this background, we set our contributions, before we round off this chapter by a summary of the publications yielded by the thesis, and an overview of its structure.

1.1. Background

1.1.1. Dependency Grammar

According to the structures that they talk about, grammar formalisms for natural language can be divided into two basic classes:

1. *Phrase Structure Grammar* (*PSG*)
2. *Dependency Grammar* (*DG*)

PSG is the approach originally taken by Noam Chomsky (Chomsky 1957, Chomsky 1965), and has also been adopted by the popular grammar formalisms of *Government and Binding* (*GB*) (Chomsky 1981), *Lexical Functional Grammar* (*LFG*) (Bresnan & Kaplan 1982, Bresnan 2001), *Generalized Phrase Structure Grammar* (*GPSG*) (Gazdar, Klein, Pullum & Sag 1985), *Head-driven Phrase Structure Grammar* (*HPSG*) (Pollard & Sag 1987, Pollard & Sag 1994), and *Tree Adjoining Grammar* (*TAG*) (Joshi, Levy & Takahashi 1975, Joshi 1987). A PSG analysis divides a sentence into continuous substrings called *phrases* or *constituents*, which are labeled by *syntactic categories* like S (sentence), NP (noun phrase) and VP (verb phrase). These constituents are then arranged hierarchically in a *phrase structure tree*. Figure 1.1 shows an example phrase structure tree for the sentence *Mary wants to eat spaghetti today*. The root has category S, i.e., is a sentence, which consists of the NP *Mary*, the V (verb) *wants*, the VP *to eat spaghetti* and the Adv (adverb) *today*. The VP in turn consists of the Part (particle) *to*, the V *eat* and the NP *spaghetti*.

DG stands for a different way of analyzing natural language. Its roots can be traced back as far as to Panini's grammar for Hindi (600 BC), the Arabic grammarians of Basra and Kufa in Iraq (800 AD) (Owens 1988), and Latin grammarians (1200 AD). Modern DG is attributed to Tesnière (1959). A DG analysis does not hierarchically arrange substrings but just words, based on the syntactic relations between them called *dependency relations* or *grammatical functions*. A DG analysis is called *dependency graph* or *dependency tree*, in which mothers are called *heads* and daughters *dependents*. Figure 1.2 shows an example dependency tree of *Mary wants to eat spaghetti today*. Here, each node (circle) is identified with a word in the

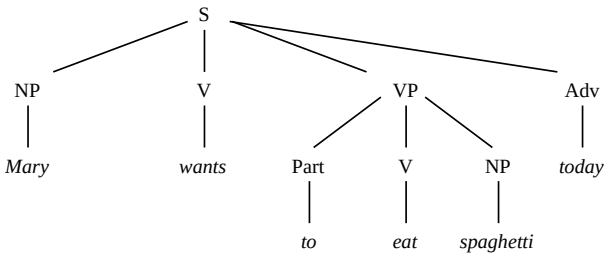


Figure 1.1.: Example phrase-structure analysis

sentence (as indicated by the dotted vertical lines called *projection edges*), and the tree edges are drawn as solid lines interrupted by edge labels which reflect the grammatical functions: *Mary* is the subject (edge label subj), *eat* is the infinitival complement (vinf), and *today* the adverbial modifier (adv) of *wants*. In turn, *to* is a particle (part), and *spaghetti* the object (obj) of *eat*.

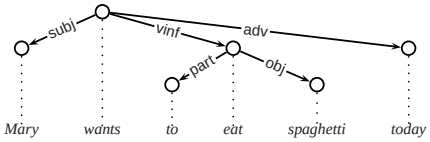


Figure 1.2.: Example dependency tree

In a phrase structure tree, only continuous substrings can be arranged. This restriction poses problems for the analysis of word order variation, even for *rigid word order languages* (Steele 1978) such as English, which exhibits e.g. the discontinuous syntactic phenomena of wh-questions and topicalization. An example for the latter is the sentence *Spaghetti, Mary wants to eat today*, where the object *spaghetti* of *eat* has been dislocated to the very left. The result is the discontinuous VP constituent *spaghetti to eat*, which has a gap between *spaghetti* and *to*, comprising the words *Mary* and *wants*. This is shown in the impossible phrase structure of Figure 1.3. Such sentences can only be analyzed in PSG by either changing the analysis, by which the connection between the verb *eat* and its object *spaghetti* is lost, or by extending the grammar formalism with additional mechanisms like traces (GB) or feature percolation (GPSG, HPSG).

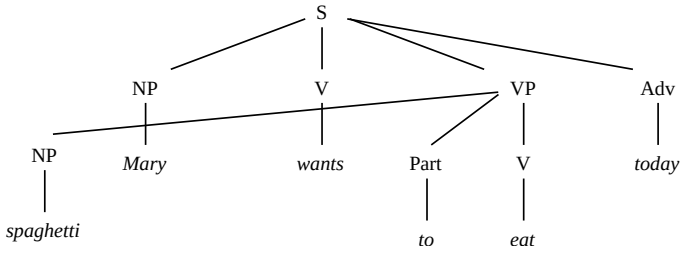


Figure 1.3.: (Impossible) discontinuous phrase structure analysis

For DG, discontinuous constructions can be represented straightforwardly: as we have already emphasized, the analyses are based on words and not substrings. In fact, Figure 1.4 shows a perfectly acceptable dependency tree for *Spaghetti, Mary wants to eat today*.

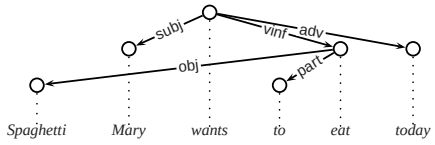


Figure 1.4.: Example discontinuous (non-projective) dependency tree

For some applications however, it is desirable to define a restriction analogous to that of the continuity of substrings in PSG also for DG. Here, the idea is to forbid that any edge crosses a projection edge of a node higher up or to the side in the tree, such as the edge from *eat* to *spaghetti* in Figure 1.4, which crosses the projection edges of the nodes corresponding to *Mary* and *wants*. Analyses without crossing edges are then called *projective*, and those which include them *non-projective*. The crucial advantage of DG is now that the projectivity restriction is optional, whereas the continuity restriction of PSG is obligatory. This crucial difference was overseen in early formalizations of DG (Gross 1964, Hays 1964, Gaifman 1965), where it was proven to be equivalent to *Context-Free Grammar* (*CFG*) in general, even though this is only true given the projectivity restriction.

Theoretically, a DG analysis need not be ordered at all. This grants DG the flexibility of not being confined to model the syntax of natural language alone—dependency analyses can also be used to model e.g. the semantics, where order is irrelevant. This is used for example in the traditional DG frameworks of *Functional Generative Description* (*FGD*) (Sgall, Hajicova & Panevova 1986) and *Meaning Text Theory* (*MTT*) (Mel'čuk 1988) to model *predicate-argument structure*. Figure 1.5 shows an example. Here, the edge labels are *thematic roles* (Panenová 1974), (Dowty 1989) instead of grammatical functions. The word *wants* is the theme (edge label th) of *today*, and has itself the agent (ag) *Mary* and the theme *eat*. The word *eat* has the agent *Mary* and the patient (pat) *spaghetti*.

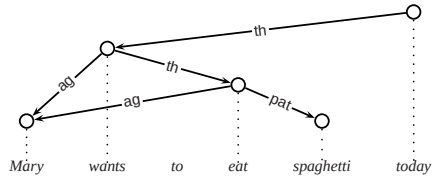


Figure 1.5.: Example semantic dependency graph

How do PSG and DG *grammars* look like? In PSG, a grammar is traditionally made up of production rules (rewriting rules) such as the one below, which rewrites category S into an NP, a V, a VP and an Adv:

$$\text{S} \rightarrow \text{NP V VP Adv} \tag{1.1}$$

In DG, grammars are traditionally expressed in terms of *valency*. The term is taken from chemistry, where valency specifies for each atom the number of electrons which it will give,

take, or share to form a chemical compound. In DG, valency specifies for each node the required incoming edges (*in valency*) and outgoing edges (*out valency*).[1] For instance, the verb *eat* is an infinitive, and requires a particle and an object. This is reflected in its in and out valencies: its in valency licenses at most one incoming edge labeled vinf, and its out valency requires precisely one outgoing edge labeled part, and one labeled obj. No other incoming and outgoing edges are licensed. As DG is word-based, valencies are typically expressed in a *lexicon* of *lexical entries*. For example, the lexical entry below specifies the in and out valencies of the word *eat*, where the question mark represents optionality, and the exclamation mark obligation:

$$\left\{ \begin{array}{rl} word & = eat \\ in & = \{\mathsf{vinf?}\} \\ out & = \{\mathsf{part!}, \mathsf{obj!}\} \end{array} \right\} \tag{1.2}$$

Even though most ideas of DG (heads/dependents, valency, lexicalization) have been gradually adopted by most grammar formalisms, including GB, LFG, GPSG, HPSG, TAG and also *Combinatory Categorial Grammar* (*CCG*) (Steedman 2000*b*), none of the frameworks directly based on DG have really got into the mainstream, be it FGD, MTT, *Abhängigkeitsgrammatik* (Kunze 1975), *Word Grammar* (*WG*), (Hudson 1990), or the more recent frameworks of *Constraint Dependency Grammar* (*CDG*) (Menzel & Schröder 1998), *Free Order Dependency Grammar* (*FODG*) (Holan, Kubon, Oliva & Platek 2000), Bröker's (1999) approach, *Topological Dependency Grammar* (*TDG*) (Duchier & Debusmann 2001), and Gerdes & Kahane's (2001) approach also called Topological Dependency Grammar. The reasons for this are manifold:

- None of the frameworks is completely logically formalized, although there are partial formalizations of e.g. MTT (Kahane 2001).
- Although word order variation can be perfectly represented in DG, the frameworks have for a long time lacked a declarative and workable account of word order. This defect has only recently been addressed, cf. (Bröker 1999), (Duchier & Debusmann 2001) and (Gerdes & Kahane 2001), but only in frameworks that are confined to syntax.
- They lack a syntax-semantics interface to a deep semantics, i.e., a semantics beyond predicate-argument structure, including the handling of quantifier scope.

1.1.2. Model-Theoretic Syntax

Grammar formalisms cannot only be distinguished as to the structures that they talk about, but also with respect to the perspective they take on them. Following (Pullum & Scholz 2001), we distinguish two perspectives:

1. *Generate-Enumerative Syntax* (*GES*)
2. *Model-Theoretic Syntax* (*MTS*)

[1] Traditionally (Peirce 1898), valency only refers to the outgoing edges. Following e.g. (Duchier & Debusmann 2001), we generalize it to encompass also incoming edges.

GES originates in Chomsky's original approach (Chomsky 1957) and *Categorial Grammar* (Ajdukiewicz 1935, Bar-Hillel 1953). In GES, a grammar of language *L* is a device for recursively enumerating sets of expressions, based on production rules or inference rules. An expression *E* is grammatical according to a grammar *G* if and only if *E* is derivable in *G*. That is, GES takes a *syntactic* or *proof-theoretic* perspective on grammar by asking the question how expressions can be derived from other expressions. A production rule such as (1.1) above is interpreted as "from category S, we can derive the concatenation of the categories NP, V, VP and Adv".

McCawley (1968) had the insight to instead take a *semantic* or *model-theoretic* perspective, and interpret the rule with respect to the models, i.e., the phrase structure trees that it licenses. From this perspective, (1.1) is the description of a local tree rooted in S, having the daughters NP, V, VP and Adv, in this order. Using a term coined by Rogers (1996), we call this perspective *Model-Theoretic Syntax* (*MTS*). In MTS, a grammar of language *L* is a logical description of the well-formed models of *L*, and an expression *E* is grammatical according to a grammar *G* if and only if *E* is a model of *G*.

Of the grammar formalisms mentioned above, the pure GES perspective is only taken by TAG, CCG, and the dependency-based FGD and MTT. The other frameworks (GB, LFG, GPSG and HPSG) can be regarded as hybrids: they all have a generative backbone based on PSG which generates a large set of structures, from which the ill-formed structures are then filtered out by constraints.

Compared to GES, MTS is clearly more declarative: it fully abstracts away from any underlying mechanisms, and can thus offer a clarifying perspective. This allows for better comparisons between grammar formalisms. Combined with a syntax-semantics interface, MTS also has the potential for reversibility, i.e, the same MTS grammar can be used for parsing and generation. These advantages have yielded a considerable body of work devoted to the reformulation of GES and hybrid GES/MTS frameworks into pure MTS frameworks (Blackburn & Gardent 1995, Rogers 1996, Rogers 1998).

1.1.3. Parallel Grammar Architecture

Traditionally, grammar formalisms have not only adopted the perspective of GES, but also, consequently, a *syntacto-centric architecture*: only the well-formedness conditions of syntax are independently specified, and all other linguistic structures such as semantics are derived from it via functional, directed interfaces. We depict this architecture in Figure 1.6, where the slanted arrow entering the Syntax bubble represents the well-formedness conditions of syntax, and the curved directed arrows from Syntax to Phonology and to Semantics represent the corresponding functional interfaces. Typical instances of this architecture are GB, TAG and CCG, and also FGD and MTT.

With the advent of the perspective of MTS, and inspired by *Autosegmental Phonology* (Goldsmith 1979, Goldsmith 1990), the syntacto-centric architecture has recently been challenged by Sadock's (1991) *Autolexical Syntax*, Valin & LaPolla's (1997) *Role and Reference Grammar* (*RRG*) and Jackendoff's (2002) approach, all of which propose a *parallel grammar architecture*. Here, all linguistic structures, not only syntax, are promoted to the status of autonomous modules, which are determined independently by their own well-formedness

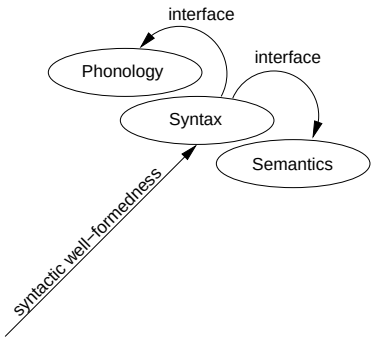


Figure 1.6.: Syntacto-centric grammar architecture

conditions. The modules co-constrain each other through relational, bi-directional, instead of functional, directed interfaces. Figure 1.7 depicts this architecture. Here, the three slanted arrows entering the Phonology, Syntax and Semantics bubbles represent the independent well-formedness conditions of these modules, and the curved bi-directional arrows between them the relational interfaces.

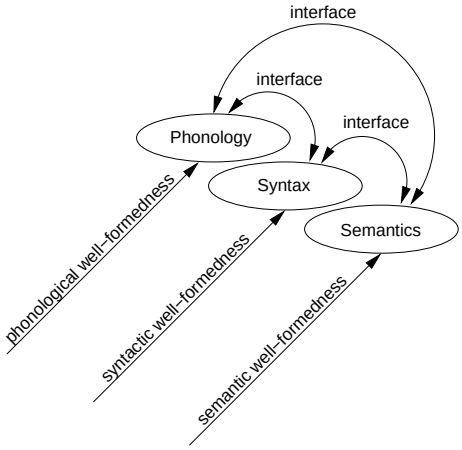


Figure 1.7.: Parallel grammar architecture

The parallel grammar architecture is clearly more modular than the syntacto-centric one: the linguistic modules can be developed separately, and be reused more easily. For instance, the same semantic module could be reused for a number of languages which differ only in their syntax. In addition, the parallel grammar architecture gives rise to what we call *emergence*: many complex phenomena simply *emerge* from the interaction of the individually simple modules, without further stipulation. This is because the burden of analysis is not carried by syntax alone, but is instead shared with the other linguistic modules.

However, the parallel grammar architecture has not yet been put into practice, except for a tiny fragment described in (Higgins 1998). As Jackendoff (2002) notes, the parallel grammar architecture presupposes a model-theoretic approach and could thus in principle be implemented in LFG and HPSG. In their practical realizations, however, both LFG and HPSG are applied in syntacto-centrically: the semantics, e.g. *Glue Semantics* (Dalrymple, Lamp-

ing, Pereira & Saraswat 1995) in LFG and *Minimal Recursion Semantics* (*MRS*) (Copestake, Flickinger, Pollard & Sag 2004) in HPSG, are still derived from syntax, and not granted the status of independent modules.

1.1.4. Constraint Parsing

In computational linguistics, parsing is usually done using context-free *chart parsing* (Earley 1970, Kay 1980) or extensions thereof, e.g. for TAG (Sarkar 2000). Chart parsing is an application of dynamic programming and uses a data structure called chart to memorize already parsed subtrees. This removes the need for backtracking and prevents combinatorial explosion.

An alternative approach is *constraint parsing* (Maruyama 1990, Duchier 1999), where parsing is viewed as finding the solutions of a *Constraint Satisfaction Problem* (*CSP*) using *Constraint Programming* (*CP*) (Jaffar & Lassez 1988, Jaffar & Maher 1994, Hentenryck & Saraswat 1996, Schulte 2002, Apt 2003). Constraint programming is the study of computational systems based on constraints, where constraints are precisely specifiable relations among several unknowns called *constraint variables*. Work in this area can be traced back to research in artificial intelligence and computer graphics in the 1960s and 1970s (Sutherland 1963, Montanari 1970, Waltz 1975); Wallace (1996) gives an overview of the practical applications of constraint programming, e.g. in artificial intelligence (reasoning, abduction, planning, scheduling, resource allocation and configuration), in the context of databases, user interfaces, operations research, robotics and control theory. In CP, the search for solutions is determined by two processes: *propagation* and *distribution*. Propagation is the application of deterministic inference rules to narrow down the search space, and distribution corresponds to non-deterministic choice. Both processes are interleaved: distribution ensues whenever the information accumulated by propagation is not sufficient for further disambiguation, and propagation ensues again after each distribution step. This paradigm is called *propagate and distribute*, and contrasts with the naive *generate and test* paradigm, where every candidate solution must be generated before it can be tested, rapidly leading into a combinatorial explosion.

As ambiguity is prevalent in parsing, parsers based on CP can greatly benefit from constraint propagation in order to narrow down the search space. Maruyama (1990) was the first to propose a treatment of dependency grammar using CP, and described parsing as a process of incremental disambiguation. Harper, Hockema & White (1999) continued this line of research by proposing several algorithmic improvements, and Menzel (1998), Heinecke, Kunze, Menzel & Schröder (1998) and Menzel & Schröder (1998) proposed the use of soft, graded constraints for robustness. Duchier (1999) developed an account of dependency parsing using concurrent constraint programming (Saraswat 1993) in *Mozart/Oz* (Smolka 1995, Mozart Consortium 2006), where computation is viewed as arising from the activities of concurrently operating agents that communicate via a shared set of constraint variables. Duchier's approach made use of the unique combination of finite set constraints and encapsulated speculative computations in the form of *deep guards* (Schulte 2002) only found in Mozart/Oz.

Constraint parsing has a number of advantages. Firstly, it is not tied to word order and continuity of constituents: it is indeed perfectly possible to do constraint parsing without taking

word order into account at all. This makes it ideal for the implementation of parsers for dependency grammar. Secondly, constraint parsing is perfectly suited for the implementation of grammar formalisms based on MTS and the parallel grammar architecture, as they guarantee both the reversibility of MTS approaches, and concurrency, i.e., the ability to simultaneously process multiple levels of representation. However, compared to chart parsing, constraint parsing is less efficient.

1.2. Contributions and Structure of the Thesis

In this thesis, we make three main contributions. The first is a combination of the paradigms of dependency grammar, MTS and the parallel architecture, resulting in the grammar formalism of *Extensible Dependency Grammar* (*XDG*), which we formalize as a multigraph description language in higher order logic. The second is an implementation of a constraint parser for XDG within an extensive grammar development environment, the *XDG Development Kit* (*XDK*) (Debusmann & Duchier 2006). The third is an application of XDG to natural language, modeling a fragment of English syntax, semantics and phonology. The presentation of these contributions is preceded by a first overview of XDG and the XDK in chapter 2, and followed by a summary and an outlook in chapter 13.

Part I develops the first formalization of XDG as a multigraph description language in higher order logic (chapter 3). This brings us the position to recast the key concepts of dependency grammar, including lexicalization, valency and order, as principles on multigraphs (chapter 4). We then investigate the expressivity of XDG in chapter 5, and its computational complexity in chapter 6.

Part II develops the XDG Development Kit (XDK) (chapter 7), which is centered around a constraint parser based on the dependency parser introduced in (Duchier 1999, Duchier 2003), (chapter 8). The XDK includes the statically typed *XDK description language*, which serves mainly as a *metagrammar* (Duchier, Le Roux & Parmentier 2004, Crabbé & Duchier 2004) for convenient grammar development, a comprehensive Graphical User Interface (GUI) (cf. Figure 1.8), and extensive documentation (more than 200 pages). The XDK spans 35000 lines of Mozart/Oz code, and comes with example handcrafted grammars for Arabic, Czech, Dutch, English, French and German, which span an additional 24000 lines.

Part III applies XDG to model a fragment of the syntax, semantics and phonology of English. The grammar subdivides the linguistic modules of phonology, syntax and semantics: within syntax (chapter 9), we make use of the declarative account of word order introduced by *Topological Dependency Grammar* (*TDG*) (Duchier & Debusmann 2001): we distinguish the two dimensions of *Immediate Dominance* (ID) and *Linear Precedence* (LP), where the ID dimension models grammatical functions, and the LP dimension word order. Within semantics (chapter 10), we distinguish the *Predicate-Argument structure* (PA) to model predicate logical functor-argument relationships, the *SCope structure* (SC) to model quantifier scope, and *Information Structure* (IS) to model the theme/rheme and focus/background relationships.[2] Phonology (chapter 11) contains only the *Prosodic Structure* (PS).[3] The syntax-semantics interface

[2] We follow (Jackendoff 2002) in associating information structure with semantics and not with pragmatics.

[3] This thesis does not include a thorough treatment of phonology. For this, we would need many more structures,

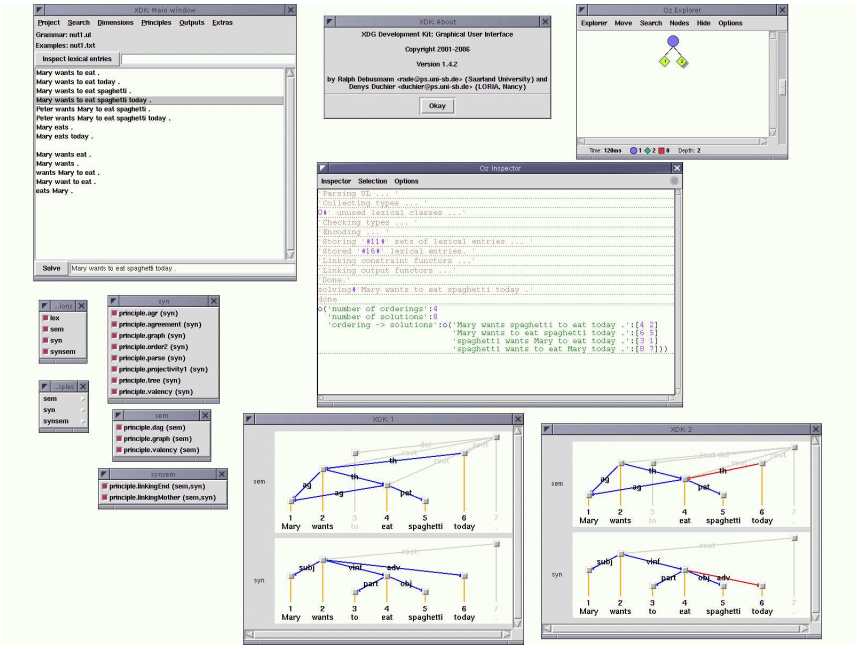


Figure 1.8.: The XDK GUI (`xdk`)

(chapter 12) is relational, supports underspecification, and has an interface to the *Constraint Language for Lambda Structures* (*CLLS*) (Egg, Koller & Niehren 2001). The *phonology-semantics interface* (also chapter 12) is a modular adaptation of Steedman's (2000*a*) prosodic account of information structure. We depict the complete architecture in Figure 1.9.

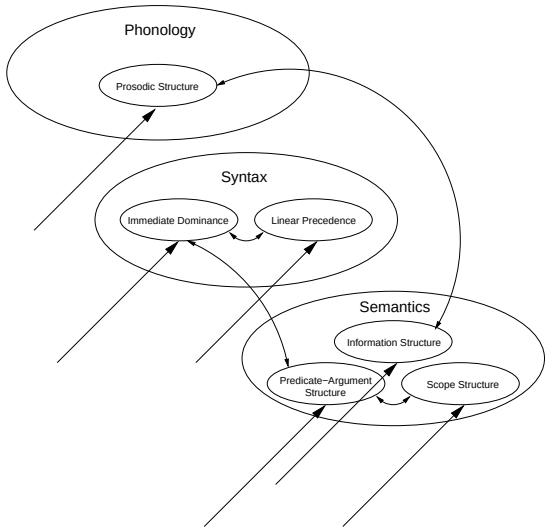


Figure 1.9.: The XDG grammar architecture in this thesis

cf. the tiers in Autosegmental Phonology (Goldsmith 1979, Goldsmith 1990). We include the prosodic dimension for two reasons: 1) to support the realization of Steedman's (2000*a*) prosodic account of information structure, and 2) to be able to more clearly illustrate the benefits of the parallel grammar architecture.

1.3. Publications

This section lists the publications resulting from the research for this thesis. The first papers are centered around the XDG grammar formalism and its formalization:

- Ralph Debusmann (2003), *Dependency Grammar as Graph Description*, Workshop: Prospects and Advances in the Syntax-Semantics Interface, Nancy/FR
- Ralph Debusmann, Denys Duchier, Marco Kuhlmann and Stefan Thater (2004), *TAG Parsing as Model Enumeration*, 7th International Workshop on Tree Adjoining Grammar and Related Formalisms, TAG+7, Vancouver/CN
- Ralph Debusmann, Denys Duchier and Geert-Jan Kruijff (2004), *Extensible Dependency Grammar: A New Methodology*, The 20th International Conference on Computational Linguistics, COLING 2004, Workshop: Recent Advances in Dependency Grammar, Geneva/CH
- Ralph Debusmann, Denys Duchier and Marco Kuhlmann (2004), *Multi-dimensional Graph Configuration for Natural Language Processing*, International Workshop on Constraint Solving and Language Processing, Roskilde/DK
- Ralph Debusmann, Denys Duchier and Andreas Rossberg (2005), *Modular Grammar Design with Typed Parametric Principles*, The 10th Conference on Formal Grammar and The 9th Meeting on Mathematics of Language, FG-MOL 2005, Edinburgh/UK
- Ralph Debusmann and Gert Smolka (2006), *Multi-dimensional Dependency Grammar as Multigraph Description*, The 19th International FLAIRS Conference, FLAIRS 2006, Melbourne Beach/US

The XDG Development Kit is published in:

- Ralph Debusmann (2003), *A Parser System for Extensible Dependency Grammar*, Workshop: Prospects and Advances in the Syntax-Semantics Interface, Nancy/FR
- Ralph Debusmann, Denys Duchier and Joachim Niehren (2004), *The XDG Grammar Development Kit*, Second International Mozart/Oz Conference, MOZ 2004, Charleroi/BE

The following papers describe the modeling of natural language, including the interfaces from syntax to semantics and from phonology to semantics:

- Christian Korthals and Ralph Debusmann (2002), *Linking syntactic and semantic arguments in a dependency-based formalism*, The 19th International Conference on Computational Linguistics, COLING 2002, Taipei/TW
- Alexander Koller, Ralph Debusmann, Malte Gabsdil and Kristina Striegnitz (2004), *Put my galakmid coin into the dispenser and kick it: Computational Linguistics and Theorem Proving in a Computer Game*, Journal of Logic, Language And Information

- Ralph Debusmann, Denys Duchier, Alexander Koller, Marco Kuhlmann, Gert Smolka and Stefan Thater (2004), *A Relational Syntax-Semantics Interface Based on Dependency Grammar*, The 20th International Conference on Computational Linguistics, COLING 2004, Geneva/CH

- Ralph Debusmann (2004), *Multiword expressions as dependency subgraphs*, 42nd Annual Meeting of the Association for Computational Linguistics, ACL 2004, Workshop: Multiword Expressions: Integrating Processing, Barcelona/ES

- Ralph Debusmann, Oana Postolache and Maarika Traat 2005, *A Modular Account of Information Structure in Extensible Dependency Grammar*, 6th International Conference on Intelligent Text Processing and Computational Linguistics, CICLING 2005, Mexico City/MX

2. XDG in a Nutshell

This chapter gives a walkthrough of the main concepts of Extensible Dependency Grammar XDG: we introduce the models of XDG, explain how to write grammars and how to implement them using the XDG Development Kit (XDK). Then, we compare XDG with a number of existing grammar formalisms.

2.1. XDG Models

We first introduce the specific form of dependency graphs used in XDG. Then, we define the models of XDG, which are tuples of dependency graphs sharing the same set of nodes called *multigraphs*.

2.1.1. Dependency Graphs

Dependency graphs in XDG (cf. the example in Figure 2.1) are a specific form of dependency graphs having the following properties:

1. Each node (round circle) is associated with an index (1, 2, 3 etc.) indicating its position. The connection is made explicit by the dotted vertical lines called projection edges.

2. Each node is associated with a word (*Mary*, *wants*, *to* etc.), which we write below its index.[1]

3. Each node is associated with *attributes* arranged in attribute-value-matrices which we call *records*. Attributes incorporate lexical information (in the *lex* subrecord) and non-lexical information (outside the *lex* subrecord). We draw the attributes of the nodes below their associated words. In Figure 2.1, we have drawn the attributes schematically because of lack of space, and have highlighted only those of nodes 1 and 2 by magnification. The attributes include the *lexical attributes in* and *out* describing the in valencies and out valencies of the node (cf. section 1.1.1). For example, the in valency of node 2 is $\{\mathsf{root?}\}$, where the *cardinality* ? stands for "at most one", i.e., there must be at most one incoming edge labeled root, and no other incoming edges are licensed. The out valency of node 2 is $\{\mathsf{subj!}, \mathsf{vinf!}, \mathsf{adv}*\}$, where the cardinality ! stands for "precisely one" and the $*$ for "arbitrary many". *order* is a set of pairs describing a strict partial order on the dependents and the head (signified by the special *anchor label* $\uparrow$) with respect to its

[1] In this and the subsequent analyses of natural language sentences, we assume that end-of-sentence markers such as the full stop (node 7) form the root of the dependency graph to ease the modeling of non-syntactic linguistic aspects, e.g. predicate-argument structure and information structure.

dependents. For example, for node 2, the subject of the head must precede it and also its infinitival complement. *agrs* describes a set of possible *agreement tuples* consisting of person and number, and *agree* the set of edge labels of dependents with which the node must agree. In the example, the finite verb *wants* can only have third person singular agreement ($agrs = \{(3, sg)\}$), and must agree with its subject *Mary* ($agree = \{\mathsf{subj}\}$). *agr* is a *non-lexical attribute* representing the agreement tuple assigned to the node, picked out from the *agrs* set.

4. The nodes are connected to each other by labeled and directed edges. In the example, there is an edge from node 7 to node 2 labeled root to express that *wants*, the finite verb, is the root of the analysis. There are also edges from node 2 to node 1 (labeled subj), from 2 to 4 (labeled vinf), and from 2 to 6 (labeled adv), which express that *Mary* is the subject, *eat* the infinitival complement, and *today* the adverbial modifier of *wants*. The edges from node 4 to nodes 3 and 5 labeled part and obj express that *to* is a particle of *eat* and *spaghetti* its object.

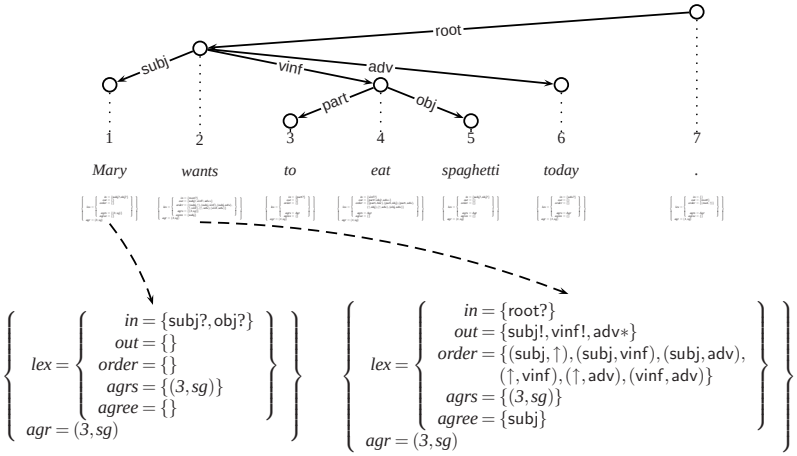


Figure 2.1.: Dependency Graph (syntactic analysis)

As we have already mentioned in section 1.1.1, dependency graphs are not restricted to describing syntactic structures. In fact, in XDG, they do not even have to be trees but can be any kind of directed graph. Figure 2.2, for example, shows a *Directed Acyclic Graph* (*DAG*) describing the predicate-argument structure of the example sentence, where the edge labels are thematic roles.[2] Here, the additional root node corresponding to the end-of-sentence marker helps us to distinguish nodes which correspond to semantic predicates, which we take to be the "roots" of the analysis, and nodes without semantic content, which take to be "deleted". Roots are connected to the end-of-sentence marker by edges labeled root, and deleted nodes

[2]This structure does not provide us with all the information required for a complete semantic representation, but only with the relations between predicates (e.g. verbs like *eat*) and their arguments (e.g. nouns like *Mary* and *spaghetti*). What is missing to build e.g. a representation of the semantics in predicate logic is the modeling of quantification, which we omit in this chapter for simplicity. We will pick up this issue again in chapter 10, where we also provide a means of modeling quantification using an additional dependency graph.

by edges labeled del. In the example, *wants*, *eat* and *today* are semantic predicates. *wants* is additionally the theme (th) of the adverb *today*, and has in turn the agent (ag) *Mary* and the theme *eat*. *eat* has *Mary* as its agent, too, and the patient (pat) *spaghetti*. The particle *to* (node 3) has no semantic content and can thus be "deleted" from the semantic analysis. The attributes of the nodes are the lexical attributes *in* and *out*, standing for the in and out valency of the nodes (cf. section 1.1.1), respectively.

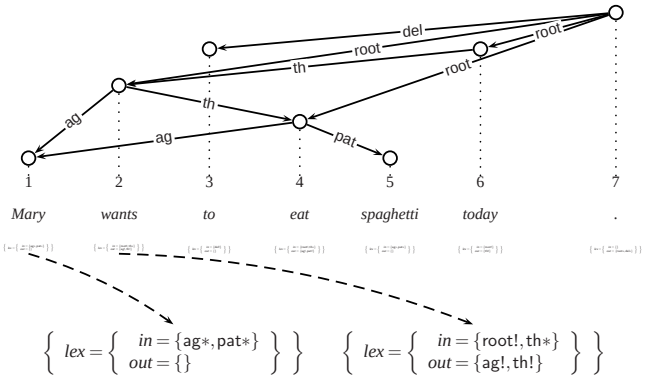


Figure 2.2.: Dependency Graph (semantic analysis)

XDG also supports dependency graphs without edges. The purpose of such graphs is to carry attributes which do not fit properly on any of the other dependency graphs. These are typically attributes which specify the interface between dimensions. For example, in Figure 2.3 the attributes describe the realization of semantic arguments like agent and theme by grammatical functions like subject and infinitival complement (ag = {subj} and th = {vinf}).

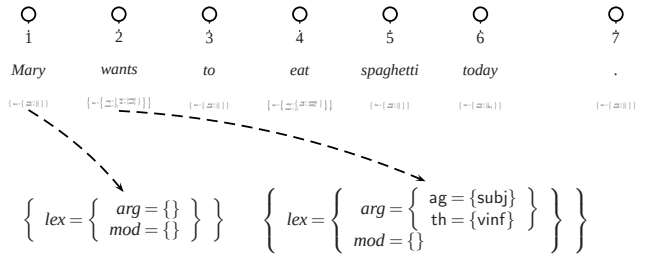


Figure 2.3.: Dependency Graph (syntax-semantics analysis)

2.1.2. Multigraphs

The models of XDG are tuples of dependency graphs. The component dependency graphs are called *dimensions*, which must all share the same set of nodes. Because of that, the tuples can be regarded as *multigraphs*, i.e., graphs with multiple edges between the nodes from graph theory (Harary 1994). In fact, this is how we will call the models of XDG for the remainder of the thesis.

We show an example multigraph in Figure 2.4. It consists of three dimensions which we call SYN (syntax), SEM (semantics) and SYNSEM (syntax-semantics interface). For clarity, we

draw the three dimensions as individual dependency graphs (cf. Figure 2.1, Figure 2.2 and Figure 2.3), and indicate the node sharing by arranging shared nodes in the same columns. The multigraph describes at the same time the syntactic and semantic analysis of the sentence, and expresses e.g. that *Mary* (node 1), the subject of *wants* on SYN, is the realization of the agent of both *wants* and *eat* on SEM. The SYNSEM dimension carries the attributes needed for the syntax-semantics interface, e.g. specifying how semantic arguments are realized by grammatical functions.

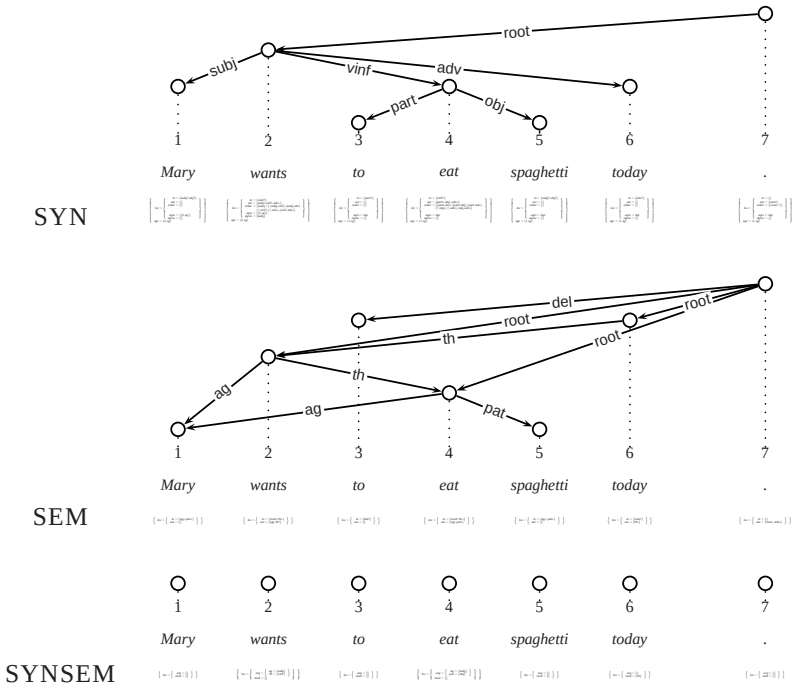


Figure 2.4.: Multigraph (simultaneous syntactic and semantic analysis)

For clarity, we will in the following abbreviate multigraphs adopting the following conventions:

- we omit all dimensions without edges
- we omit the attributes of the nodes
- we "ghost" the node corresponding to the end-of-sentence marker (i.e., we draw it in gray instead of black) and all deleted nodes (i.e., whose incoming edge labels include del)
- we "ghost" all edges labeled root or del

We display an example in Figure 2.5, which is a "ghosted" version of Figure 2.4.

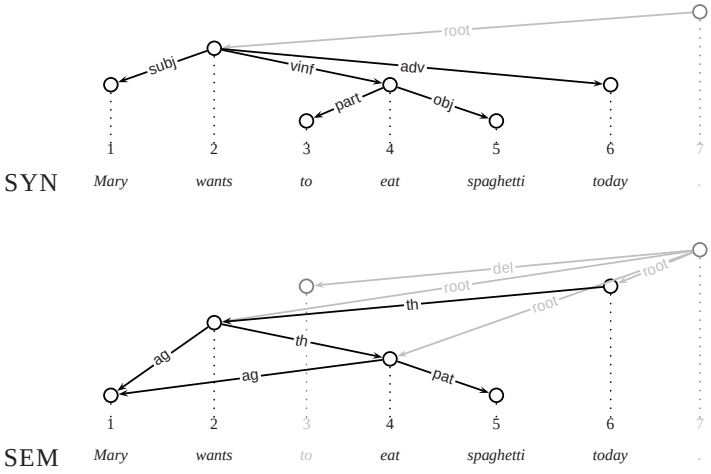


Figure 2.5.: Abbreviated Multigraph

2.2. XDG Grammars

The models of XDG, multigraphs, are described by grammars. An XDG grammar is defined by:

1. defining the dimensions
2. defining the principles
3. defining the lexicon

2.2.1. Dimensions

Each *dimension* is associated with a unique name (e.g. SYN), a set of edge labels and a set of attributes. The latter will in this thesis always be characterized by a record type.

2.2.2. Principles

The *principles* state the well-formedness conditions of the XDG models. New principles can be freely written, but usually, the grammar writer will only need to pick out a subset of the predefined principles such as the *Tree principle* (to state that the models of a dimension must be trees) and the *Valency principle* (to constrain the in and out valencies of the nodes). The set of predefined principles is already sufficient to model an interesting fragment of the syntax, semantics and even phonology of English, as we will demonstrate in part III. The principles have also been successfully employed for modeling fragments of Arabic (Odeh 2004), Czech, Dutch (Debusmann & Duchier 2002), French, and German (Debusmann 2001, Bader, Foeldesi, Pfeiffer & Steigner 2004).

2.2.3. Lexicon

The *lexicon* is a set of records called *lexical entries*. Each lexical entry is indexed by a word called its *anchor*, and simultaneously specifies the attributes of all dimensions and thereby synchronizes them. For example, the lexical entry below has the anchor *wants* and specifies the *in* and *out* attributes of the dimensions SYN and SEM:

$$\left\{\begin{array}{l} \mathit{word} = \mathit{wants} \\ \textsc{syn} = \left\{\begin{array}{l} \mathit{in} = \{\mathsf{root?}\} \\ \mathit{out} = \{\mathsf{subj!}, \mathsf{vinf!}, \mathsf{adv}*\} \end{array}\right\} \\ \textsc{sem} = \left\{\begin{array}{l} \mathit{in} = \{\mathsf{root!}, \mathsf{th}*\} \\ \mathit{out} = \{\mathsf{ag!}, \mathsf{th!}\} \end{array}\right\} \end{array}\right\} \tag{2.1}$$

2.2.4. Example Grammar

We present a first example grammar formulated over the three dimensions SYN, SEM and SYNSEM.

Dimensions. We begin the definition of the grammar with the definition of the SYN dimension. We define SYN given a type $\mathit{Agr} = \mathit{tuple}(\{1,2,3\},\{\mathit{sg},\mathit{pl}\})$ of agreement tuples consisting of a person (*1*, *2* or *3*) and a number (*sg* for "singular" and *pl* for "plural").

- The edge labels $L_{\textsc{syn}}$ of SYN are:

$$\{\mathsf{root}, \mathsf{subj}, \mathsf{part}, \mathsf{obj}, \mathsf{vinf}, \mathsf{adv}\} \tag{2.2}$$

  where root stands for the root of the analysis, subj for subject, part for particle, obj for object, vinf for infinitival complement, and adv for adverb.

- The attributes on SYN are defined by the following record type:

$$\left\{\begin{array}{l} \mathit{lex} : \left\{\begin{array}{r} \mathit{in} : \mathit{valency}(L_{\textsc{syn}}) \\ \mathit{out} : \mathit{valency}(L_{\textsc{syn}}) \\ \mathit{order} : \mathit{set}(\mathit{tuple}(L_{\textsc{syn}} \mid \{\uparrow\}, L_{\textsc{syn}} \mid \{\uparrow\})) \\ \mathit{agrs} : \mathit{set}(\mathit{Agr}) \\ \mathit{agree} : \mathit{set}(L_{\textsc{syn}}) \end{array}\right\} \\ \mathit{agr} : \mathit{Agr} \end{array}\right\} \tag{2.3}$$

  where the attributes in the *lex* subrecord are called *lexical attributes* since they will be determined by the lexicon. The lexical attributes *in* and *out* are valencies specifying the licensed incoming and outgoing edges, i.e., mappings from edge labels on SYN to cardinalities (!, ? or $*$). *order* specifies a strict partial order on the dependents and on the anchor $\uparrow$ with respect to its dependents.[3] *agrs* specifies the licensed agreement tuples for the node, and *agree* the set of dependents with which it must agree. The non-lexical attribute *agr* stands for the one agreement tuple out of the licensed agreement tuples which is picked out by the node in each analysis.

[3] Here, for two domains T and T', we write $T \mid T'$ for the union of T and T'.

On the SEM dimension, the set of edge labels and the attributes are:

- edge labels:

$$\{\mathsf{root}, \mathsf{del}, \mathsf{ag}, \mathsf{pat}, \mathsf{th}\} \tag{2.4}$$

  where root, standing for the roots of the analysis, and del for deleted nodes are used to connect roots and deleted nodes to the additional root of the analysis, and ag, pat and th are thematic roles.

- attributes:

$$\left\{ \mathit{lex} : \left\{ \begin{array}{c} \mathit{in} : \mathit{valency}(L_{\mathrm{SEM}}) \\ \mathit{out} : \mathit{valency}(L_{\mathrm{SEM}}) \end{array} \right\} \right\} \tag{2.5}$$

Finally, on the SYNSEM dimension, the set of edge labels is empty since its models are graphs without edges. The attributes on SYNSEM are:

$$\left\{ \mathit{lex} : \left\{ \begin{array}{c} \mathit{arg} : \mathit{vec}(L_{\mathrm{SEM}}, \mathit{set}(L_{\mathrm{SYN}})) \\ \mathit{mod} : \mathit{set}(L_{\mathrm{SEM}}) \end{array} \right\} \right\} \tag{2.6}$$

arg is a *vector* used to map SEM edge labels to sets of SYN edge labels to constrain the realization of the semantic arguments of verbs, such as agents, by grammatical functions such as subjects. *mod* is a set of SEM edge labels to constrain the realization of the semantic arguments of adverbs by their syntactic mothers.

Principles. Our grammar makes use of the following principles on the SYN dimension:

- *Tree principle*: the graph on SYN must be a tree.
- *Projectivity principle*: SYN must be projective.
- *Valency principle*: the nodes on SYN must satisfy their in and out valencies (lexical attributes *in* and *out*).
- *Order principle*: the dependents of each node and the node itself must be ordered according to the lexicalized strict partial order given by the *order* attribute.
- *Agr principle*: each node must pick out one agreement tuple (*agr*) from the lexicalized set of licensed agreement tuples (*agrs*).[4]
- *Agreement principle*: the agreement tuple *agr* of each node must agree with the agreement tuple of all dependents in the lexicalized set *agree*.

On the SEM dimension, the grammar makes use of the following principles:

- *DAG principle*: the graph on SEM must be a DAG.
- *Valency principle*: the nodes on SEM must satisfy their lexicalized in and out valencies (lexical attributes *in* and *out*).

[4] For nodes associated with words not having agreement linguistically, e.g. adverbs, we license all possible agreement tuples.

On the SYNSEM dimension, we make use of the following principles:

- *Edgeless principle*: the graph on SYNSEM must be edgeless.
- *LinkingEnd principle*: the SYN and SEM dimensions must satisfy the lexical *arg* specifications for realization of the semantic arguments of verbs.
- *LinkingMother principle*: the SYN and SEM dimensions must satisfy the lexical *mod* specifications for realization of the semantic arguments of adverbs. This principle ensures e.g. that only verbs that are modified by adverbs on SYN (e.g. *wants* by *today* in Figure 2.4) can be their arguments on SEM.

Finally, the *Lexicalization principle* ensures that each node is assigned a suitable lexical entry from the lexicon, i.e., one associated with the same word as the node.

Lexicon. The lexicon at the same time specifies the lexical attributes of SYN, SEM and SYNSEM. For example, here is the lexical entry for *wants*:

$$\left\{\begin{array}{l} \mathit{word} = \mathit{wants} \\ \textsc{syn} = \left\{\begin{array}{rl} \mathit{in} & = \{\mathsf{root?}\} \\ \mathit{out} & = \{\mathsf{subj!}, \mathsf{vinf!}, \mathsf{adv}*\} \\ \mathit{order} & = \{(\mathsf{subj}, \uparrow), (\mathsf{subj}, \mathsf{vinf}), (\mathsf{subj}, \mathsf{adv}), \\ & \quad (\uparrow, \mathsf{vinf}), (\uparrow, \mathsf{adv}), (\mathsf{vinf}, \mathsf{adv})\} \\ \mathit{agrs} & = \{(3, \mathit{sg})\} \\ \mathit{agree} & = \{\mathsf{subj}\} \end{array}\right\} \\ \textsc{sem} = \left\{\begin{array}{l} \mathit{in} = \{\mathsf{root!}, \mathsf{th}*\} \\ \mathit{out} = \{\mathsf{ag!}, \mathsf{th!}\} \end{array}\right\} \\ \textsc{synsem} = \left\{\begin{array}{l} \mathit{arg} = \left\{\begin{array}{l} \mathsf{ag} = \{\mathsf{subj}\} \\ \mathsf{th} = \{\mathsf{vinf}\} \end{array}\right\} \\ \mathit{mod} = \{\} \end{array}\right\} \end{array}\right\} \tag{2.7}$$

2.3. Implementing XDG Grammars

In this thesis, we not only develop XDG theoretically, but also implement a parser and an extensive grammar development kit: the XDK. In the XDK, grammars are written in the *XDK description language*, a *metagrammar* with a number of concrete syntaxes (including one based on XML). The metagrammar is statically typed, which makes it very easy to spot errors.

2.3.1. Metagrammar

Using the XDK description language, XDG grammars can be written down just as described above: by first defining the dimensions, then choosing the principles from the set of predefined ones from the *principle library*, and then defining the lexicon. The set of principles is extensible, and each of the existing ones can be freely replaced.

Dimensions. As an example, we show how the types of edge labels (`deflabeltype`), lexical attributes (`defentrytype`) and non-lexical attributes (`defattrstype`) are defined for the SYN dimension of our example metagrammar:

```
deftype "syn.label" {root subj part obj vinf adv}
deftype "syn.label1" "syn.label" | {"^"}
deftype "syn.person" {"1" "2" "3"}
deftype "syn.number" {sg pl}
deftype "syn.agr" tuple("syn.person" "syn.number")

deflabeltype "syn.label"
defentrytype {in: valency("syn.label")
              out: valency("syn.label")
              order: set(tuple("syn.label1" "syn.label1"))
              agrs: iset("syn.agr")
              agree: set("syn.label")}
defattrstype {agr: "syn.agr"}
```
(2.8)

Principles. The principles of the SEM dimension are instantiated as follows:[5]

```
useprinciple "principle.graph" { dims {D: sem} }
useprinciple "principle.dag" { dims {D: sem} }
useprinciple "principle.valency" { dims {D: sem} }
```
(2.9)

Lexicon. The lexical entries can be written down as before, with the slight difference that the *word* attribute is encapsulated in an additional dimension called `lex`. For example, the lexical entry (2.7) is then written as:

```
defentry {
  dim lex {word: "wants"}
  dim syn {in: {root?}
           out: {subj! vinf! adv*}
           order: {[subj "^"] [subj vinf] [subj adv]
                   ["^" vinf] ["^" adv] [vinf adv]}
           agrs: {["3" sg]}
           agree: {subj}}
  dim sem {in: {root! th*}
           out: {ag! th!}}
  dim synsem {arg: {ag: {subj}
                    th: {vinf}}
              mod: {}}}
```
(2.10)

However, simply spelling out the lexical entries quickly becomes infeasible. Therefore, the metagrammar provides means for factorization and combination of partial lexical entries called *lexical classes*, and for the easy statement of alternations. Lexical classes are basically lexical types with complete inheritance. Here are some example lexical classes:

[5] In addition to the DAG (”principle.dag”) and Valency (”principle.valency”) principles of our example grammar, in the XDK we also need to instantiate a principle to establish that the models are graphs (”principle.graph”).

- finite verbs:

```
defclass "fin" Word Agrs {
  dim lex {word: Word}
  dim syn {in: {root?}
           out: {subj!}
           order: <subj "^" obj vinf adv>
           agrs: Agrs
           agree: {subj}}}
```
(2.11)

  `fin` has the two arguments `Word` and `Agrs` for the word of the lexical entry and its set of licensed agreement tuples. On SYN, it licenses at most one incoming edge labeled root, and requires precisely one outgoing edge labeled subj, reflecting that finite verbs always require a subject. The subject must be ordered to the left of the head, and the head must be ordered to the left of the infinitival complement and that to the left of the adverb.[6] The word must agree with its subject.

- verbs in general:

```
defclass "verb" {
  dim syn {out: {adv*}}
  dim sem {in: {root! th*}}}
```
(2.12)

  On SYN, `verb` licenses arbitrary many outgoing edges labeled adv to reflect that verbs can always be modified by adverbs. On SEM, it requires precisely one incoming edge labeled root and licenses arbitrary many labeled th, i.e., it can be the theme of arbitrary many adverbs.

- intransitive verbs:

```
defclass "intrans" {
  dim sem {out: {ag!}}
  dim synsem {arg: {ag: {subj}}}}
```
(2.13)

  On SYN, `intrans` requires one outgoing edge labeled ag for its agent. This agent must be realized by a subject (SYNSEM).

- transitive verbs:

```
defclass "trans" {
  "intrans"
  dim syn {out: {obj!}}
  dim sem {out: {pat!}}
  dim synsem {arg: {pat: {obj}}}}
```
(2.14)

  Transitive verbs inherit the specifications of the class `intrans`. In addition, they syntactically require precisely one object and semantically precisely one patient. The patient is realized by the object.

- verbs requiring an infinitival complement:

```
defclass "vinfc" {
  dim syn {out: {vinf!}}
  dim sem {out: {th!}}
  dim synsem {arg: {th: {vinf}}}}
```
(2.15)

  Such verbs syntactically require an infinitival complement and semantically a theme. The theme is realized by the infinitival complement.

[6] The metagrammar allows to abbreviate the specification of strict partial orders with a notation using angle brackets.

We can then use the classes to generate the lexical entries e.g. for both the intransitive and the transitive alternations of *wants* as follows:

```
defentry {
  "verb"
  ("intrans" | "trans")
  "vinfc"
  "fin" {Word: "wants"
         Agrs: {["3" sg]}}}
```
(2.16)

where the bar between ”intrans” and ”trans” represents a disjunction.

2.3.2. Parser

The constraint parser of the XDK is based on constraint programming in Mozart/Oz, and implements the complete XDG grammar formalism as presented in this thesis, including all principles. All dimensions are processed concurrently. Sentences can be parsed either using the GUI or the commandline version of the solver. If the GUI (Figure 2.6) is used, the *Oz Explorer* (Schulte 1997) displays the solutions of a parse as in Figure 2.7. The solutions can be visualized using several output functors e.g. for LaTeX-output (as in e.g. Figure 2.1) or output in a window, as shown in Figure 2.8.

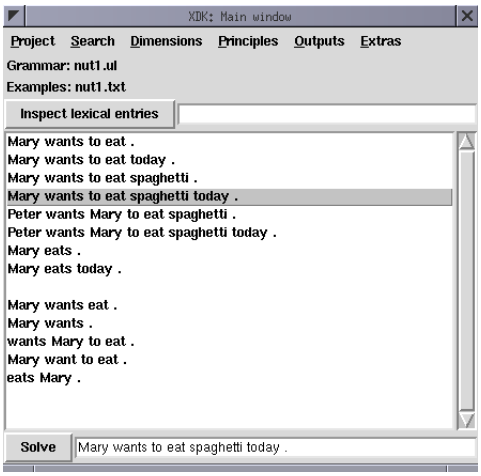


Figure 2.6.: GUI of the XDK

2.4. Comparison with Other Grammar Formalisms

In this section, we compare the main notions of XDG, i.e., dimensions, principles and the lexicon, to their embodiments in the the popular grammar formalisms of CCG, TAG, GB, HPSG, LFG and MTT, before we compare their grammar theories and implementations.

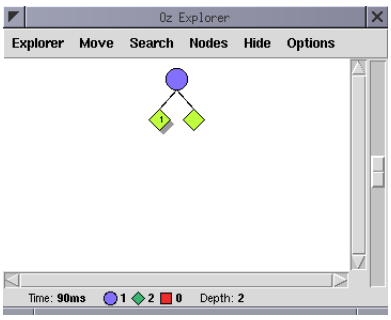


Figure 2.7.: Oz Explorer displaying a parse solution

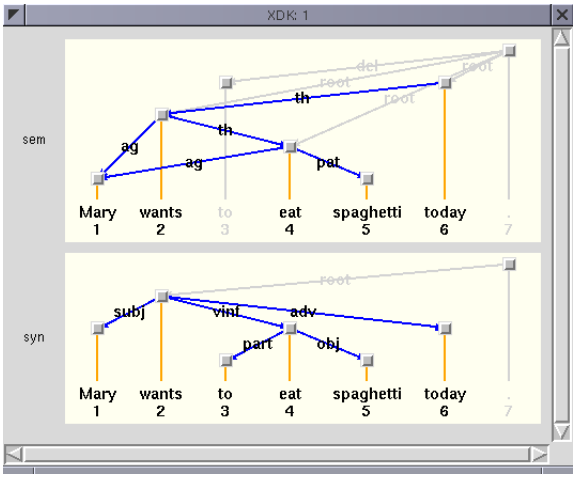


Figure 2.8.: Output functor

2.4.1. Dimensions

By *dimensions*, we mean linguistic aspects such as syntax, semantics and phonology. Dimensions can also be defined in a finer-grained fashion, e.g. by distinguishing within syntax the aspects of grammatical functions and word order, or, within semantics, predicate-argument structure and quantifier scope. In XDG, each dimension is modeled by a different dependency graph. As XDG adopts the *parallel grammar architecture* (see Figure 1.7), all dimensions are autonomous modules, which can be processed concurrently.

Combinatory Categorial Grammar. In CCG, an analysis is a type-logical proof carrying out a syntactic analysis. Prosodic structure is encoded in the syntactic categories, and semantics and information structure are derived from syntax. That is, CCG distinguishes the dimensions of prosody, semantics and information structure, but contrary to XDG, they are no autonomous modules, but encoded in or derived from syntax. CCG has thus a prototypically syntacto-centric architecture (see Figure 1.6). The same holds for the generalization of structures other than syntax proposed by Kruijff & Baldridge (2004), because crucially, they are still derived in lockstep with syntax.

Tree Adjoining Grammar. In TAG, an analysis corresponds to a series of substitutions and adjunctions of lexicalized phrase structure trees called elementary trees. The result of an analysis are two structures: the *derived tree* itself and the "history" of the derivation called *derivation tree*. The derivation tree, which is unordered, more closely corresponds with the dimension of syntactic relations, and the derived tree, which is ordered, with the dimension of word order. Thus, many proposals for a TAG syntax-semantics interface (Candito & Kahane 1998), (Joshi & Shanker 1999), (Kallmeyer & Joshi 2003) use the derivation tree as a starting point, although there are other proposals that use the derived tree (Frank & van Genabith 2001), (Gardent & Kallmeyer 2003). In any case, the resulting architecture is syntacto-centric, as dimensions other than syntactic dimensions, e.g. semantics, are not granted the status of autonomous modules. A proposal for TAG more akin to the parallel grammar architecture is *Synchronous TAG* (*STAG*) (Shieber & Schabes 1990), where sets of trees are synchronously derived, e.g. one tree for syntax, and one for semantics.

Government and Binding. GB has the dimensions of D-Structure (formerly Deep Structure in (Chomsky 1965)), from which it derives the S-Structure (Surface Structure) via application of the generic rule move-α. From the S-Structure, GB derives the dimensions of phonology (Phonetic Form) and semantics (Logical Form). We depict the architecture in Figure 2.9. The architecture is syntacto-centric, like that of CCG and TAG.

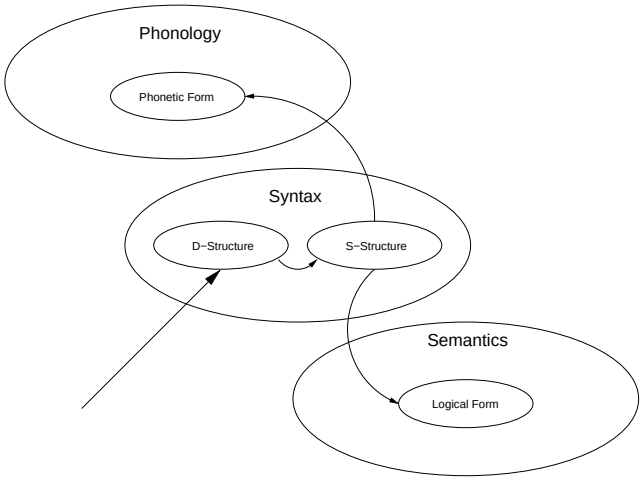


Figure 2.9.: The architecture of Government and Binding

Head-driven Phrase Structure Grammar. In HPSG, linguistic analyses are described in terms of feature structures using a feature logic defined in (Carpenter 1992). In theory, HPSG is able to formulate any kind of architecture, i.e., also the parallel grammar architecture. In practice, however, the HPSG grammar theory founded in (Pollard & Sag 1987, Pollard & Sag 1994) is syntacto-centric just like CCG, TAG and GB: the dimensions of syntax and semantics are both constructed in lockstep according to the feature structure-encoded syntax tree.

Lexical Functional Grammar. LFG defines a clean separation between the syntactic dimensions of constituent structure (c-structure) and functional structure (f-structure): the c-structure is a phrase structure tree, whereas the f-structure is a feature structure capturing syntactic relations, which can also be viewed as a dependency graph. Both c- and f-structure have their own well-formedness conditions. The so-called ϕ mapping provides a bi-directional interface between the two. The interfaces from syntax to phonology and to semantics are not part of the standard LFG theory, but there are proposals for a bi-directional syntax-phonology interface (Butt & King 1998), and for a bi-directional syntax-semantics interface (Frank & Erk 2004). The resulting architecture, depicted in Figure 2.10, is parallel. However, the standard syntax-semantics interface of LFG to *Glue Semantics* (Dalrymple et al. 1995) is not bi-directional but functional (from syntax to semantics), rendering the architecture syntacto-centric again.

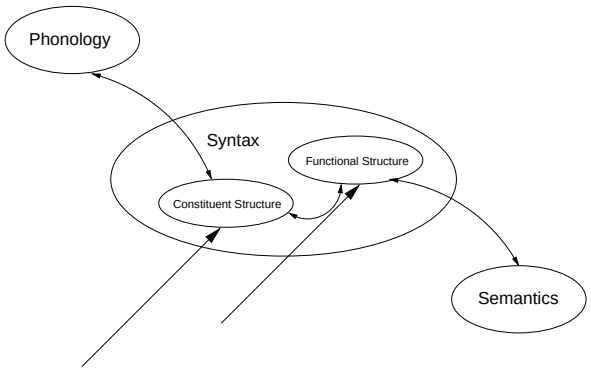


Figure 2.10.: The architecture of Lexical Functional Grammar

Meaning-Text-Theory. MTT (Mel'čuk 1988) makes use of seven dimensions which are called *strata*:

1. Semantic Representation (SemR) (meaning)
2. Deep Syntactic Representation (DSyntR)
3. Surface Syntactic Representation (SSyntR)
4. Deep Morphological Representation (DMorphR)
5. Surface Morphological Representation (SMorphR)
6. Deep Phonological Representation (DPhonR)
7. Surface Phonological Representation (SPhonR) (text)

where the endpoints of this architecture are meaning (SemR) and text (SPhonR). Each stratum has its own well-formedness conditions called well-formedness rules in (Mel'čuk & Polguère 1987) and later criteria in (Iordanskaja & Mel'čuk 2005). The relation between meaning and

text is mediated via bi-directional interfaces. Contrary to the parallel grammar architecture of XDG, however, interfaces exist only for adjacent strata, but not for non-adjacent ones such as SMorphR and SemR. This leads to the architecture outlined in Figure 2.11.

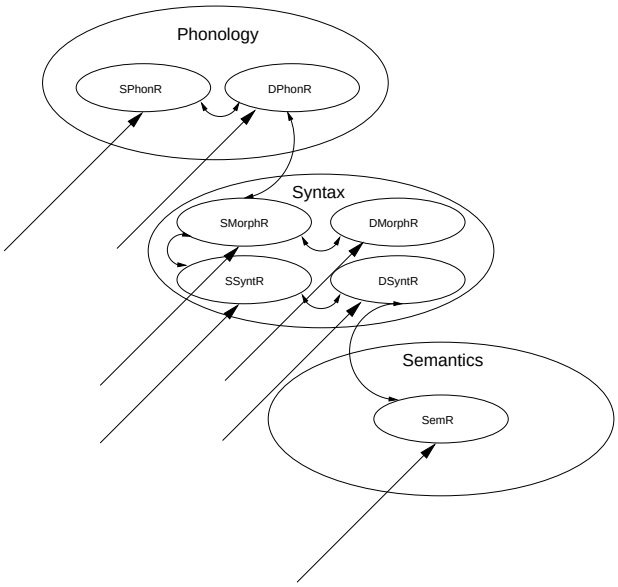


Figure 2.11.: The architecture of Meaning Text Theory

2.4.2. Principles

What are the concepts related to the XDG principles in the other grammar formalisms?

Combinatory Categorial Grammar. The principles in XDG roughly correspond to the combinatory rules of CCG: functional application, functional composition and type raising, which exist in various flavors (forward, backward, crossing), where e.g. functional application can be likened to the notion of valency in XDG. CCG constrains the number of these rules by meta rules called principles: the principle of adjacency, the principle of consistency, and the principle of inheritance, which have no counterpart in XDG. The main difference of the CCG rules to the XDG principles is that they are formulated from a proof-theoretic perspective, whereas XDG principles take a model-theoretic stance.

Tree Adjoining Grammar. Compared to TAG, the principles of XDG correspond to the two simple modes of tree composition, i.e., substitution and adjunction. The two can be likened to valency in XDG, where substitution is used for complementation, and adjunction for modification. TAG has no other principles or rules; everything else (e.g. order) is encoded in the elementary trees in the lexicon. However, this minimal approach needs to be extended in practice (XTAG Research Group 2001), leading e.g. to the feature extensions of *Feature-Based Tree Adjoining Grammar* (*FB-TAG*).

Government and Binding. The principles of GB comprise e.g. the move-α rule schema, the θ-criterion, the projection principle and the case filter. GB principles are similar to XDG principles, but there are two main differences: GB principles are not formulated in a logic but in natural language, and they are mutually dependent and thus less modular than in XDG: for instance, to account fully for the notion of valency, GB relies on interactions of the θ-criterion with the projection principle and the case filter.

Head-driven Phrase Structure Grammar. HPSG proposes two kinds of well-formedness conditions: HPSG principles such as the Head Feature Principle and the Subcategorization Principle, and HPSG rules such as the Head Complement Rule and the Head Modifier Rule. HPSG principles are more general and language-independent, whereas HPSG rules are generalizations of context-free rules and language-dependent. XDG principles are more similar to HPSG principles than HPSG rules. For example, the Subcategorization Principle (replaced by the Valence Principle in later versions of HPSG) is analogous to the Valency principle of XDG.

Lexical Functional Grammar. The principles of LFG are very general: c-structure is constrained by X-bar theory (Jackendoff 1977), and f-structure by functional uniqueness, functional completeness and functional coherence. Functional completeness and coherence form the counterpart of the Valency principle in XDG. Other XDG principles, e.g. agreement, are not formulated as LFG principles, but as path equations in the lexicon.

Meaning-Text-Theory. In MTT, the counterparts of the XDG principles are called well-formedness rules of the individual strata, which were later called criteria.

2.4.3. Lexicon

We now compare the lexicon of XDG with that of the other grammar formalisms.

Combinatory Categorial Grammar. In CCG, the lexicon pairs each word with a pair consisting of a syntactic category and a semantic representation (a λ-term). The syntactic category encodes simultaneously the syntactic valency requirements and word order, whereas the semantic representation encodes the meaning of the word.

Tree Adjoining Grammar. In TAG, the lexicon consists of elementary trees. In the specialization of TAG most often used for modeling natural language, *Lexicalized Tree Adjoining Grammar* (*LTAG*), each of these trees must have at least one anchor, i.e., it must be associated with a word. All alternations, e.g. of verbs, must be compiled out into different elementary trees, which leads to very large lexicons. To reduce their size, many extensions such as *metagrammar* (Candito 1996), and *eXtensible MetaGrammar* (*XMG*) (Crabbé & Duchier 2004) (Crabbé 2005) have been proposed.[7]

[7] XMG, was actually the major source of inspiration for the metagrammar of the XDK.

Government and Binding. In the GB lexicon, words are basically paired with a valency frame specifying the semantic valency requirements in terms of θ-roles. That is, the lexicon of GB includes less information than that of XDG, lacking specifications of agreement, government, and also linking.

Head-driven Phrase Structure Grammar. The HPSG lexicon pairs words with feature structures. These structures are more complex than XDG lexical entries: they are often deeply nested, make use of structure sharing, and allow even arbitrary relations (e.g. append) to be expressed. HPSG lexical entries can be easily extended with new features, and lexical economy is ensured by the HPSG type hierarchy and lexical rules.

Lexical Functional Grammar. In the LFG lexicon, words are paired with valency frames and f-structure path equations. The latter have no direct counterpart in XDG. In the implementations of LFG, the mechanisms of template and lexical rules ensure lexical economy.

Meaning-Text-Theory. The MTT lexicon is called Explanatory Combinatorial Dictionary (ECD). In ECD, lexical entries are split into three zones:

1. semantic zone
2. syntactic zone
3. lexical combinatorics zone

In the semantic zone, the semantics of the lexical entry are described using a semantic network. The syntactic zone defines syntactic valency and the government pattern, which establishes a linking between the syntactic and semantic arguments called actants. The lexical combinatorics zone describes relations between lexemes, e.g. *multiword expressions*multiword expression. The MTT lexicon is by far the most complex of the presented grammar formalisms, and is also far more complex than the XDG lexicon. Interestingly, similar to XDG, the specifications for syntax and semantics are largely independent, and the lexical entries also contain linking specifications. MTT is the only one of the presented grammar formalisms to handle multiword expressions. For XDG, ideas to handle multiword expressions using a notion called *groups* are presented in (Debusmann 2004*b*) and extended in (Pelizzoni & das Gracas Volpe Nunes 2005).

2.4.4. Grammar Theory

So far, the emphasis of our research was on the modeling of complex, hand selected phenomena. Thus, so far, there are no large-scale grammars comparable to those for the established grammar formalisms, e.g. *XTAG* (XTAG Research Group 2001) for TAG, or the *English Resource Grammar* (*ERG*) (Copestake & Flickinger 2000) for HPSG, available for XDG.

However, with respect to syntax, we have developed grammars for German (Debusmann 2001, Bader et al. 2004), Dutch (Debusmann & Duchier 2002), and English (this thesis),

covering e.g. the phenomena of topicalization, pied piping, scrambling and cross-serial dependencies. With respect to semantics and the syntax-semantics interface, we have developed accounts of control and raising (e.g. Debusmann, Duchier & Kruijff 2004), scope ambiguities and underspecification (Debusmann, Duchier, Koller, Kuhlmann, Smolka & Thater 2004), and a modular version of Steedman's (2000*a*) prosodic account of information structure (Debusmann, Postolache & Traat 2005). These hand selected phenomena serve as a proof-of-concept of XDG grammar theory, and combined with the modular design of XDG, they are a strong indication for its scalability: that given enough resources, large-scale grammars can indeed be constructed.

2.4.5. Implementation

From the beginning, XDG was geared towards an extensible concurrent implementation using constraint programming, which was in fact developed in parallel with the grammar formalism. The resulting constraint parser is reasonably fast on the existing handwritten grammars, and the extensive grammar development kit, the XDK, is comfortable and instructive, e.g. for experimenting with grammar formalisms based on dependency grammar and for teaching. As a result, the XDK has already been successfully employed for teaching, e.g. in a course at ESSLLI 2004 (Debusmann & Duchier 2004), and a Fortgeschrittenenpraktikum at the Universität des Saarlandes, also in 2004 (Debusmann 2004*a*).

As there are no large-scale grammars for XDG available yet, we could not prove that the parser is scalable. Negative evidence comes from grammar induction studies (Korthals 2003, Möhl 2004, Bojar 2004, Narendranath 2004), indicating that the current XDG parser is not usable for large-scale parsing, which would not be a reason to wonder: the parser is almost unoptimized, not yet profiled, and does not use global constraints, which are usually indispensable for efficient constraint programming. In addition, the parser does not use any of the statistical techniques used to boost the efficiency of the parsers for other grammar formalisms, such as supertagging in *OpenCCG* (White 2004).

2.5. Summary

In this chapter, we have given a walkthrough of the main concepts of XDG. The models of XDG are multi-dimensional dependency graphs called multigraphs. These models are described by XDG grammars, which are defined in three steps: defining the dimensions, then the principles, and then the lexicon. The implementation of XDG, the XDK, provides a constraint parser and a metagrammar for convenient grammar development. The metagrammar facilitates grammar writing by providing means for factorization and alternation using lexical classes. We compared the main concepts of XDG in relation to their counterparts in a number of existing grammar formalisms, and compared compared their grammar theory and implementation.

Part I.

Formalization

3. XDG—A Description Language for Multigraphs

After the informal introduction to the main concepts of XDG in the previous chapter, we now proceed with presenting a formalization of XDG as a description language for multigraphs, which will serve as the basis for the formalization of the key concepts of dependency grammar in chapter 4, and for our investigations of the expressivity (chapter 5) and computational complexity (chapter 6) of XDG.

3.1. Multigraphs

We begin in this section with formalizing multigraphs and the relations induced by them. We define *multigraphs* as follows.

Definition 1 (Multigraph)**.** *A multigraph is a tuple* (V, D, W, w, L, E, A, a) *consisting of:*

1. *a finite interval* V *of the natural numbers starting from* 1 *called nodes*
2. *a finite set* D *of dimensions*
3. *a finite set* W *of words*
4. *the* node-word mapping $w \in V \to W$
5. *a finite set of* L *of edge labels*
6. *a finite set* $E \subseteq V \times V \times D \times L$ *of edges*
7. *a finite set* A *of attributes*
8. *the* node-attributes mapping $a \in V \to D \to A$

Figure 3.1 shows an example multigraph, repeating Figure 2.4.[1] As explained in section 2.1.1, we assume an additional root node corresponding to the end-of-sentence marker:

1. the set of nodes V is $\{1, 2, 3, 4, 5, 6, 7\}$
2. the set of dimensions D is $\{\text{SYN}, \text{SEM}, \text{SYNSEM}\}$

[1] Only the attributes of the nodes 1 and 2 on SYN are highlighted, as the attributes of the other nodes are irrelevant here.

3. the set of words W is $\{Mary, wants, to, eat, spaghetti, today, .\}$

4. the node-word mapping w is $\{1 \mapsto Mary,\ 2 \mapsto wants,\ 3 \mapsto to,\ 4 \mapsto eat \ldots\}$

5. the set L of edge labels is defined as the union of the edge labels of the SYN and SEM dimensions and the additional anchor label $\uparrow$:

$$\begin{array}{rcl} L_{\text{SYN}} & = & \{\mathsf{root}, \mathsf{subj}, \mathsf{part}, \mathsf{obj}, \mathsf{vinf}, \mathsf{adv}\} \\ L_{\text{SEM}} & = & \{\mathsf{root}, \mathsf{del}, \mathsf{ag}, \mathsf{pat}, \mathsf{th}\} \\ L & = & L_{\text{SYN}} \cup L_{\text{SEM}} \cup \{\uparrow\} \end{array} \tag{3.1}$$

6. the set E of edges is:

$$\{(2,1,\text{SYN},\mathsf{subj}), (2,4,\text{SYN},\mathsf{vinf}), \ldots, (2,1,\text{SEM},\mathsf{ag}), (2,4,\text{SEM},\mathsf{th}), \ldots\} \tag{3.2}$$

7. the set A of attributes is characterized by the following three record types (cf. section 2.2.4):

   a) record type for SYN:

$$\left\{ \begin{array}{l} lex : \left\{ \begin{array}{r} in : valency(L_{\text{SYN}}) \\ out : valency(L_{\text{SYN}}) \\ order : set(tuple(L_{\text{SYN}} \,|\, \{\uparrow\}, L_{\text{SYN}} \,|\, \{\uparrow\})) \\ agrs : set(Agr) \\ agree : set(L_{\text{SYN}}) \end{array} \right\} \\ agr : Agr \end{array} \right\} \tag{3.3}$$

   b) record type for SEM:

$$\left\{ lex : \left\{ \begin{array}{r} in : valency(L_{\text{SEM}}) \\ out : valency(L_{\text{SEM}}) \end{array} \right\} \right\} \tag{3.4}$$

   c) record type for SYNSEM:

$$\left\{ lex : \left\{ \begin{array}{r} arg : vec(L_{\text{SEM}}, set(L_{\text{SYN}})) \\ mod : set(L_{\text{SEM}}) \end{array} \right\} \right\} \tag{3.5}$$

8. the node-attributes mapping is:

$$\left\{ \begin{array}{l} 1 \mapsto \text{SYN} \mapsto \left\{ \begin{array}{l} lex = \left\{ \begin{array}{r} in = \{\mathsf{subj?}, \mathsf{obj?}\} \\ out = \{\} \\ order = \{\} \\ agrs = \{(3, sg)\} \\ agree = \{\} \end{array} \right\} \\ agr = (3, sg) \end{array} \right\}, \\ 2 \mapsto \text{SYN} \mapsto \left\{ \begin{array}{l} lex = \left\{ \begin{array}{r} in = \{\mathsf{root?}\} \\ out = \{\mathsf{subj!}, \mathsf{vinf!}, \mathsf{adv}*\} \\ order = \{(\mathsf{subj}, \uparrow), (\mathsf{subj}, \mathsf{vinf}), (\mathsf{subj}, \mathsf{adv}), \\ \qquad (\uparrow, \mathsf{vinf}), (\uparrow, \mathsf{adv}), (\mathsf{vinf}, \mathsf{adv})\} \\ agrs = \{(3, sg)\} \\ agree = \{\mathsf{subj}\} \end{array} \right\} \\ agr = (3, sg) \end{array} \right\}, \\ \ldots \end{array} \right\} \tag{3.6}$$

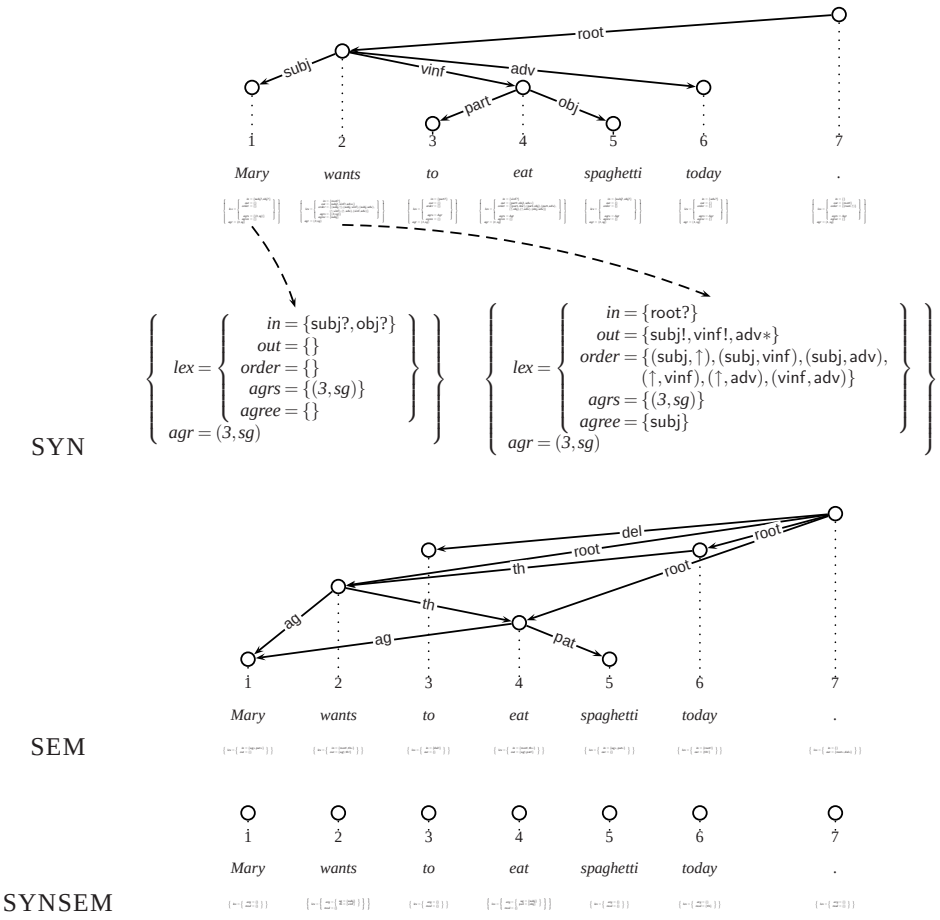


Figure 3.1.: Multigraph (simultaneous syntactic and semantic analysis)

Each dimension $d \in D$ of a multigraph induces two relations: the *labeled edge relation* ($\xrightarrow{\cdot}_d$) and the *precedence relation* ($<$).

Definition 2 (Labeled Edge Relation)**.** *Given two nodes v and v' and a label l, the labeled edge relation $v \xrightarrow{l}_d v'$ holds if and only if there is an edge from v to v' labeled l on dimension d:*

$$\xrightarrow{\cdot}_d \quad = \quad \{(v,v',l) \mid (v,v',d,l) \in E\} \tag{3.7}$$

where the dot $\cdot$ is a placeholder for the edge label.

Definition 3 (Precedence Relation)**.** *Given two nodes v and v', the total order on the natural numbers induces the precedence relation: $v < v'$ holds if and only if v is smaller than v'.*

3.2. A Description Language for Multigraphs

Having introduced multigraphs formally, we can define XDG as a description language for them. We formulate XDG in higher order logic (Church 1940, Andrews 2002), which we

use as a tool to illustrate the semantics of XDG. Thereby, we deliberately neglect that in practice, XDG does not seem to require the full expressivity of higher order logic. In fact, in the grammars which we will present throughout this thesis, we will only make use of its first order fragment.

We define XDG by first defining the types of the language, then its terms, and then its signature. Since each multigraph has different dimensions, words, edge labels and attributes, the types in the signature vary. We capture this by parametrizing the signature with a tuple characterizing the type of the dimensions, words, edge labels and attributes of a multigraph called *multigraph type*. Figure 3.2 illustrates this idea: the signature relates the types and terms of XDG. The multigraph type, made up from types, determines the types in the signature.

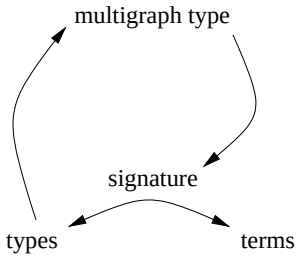


Figure 3.2.: Structure of XDG

3.2.1. Types

We begin with defining the types of the logic and their interpretation.

Definition 4 (Types)**.** *We define the types Ty of XDG given a set At of atoms (arbitrary symbols) in simply typed lambda calculus with records:*

$$\begin{array}{lll} a \in At & & \\ T \in Ty \;::=\; \mathsf{B} & & \textit{boolean} \\ \quad\mid\; \mathsf{V} & & \textit{node} \\ \quad\mid\; T_1 \to T_2 & & \textit{function} \\ \quad\mid\; \{a_1,\ldots,a_n\} & & \textit{finite domain } (n \geq 1) \\ \quad\mid\; \{a_1 : T_1,\ldots,a_n : T_n\} & & \textit{record} \end{array} \tag{3.8}$$

where for finite domains and records, $a_1,\ldots,a_n$ are pairwise distinct, and we forbid empty finite domains.

Definition 5 (Interpretation of Types)**.** *We interpret the types as follows:*

- B *as* $\{0,1\}$
- V *as a finite interval of the natural numbers starting with* 1
- $T_1 \to T_2$ *as the set of all functions from the interpretation of* T_1 *to the interpretation of* T_2

- $\{a_1,\ldots,a_n\}$ *as the set* $\{a_1,\ldots,a_n\}$
- $\{a_1 : T_1,\ldots,a_n : T_n\}$ *as the set of all functions* f *with*
  1. *Dom* $f = \{a_1,\ldots,a_n\}$
  2. *for all* $1 \leq i \leq n$, $f\ a_i$ *is an element of the interpretation of* T_i

Definition 6 (Notational Conveniences for Types)**.** *We introduce notational conveniences for:*

- *unions:* $\{a_1,\ldots,a_k\} \mid \{a_{k+1},\ldots,a_n\} \stackrel{\text{def}}{=} \{a_1,\ldots,a_n\}$
- *sets:* $set(T) \stackrel{\text{def}}{=} T \to \mathsf{B}$*, i.e, we model sets by their characteristic functions*
- *tuples:* $tuple(T_1,\ldots,T_n) \stackrel{\text{def}}{=} \{1 : T_1,\ldots,n : T_n\}$
- *vectors:* $vec(\{a_1,\ldots,a_n\},T) \stackrel{\text{def}}{=} \{a_1 : T,\ldots,a_n : T\}$*, i.e., vectors are simply abbreviations of records where each attribute has the same type*
- *valencies:* $valency(\{a_1,\ldots,a_n\}) \stackrel{\text{def}}{=} vec(\{a_1,\ldots,a_n\},!,?,*,0)$

As examples, consider the record types defined in (3.3)–(3.5) above.

3.2.2. Multigraph Type

Definition 7 (Multigraph Type)**.** *A multigraph type is a tuple* $MT = (Dim, Word, lab, attr)$*, where*

1. $Dim \in Ty$ *is a finite domain of dimensions*
2. $Word \in Ty$ *is a finite domain of words*
3. $lab \in Dim \to Ty$ *is a function from dimensions to label types, i.e., the type of the edge labels on that dimension. Label types must be finite domains.*
4. $attr \in Dim \to Ty$ *is a function from dimensions to attributes types, i.e., the type of the attributes on that dimension. Attributes types can be any type.*

As an example, we depict the multigraph type $MT = (Dim, Word, lab, attr)$ for the grammar

presented in section 2.2.4:[2]

$$
\begin{array}{rcl}
\mathit{Dim} & = & \{\textsc{syn}, \textsc{sem}, \textsc{synsem}\} \\
\mathit{Word} & = & \{\mathit{Mary}, \mathit{wants}, \mathit{to}, \mathit{eat}, \mathit{spaghetti}, \mathit{today}, ., \ldots\} \\
\mathit{lab} & = & \left\{ \begin{array}{rl} \textsc{syn} \mapsto & \{\mathsf{root}, \mathsf{subj}, \mathsf{part}, \mathsf{obj}, \mathsf{vinf}, \mathsf{adv}, \ldots\} \\ \textsc{sem} \mapsto & \{\mathsf{root}, \mathsf{del}, \mathsf{ag}, \mathsf{pat}, \mathsf{th}, \ldots\} \\ \textsc{synsem} \mapsto & \{\mathsf{o}\} \end{array} \right\} \\
\mathit{attr} & = & \left\{ \begin{array}{rl} \textsc{syn} \mapsto & \left\{ \begin{array}{rl} \mathit{lex} : & \left\{ \begin{array}{rl} \mathit{in} : & \mathit{valency}(L_{\textsc{syn}}) \\ \mathit{out} : & \mathit{valency}(L_{\textsc{syn}}) \\ \mathit{order} : & \mathit{set}(\mathit{tuple}(L_{\textsc{syn}} \mid \{\uparrow\}, L_{\textsc{syn}} \mid \{\uparrow\})) \\ \mathit{agrs} : & \mathit{set}(\mathit{Agr}) \\ \mathit{agree} : & \mathit{set}(L_{\textsc{syn}}) \end{array} \right\} \\ \mathit{agr} : & \mathit{Agr} \end{array} \right\} \\ \textsc{sem} \mapsto & \left\{ \mathit{lex} : \left\{ \begin{array}{rl} \mathit{in} : & \mathit{valency}(L_{\textsc{sem}}) \\ \mathit{out} : & \mathit{valency}(L_{\textsc{sem}}) \end{array} \right\} \right\} \\ \textsc{synsem} \mapsto & \left\{ \mathit{lex} : \left\{ \begin{array}{rl} \mathit{arg} : & \mathit{vec}(L_{\textsc{sem}}, \mathit{set}(L_{\textsc{syn}})) \\ \mathit{mod} : & \mathit{set}(L_{\textsc{sem}}) \end{array} \right\} \right\} \end{array} \right\}
\end{array}
\tag{3.9}
$$

To bring multigraphs and multigraph types together, we must define what it means for a multigraph M to have multigraph type MT, or in other words, what it means for M to be compatible with MT. We define *compatibility* writing $M(T)$ for the interpretation of type T over M.

Definition 8 (Compatibility of Multigraphs and Multigraph Types)**.** *A multigraph $M = (V, D, W, w, L, E, A, a)$ has multigraph type $MT = (\mathit{Dim}, \mathit{Word}, \mathit{lab}, \mathit{attr})$ if and only if:*

1. *The dimensions are the same:*
$$D = M(\mathit{Dim}) \tag{3.10}$$

2. *The words of the multigraph are a subset of the words of the multigraph type:*
$$W \subseteq M(\mathit{Word}) \tag{3.11}$$

3. *The edges in E have the right edge labels for their dimension:*
$$\forall (v, v', d, l) \in E \;:\; l \in M(\mathit{lab}\; d) \tag{3.12}$$

4. *The nodes have the right attributes for their dimension:*
$$\forall v \in V \;:\; \forall d \in D \;:\; (a\; v\; d) \in M(\mathit{attr}\; d) \tag{3.13}$$

3.2.3. Terms

The terms of XDG augment simply typed lambda calculus with atoms, records and record selection.

[2] As we forbid empty finite domains, the edge labels of the SYNSEM dimension must include a "dummy" label (here: o).

Definition 9 (Terms)**.** *Given a set of atoms At and constants Con, we define the set of terms Te as:*

$$\begin{array}{rcll} a \in At & & & \\ c \in Con & & & \\ t \in Te & ::= & x & \textit{variable} \\ & | & c & \textit{constant} \\ & | & a & \textit{atom} \\ & | & \lambda x : T.\ t & \textit{abstraction} \\ & | & t_1\ t_2 & \textit{application} \\ & | & a & \textit{atom} \\ & | & \{a_1 = t_1, \ldots, a_n = t_n\} & \textit{record} \\ & | & t.a & \textit{record selection} \end{array} \tag{3.14}$$

where for records, $a_1, \ldots, a_n$ are pairwise distinct.

Definition 10 (Notational Conveniences for Terms)**.** *We introduce notational conveniences for:*

- *sets over type T:*

$$\{t_1, \ldots, t_n\} \stackrel{\text{def}}{=} \lambda x : T.\ x \doteq t_1 \vee \ldots \vee x \doteq t_n \tag{3.15}$$

  where $\doteq$ stands for equality.

- *tuples:*

$$(t_1, \ldots, t_n) \stackrel{\text{def}}{=} \{1 = t_1, \ldots, n = t_n\} \tag{3.16}$$

3.2.4. Signature

The signature of XDG defines two kinds of constants: the *logical constants* and the *multigraph constants*, where the latter are determined by a multigraph type $MT = (Dim, Word, lab, attr)$.

Definition 11 (Logical Constants)**.** *The logical constants include the type constant* B *and the following term constants:*

$$\begin{array}{rcll} 0 & : & \mathsf{B} & \textit{false} \\ \Rightarrow & : & \mathsf{B} \to \mathsf{B} \to \mathsf{B} & \textit{implication} \\ \doteq_T & : & T \to T \to \mathsf{B} & \textit{equality (for each type } T\textit{)} \\ \exists_T & : & (T \to \mathsf{B}) \to \mathsf{B} & \textit{existential quantification (for each type } T\textit{)} \end{array} \tag{3.17}$$

which are interpreted as usual.

Definition 12 (Multigraph Constants)**.** *The multigraph constants include the type constant* V *and the following term constants:*

$$\begin{array}{rcll} \dot{\longrightarrow}_d & : & \mathsf{V} \to \mathsf{V} \to lab\ d \to \mathsf{B} & \textit{labeled edge } (d \in Dim) \\ < & : & \mathsf{V} \to \mathsf{V} \to \mathsf{B} & \textit{precedence} \\ (W\ \cdot) & : & \mathsf{V} \to Word & \textit{word} \\ (d\ \cdot) & : & \mathsf{V} \to attr\ d & \textit{attributes } (d \in Dim) \end{array} \tag{3.18}$$

where we interpret

- $\dot{\longrightarrow}_d$ *as the labeled edge relation on dimension* d.
- $<$ *as the precedence relation*
- $(W\ \cdot)$ *as the word, e.g.* $(W\ v)$ *represents the word of node* v
- $(d\ \cdot)$ *as the attributes on* d*, e.g.* $(d\ v)$ *represents the attributes of node* v *on dimension* d

Definition 13 (Notational Conveniences for Logical Constants)**.** *We introduce notational conveniences for:*

- 1 *(true)*
- $\neg$ *(negation)*
- $\vee$ *(disjunction)*
- $\wedge$ *(conjunction)*
- $\Leftrightarrow$ *(equivalence)*
- $\dot{\neq}_T$ *(inequality)*
- $\exists^1_T$ *(unique existential quantification)*
- $\forall_T$ *(universal quantification)*

using the usual logical equivalences.

Definition 14 (Notational Conveniences for Sets)**.** *We introduce notational conveniences for sets, building on the definition* $x \in_T y \stackrel{\text{def}}{=} y\ x$*, and using the usual equivalences:*

- $\emptyset$ *(empty set)*
- $\notin_T$ *(exclusion)*
- $\cap_T$ *(intersection)*
- $\cup_T$ *(union)*
- $\subseteq_T$ *(subset)*

Definition 15 (Notational Conveniences for Multigraph Constants)**.** *We introduce notational conveniences for*

- *edges where the edge label is irrelevant:*

$$v \rightarrow_d v' \stackrel{\text{def}}{=} \exists_{lab\ d} l : v \stackrel{l}{\longrightarrow}_d v' \tag{3.19}$$

- *strict dominance:*

$$v \rightarrow^+_d v' \stackrel{\text{def}}{=} v \rightarrow_d v' \vee (\exists_V v'' : v \rightarrow_d v'' \wedge v'' \rightarrow^+_d v') \tag{3.20}$$

- *non-strict dominance:*

$$v \rightarrow^*_d v' \stackrel{\text{def}}{=} v \doteq v' \vee v \rightarrow^+_d v' \tag{3.21}$$

3.2.5. Grammar

The definition of an XDG grammar is now easy.

Definition 16 (Grammar)**.** *An XDG grammar $G = (MT, P)$ is defined by a multigraph type MT and a set P of formulas called* principles*. Each principle must be formulated according to the signature determined by MT.*

3.2.6. Models

Next, we define the models of an XDG grammar and its string language.

Definition 17 (Models)**.** *The models of a grammar $G = (MT, P)$ are all multigraphs M which:*

1. *have multigraph type MT*
2. *satisfy all principles P*

where M satisfies a principle if and only if it is true for M.

3.2.7. String Language

Definition 18 (String Language)**.** *Given a grammar G, $L(G)$ is the set of all strings $s = w_1 \dots w_n$ such that:*

1. *there are as many nodes as words: $V = \{1, \dots, n\}$*
2. *concatenating the words of the nodes yields s: $(W\ 1) \dots (W\ n) = s$*

3.3. Summary

In this chapter, we first presented a formal definition of multigraphs, before we developed a formalization of XDG as a description language for multigraphs based in higher order logic. Here, the crucial step was the introduction of multigraph types to parametrize the signature of the logic. Multigraph types also played a role in the subsequent definitions of XDG grammars and XDG models, preceding that of the string language of a grammar.

4. Dependency Grammar as Multigraph Description

In this chapter, we apply XDG as a grammar formalism for dependency grammar. In particular, we show how the key concepts of DG can be reformulated as principles on multigraphs.

4.1. Graph Shape

Most grammar formalisms based on DG only license graphs that have the shape of DAGs or trees, even though there are exceptions like WG (Hudson 1990), which allows unrestricted graphs. An XDG dimension can be any kind of graph. We constrain its shape using principles such as the DAG principle (cf. the SEM dimension in the grammar in section 2.2.4), the Tree principle (SYN) and the Edgeless principle (SYNSEM).

4.1.1. DAG Principle

The *DAG principle* states a dimension must have no cycles.

Principle 1 (DAG)**.** *Given a dimension d, the DAG principle is defined as:*

$$dag_d \quad = \quad \forall v : \neg(v \rightarrow_d^+ v) \tag{4.1}$$

4.1.2. Tree Principle

The *Tree principle* states, given a dimension d, that d is a tree, i.e., there must be no cycles, there must be precisely one root and each node must have at most one incoming edge.

Principle 2 (Tree)**.** *Given a dimension d, the Tree principle is defined as:*

$$tree_d \quad = \quad \forall v : \neg(v \rightarrow_d^+ v) \quad \wedge \quad \exists^1 v : \neg\exists v' : v' \rightarrow_d v \quad \wedge \quad \forall v : (\neg\exists v' : v' \rightarrow_d v) \vee (\exists^1 v' : v' \rightarrow_d v) \tag{4.2}$$

4.1.3. Edgeless Principle

The *Edgeless principle* states, given a dimension d, that d must be without edges.

Principle 3 (Edgeless)**.** *Given a dimension d, the Edgeless principle is defined as:*

$$edgeless_d \quad = \quad \forall v : \neg\exists v' : v \rightarrow_d v' \tag{4.3}$$

4.2. Projectivity

Projectivity is a central concept in DG. The idea is to forbid crossing edges, i.e., edges that cross any of the projection edges of the nodes higher up or to the side in the graph. A projective dependency tree, without crossing edges, was given in Figure 1.2, and a non-projective one in Figure 1.4. As already noted in section 1.1.1, projectivity is optional for DG. Consequently, in XDG, we can freely decide for each dimension whether it should be projective by or not.

4.2.1. Projectivity Principle

We express projectivity with the *Projectivity principle*, defined given a dimension d, and requiring that for all edges from v to v', all nodes v'' between v and v' must be below v.

Principle 4 (Projectivity)**.** *Given a dimension d, the Projectivity principle is defined as:*

$$\begin{array}{l} \mathit{projectivity}_d = \forall v, v' : \\ \quad v \rightarrow_d v' \ \wedge\ v < v' \ \Rightarrow\ \forall v'' : v < v'' \wedge v'' < v' \ \Rightarrow\ v \rightarrow_d^+ v'' \ \wedge \\ \quad v \rightarrow_d v' \ \wedge\ v' < v \ \Rightarrow\ \forall v'' : v' < v'' \wedge v'' < v \ \Rightarrow\ v \rightarrow_d^+ v'' \end{array} \tag{4.4}$$

4.3. Lexicalization

As explained in chapter 2, XDG grammars are typically lexicalized, consisting of:

1. a small set of principles
2. a large set of lexical entries which instantiate the principles

We express lexicalization using a principle. Thus, whereas in most other grammar formalisms, e.g. those presented in section 2.4, lexicalization is integral, it is optional in XDG.

4.3.1. Lexical Entries

We begin by defining the type of a lexical entry.

Definition 19 (Lexical Entry)**.** *Given the word type Word, n dimensions $d_1, \ldots, d_n$ and corresponding record types $T_1, \ldots, T_n$, the type of a lexical entry is defined as:*

$$E \ = \ \left\{ \begin{array}{c} \mathit{word} : \mathit{Word} \\ d_1 : T_1 \\ \ldots \\ d_n : T_n \end{array} \right\} \tag{4.5}$$

where T_i ($1 \leq i \leq n$) is the lexical attributes type *of dimension d_i.*

4.3.2. Lexical Attributes

The attributes of XDG connect the lexical entries with the nodes of the actual analysis.

Definition 20 (Lexical and Non-lexical Attributes)**.** *Given n dimensions $d_1, \ldots, d_n$, corresponding lexical attributes types $T_1, \ldots, T_n$, and atoms $a_1, \ldots, a_m$ ($m \geq 0$), the attributes on dimension d_i ($1 \leq i \leq n$) are defined as:*

$$attr\ d_i \quad = \quad \left\{ \begin{array}{l} lex : T_i \\ a_1 : \ldots \\ \ldots \\ a_m : \ldots \end{array} \right\} \tag{4.6}$$

where we call the attributes inside the lex subrecord lexical attributes *of d_i, and the attributes outside, i.e., $a_1, \ldots, a_m$,* non-lexical attributes.

4.3.3. Lexicalization Principle

Lexicalization is put to work by the *Lexicalization principle*, which requires the following:

1. A lexical entry *e* must be selected for each node.
2. The lexical entry *e* must be associated with the same word as the node.
3. Given *n* dimensions $d_1, \ldots, d_n$, the lexical attributes for each dimension d_i ($1 \leq i \leq n$) must be equal to the corresponding attributes for d_i in *e*.

As a result, whenever a lexical entry is selected on one of the dimensions, it immediately determines the lexical attributes of all the other dimensions as well, and thereby synchronizes them.

Principle 5 (Lexicalization)**.** *Given n dimensions $d_1, \ldots, d_n$ and a lexical entry type E, the Lexicalization principle must be instantiated with a lexicon lex, which is a set of lexical entries of type E, and is defined as:*

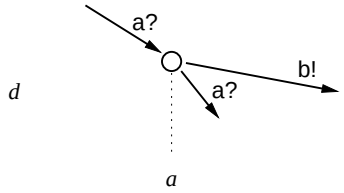
$$\begin{array}{l} lexicalization_{d_1,\ldots,d_n} = \lambda lex.\ \forall v : \\ \quad \exists e : e \in lex \quad \wedge \\ \quad e.word \doteq (W\ v) \quad \wedge \\ \quad (d_1\ v).lex \doteq e.d_1 \quad \wedge \ \ldots \ \wedge \quad (d_n\ v).lex \doteq e.d_n \end{array} \tag{4.7}$$

4.4. Valency

The next key concept of DG that we reformulate in XDG is valency. Its application to linguistics reaches back to (Peirce 1898), where valency describes the set of dependents of a lexical head, i.e., its argument structure. For XDG, we adopt a broader notion of valency, in which it lexically specifies the incoming and outgoing edges of the nodes.

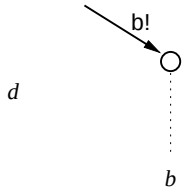
4.4.1. Fragments

We explain XDG valency using the intuitive metaphor of *fragments*. An XDG fragment is simply a lexical specification of the incoming and outgoing edges of a node. We show a picture of an example fragment below:



(4.8)

The fragment is defined for dimension *d*. The anchor of the fragment is the word *a*, and it licenses at most one incoming edge labeled a, at most one outgoing edge labeled a, and precisely one outgoing edge labeled b. It licenses no other incoming and outgoing edges. Here is a second example:

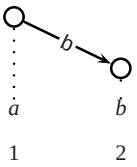


(4.9)

This fragment with anchor *b* requires precisely one incoming edge labeled b, and licenses no other incoming and outgoing edges.

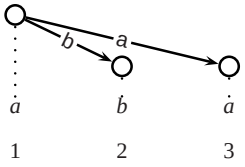
4.4.2. Configuration

We call the arrangement of fragments into graphs *configuration*. For instance, we can arrange the two fragments (4.8) and (4.9) into the graph below:



(4.10)

However, there is no way to arrange the fragments into the following graph:



(4.11)

This graph is not well-formed according to fragment (4.8), since node 3 does not have the obligatory outgoing edge labeled b.

The string language of the grammar resulting from the two fragments is the set of words with equally many *a*s and *b*s, which we call EQAB:

Language 1 (EQAB)**.**

$$\text{EQAB} \quad = \quad \{w \in (a \cup b)^+ \mid |w|_a = |w|_b\} \tag{4.12}$$

Why is this so?

1. The *a*s are arranged in a chain: each *a* must have at most one incoming edge labeled a, and at most one outgoing edge labeled a to the next *a*.
2. The number of *a*s and *b*s is always the same: the fragment for *a* (4.8) requires precisely one outgoing edge labeled b to a *b*, and the fragment for *b* (4.9) ensures that *b* cannot become the root (which excludes the string containing only *b*).

4.4.3. Valency Predicates

We capture fragments in XDG using a set of predicates called valency predicates, which we define given a dimension d, a node v and an edge label l:

- License no incoming edge labeled l for v:

$$in0_d \quad = \quad \lambda v.\, \lambda l.\quad \neg \exists v' : v' \xrightarrow{l}_d v \tag{4.13}$$

- Requiring precisely one incoming edge labeled l for v:

$$in1_d \quad = \quad \lambda v.\, \lambda l.\quad \exists^1 v' : v' \xrightarrow{l}_d v \tag{4.14}$$

- License at most one incoming edge labeled l for v:

$$in0or1_d \quad = \quad \lambda v.\, \lambda l.\quad (in0_d\ v\ l) \vee (in1_d\ v\ l) \tag{4.15}$$

For the outgoing edges, the three predicates *out0*, *out1* and *out0or1* are defined analogously.

4.4.4. Valency Principle

The *Valency principle* combines the valency predicates with lexicalization. The idea is to model fragments using the two lexical attributes *in* for the licensed incoming edges, and *out* for the licensed outgoing edges. Given a type of edge labels L, the type of *in* and *out* is $valency(L) = vec(L, !, ?, *, 0)$, i.e., a vector used to map edge labels to *cardinalities*, which restrict the number of edges with this label. The cardinalities are interpreted as follows:

- !: precisely one edge
- ?: at most one edge
- $*$: arbitrary many edges
- 0: no edges

For example, the following lexical description represents fragment (4.8):

$$\left\{ \begin{array}{l} word = a \\ \quad \text{ID} = \left\{ \begin{array}{l} in = \{\mathsf{a}=?, \mathsf{b}=0\} \\ out = \{\mathsf{a}=?, \mathsf{b}=!\} \end{array} \right\} \end{array} \right\} \tag{4.16}$$

And the following fragment (4.9):

$$\left\{ \begin{array}{l} word = b \\ \quad \text{ID} = \left\{ \begin{array}{l} in = \{\mathsf{a}=0, \mathsf{b}=!\} \\ out = \{\mathsf{a}=0, \mathsf{b}=0\} \end{array} \right\} \end{array} \right\} \tag{4.17}$$

As for convenience, we allow to omit the $=$ signs between labels and cardinalities and pairs with 0 cardinality, we can abbreviate e.g. (4.16) as:

$$\left\{ \begin{array}{l} word = a \\ \quad \text{ID} = \left\{ \begin{array}{l} in = \{\mathsf{a}?\} \\ out = \{\mathsf{a}?, \mathsf{b}!\} \end{array} \right\} \end{array} \right\} \tag{4.18}$$

We can now turn to the definition of the Valency principle. Note that we do not need to stipulate any constraint for cardinality $*$, as it stands for arbitrary many edges.

Principle 6 (Valency)**.** *Given a dimension d, the Valency principle is defined as:*

$$\begin{array}{l} valency_d = \forall v : \forall l : \\ \begin{array}{llcll} (d\ v).lex.in.l & \doteq & 0 & \Rightarrow & in0_d\ v\ l \quad \wedge \\ (d\ v).lex.in.l & \doteq & ! & \Rightarrow & in1_d\ v\ l \quad \wedge \\ (d\ v).lex.in.l & \doteq & ? & \Rightarrow & in0or1_d\ v\ l \quad \wedge \\ (d\ v).lex.in.l & \doteq & 0 & \Rightarrow & out0_d\ v\ l \quad \wedge \\ (d\ v).lex.in.l & \doteq & ! & \Rightarrow & out1_d\ v\ l \quad \wedge \\ (d\ v).lex.in.l & \doteq & ? & \Rightarrow & out0or1_d\ v\ l \end{array} \end{array} \tag{4.19}$$

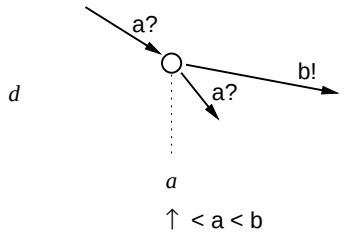
4.5. Order

The next key concept of DG is order.

4.5.1. Ordered Fragments

We begin with extending the fragments of the previous section with a local order on the daughters of the node. We impose this order indirectly by a strict partial order[1] on the set of edge labels of the daughters, and call the extended fragments *ordered fragments*.

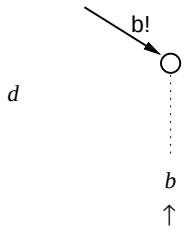
Here is an example ordered fragment:



(4.20)

[1] Strict partial orders are binary relations which are 1) irreflexive, 2) asymmetric and 3) transitive.

The fragment is defined for dimension d. It extends fragment (4.8) with the order $\uparrow < \mathsf{a} < \mathsf{b}$ on the set of edge labels, where $\uparrow$ is a special additional label representing the anchor of the fragment, which we draw directly below the anchor by convention. The meaning of the fragment is that the anchor must always precede the daughters with edge label a, and those must in turn precede the daughters with edge label b. Here is a second example:

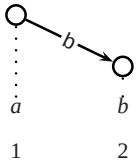


(4.21)

where nothing is ordered since the fragment does not license any outgoing edges.

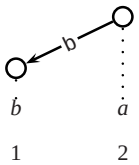
4.5.2. Ordered Configuration

Ordered fragments allow us to extend the notion of configuration: now, a well-formed configuration must not only satisfy the constraints on the incoming and outgoing edges, but also the order on the set of edge labels. We call this extended notion of configuration *ordered configuration*. For example, we can arrange the two fragments (4.20) and (4.21) into the well-formed graph below:



(4.22)

However, the following ordered configuration is not well-formed since it violates the order of fragment (4.20), requiring that the anchor must precede its b-daughter, not follow it:



(4.23)

4.5.3. Projectivity

If we require that the fragments (4.20) and (4.21) can only be configured into trees, the string language seems to be that of *n a*s followed by *n b*s, which we call ANBN:

Language 2 (ANBN)**.**

$$\text{ANBN} \quad = \quad \{w \in a^n b^n \mid n \geq 1\} \tag{4.24}$$

But this is not the case. Figure 4.1 shows a counter-example: for all nodes, the anchors do precede the a-daughters, which in turn do precede the b-daughters, yet not all *a*s precede all *b*s.

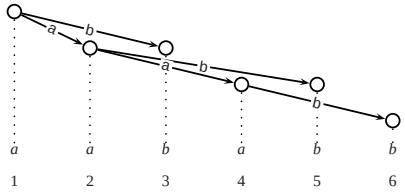


Figure 4.1.: Non-projective analysis

The problem is that we have to rule out non-projective analyses: when we order the daughters of a node, we need to ensure that the yields of the daughters must be continuous, such that it becomes impossible e.g. for the leftmost *b* (node 3) in Figure 4.1 to interrupt the sequence of *a*s. We can do this by applying the *Projectivity principle* (principle 4).

4.5.4. Order Principle

Given a domain of edge labels L, we lexicalize the strict partial order on the edge labels and the anchor of the ordered fragment by the lexical attribute *order*, a set of pairs of edge labels and the *anchor label*, i.e., having the type $set(tuple(L\,|\,\{\uparrow\}, L\,|\,\{\uparrow\}))$. For example, the following lexical description represents the ordered fragment (4.20):

$$\left\{ \begin{array}{l} word = a \\ \quad \textsc{lp} = \left\{ \begin{array}{r} in = \{\mathsf{a?}\} \\ out = \{\mathsf{a?}, \mathsf{b!}\} \\ order = \{(\uparrow, \mathsf{a}), (\uparrow, \mathsf{b}), (\mathsf{a}, \mathsf{b})\} \end{array} \right\} \end{array} \right\} \tag{4.25}$$

The *Order principle* is then stated for each node v and all pairs (l, l') in the lexicalized strict partial order of v:

1. If l is the anchor label and l' an edge label, then v must precede its l' daughter.
2. If l' is the anchor label and l an edge label, then v must follow its l daughter.
3. If l and l' are edge labels, then the l daughter of v must precede the l' daughter.

Principle 7 (Order)**.** *We define the Order principle given a dimension d as:*

$$\begin{array}{l} order_d = \\ \forall v : \forall (l, l') \in (d\ v).lex.order : \\ \begin{array}{lrcl} \forall v' : & l \doteq \uparrow & \wedge & v \xrightarrow{l'}_d v' \quad \Rightarrow \quad v < v' \quad \wedge \\ \forall v' : & l' \doteq \uparrow & \wedge & v \xrightarrow{l}_d v' \quad \Rightarrow \quad v' < v \quad \wedge \\ \forall v', v'' : & v \xrightarrow{l}_d v' & \wedge & v \xrightarrow{l'}_d v'' \quad \Rightarrow \quad v' < v'' \end{array} \end{array} \tag{4.26}$$

4.6. Agreement

The idea behind agreement is to ensure for certain nodes that they "agree" with certain dependents, e.g. for finite verbs to agree with their subjects. To this end, we assign to each node:

- a set of *agreement tuples* (e.g. consisting of person and number) by the lexical attribute *agrs*
- a set of edge labels by the lexical attribute *agree*
- an agreement tuple from *agrs* by the non-lexical attribute *agr*

Then, we model agreement using two principles: the Agr principle and the Agreement principle.

4.6.1. Agr Principle

The *Agr principle* expresses the constraint that for each node on a given dimension d, the value of *agr* must be an element of *agrs*.

Principle 8 (Agr)**.**

$$agr_d \quad = \quad \forall v : (d\ v).agr \in (d\ v).lex.agrs \tag{4.27}$$

4.6.2. Agreement Principle

The *Agreement principle* constrains each edge from v to v' labeled l on d such that if l is in the lexically specified set *agree* for v, then the values of *agr* of v and of v' must be equal.

Principle 9 (Agreement)**.**

$$\begin{aligned} &agreement_d = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_d v' \ \wedge \ l \in (d\ v).lex.agree \Rightarrow (d\ v).agr \doteq (d\ v').agr \end{aligned} \tag{4.28}$$

As an example, the analysis in Figure 4.2 is well-formed according to the Agr principle and the Agreement principle:

1. For nodes 1 and 2, the value of *agr*, is an element of *agrs*.
2. As required by *agree*, node 2 agrees with its subject, i.e., its *agr* value equals the *agr* value of node 1,

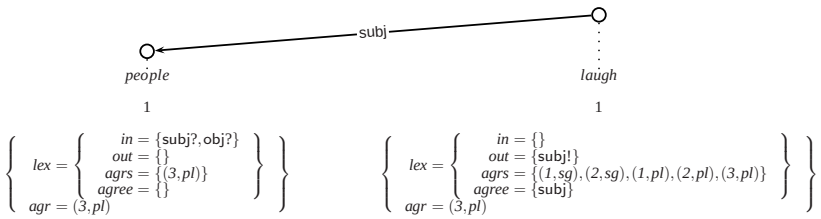


Figure 4.2.: Agr and Agreement principle: well-formed analysis

The example analysis in Figure 4.3 is not well-formed. The Agr principle is satisfied: for nodes 1 and 2, the value of *agr* is an element of *agrs*. The Agreement principle is however violated: node 2 does not agree with its subject: its *agr* value $(3, pl)$ does not equal the *agr* value $(3, sg)$ of node 1, as required by *agree*.

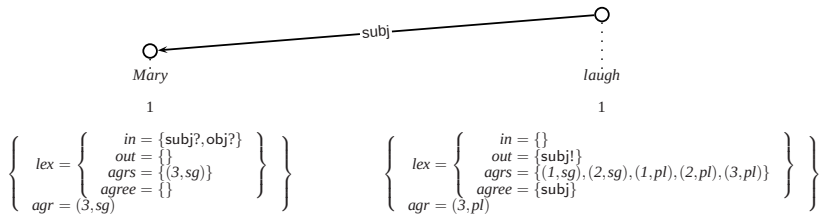


Figure 4.3.: Agr and Agreement principle: ill-formed analysis

4.7. Linking

The example grammar in section 2.2.4 made use of the LinkingEnd principle and the LinkingMother principle to constrain the syntactic realization of semantic arguments. These principles are instances of an entire family of principles called *linking principles*, whose purpose is to "link" together pairs of dimensions. The idea behind the linking principles is, given an edge from a node v to a node v' labeled l on d_1, to constrain the path to v' on another dimension d_2. Linking principles are lexicalized by attributes on a third interface dimension d_3, which acts as an interface.

4.7.1. LinkingEnd Principle

The *LinkingEnd principle* constrains the incoming edge label of v' on d_2, which we call the endpoint of the path to v' on d_2 (hence the name LinkingEnd). It is lexicalized by the attribute *linkEnd*, whose type is a vector used to map edge labels on d_1 to sets of edge labels on d_2. The principle is stated as follows. If for an edge from v to v' labeled l on d_1, the value of *linkEnd* for v and l on d_3 is non-empty, then for at least one edge label l' in this set, there must be an edge from any node v'' to v' on d_2 labeled l'. Figure 4.4 shows an illustration.

Principle 10 (LinkingEnd)**.** *Given three dimensions d_1, d_2 and d_3, the LinkingEnd principle is defined as:*

$$\begin{aligned} &linkingEnd_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} v' \ \wedge \ (d_3\ v).lex.linkEnd.l \neq \emptyset \Rightarrow \\ &\exists l' : l' \in (d_3\ v).lex.linkEnd.l \ \wedge \ \exists v'' : v'' \xrightarrow{l'}_{d_2} v' \end{aligned} \tag{4.29}$$

4.7.2. LinkingMother Principle

The *LinkingMother principle* constrains v' to be the mother of v on d_2. It is lexicalized by the attribute *linkMother*, whose type is a set of edge labels on d_1. The principle is stated as follows. If for an edge from v to v' labeled l on d_1, l is in the set *linkMother* of v on d_3, then v' must be the mother of v on d_2. Figure 4.5 shows an illustration.

Principle 11 (LinkingMother)**.** *Given three dimensions d_1, d_2 and d_3, we define the LinkingMother principle as:*

$$\begin{aligned} &linkingMother_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} v' \ \wedge \ l \in (d_3\ v).lex.linkMother \Rightarrow v' \longrightarrow_{d_2} v \end{aligned} \tag{4.30}$$

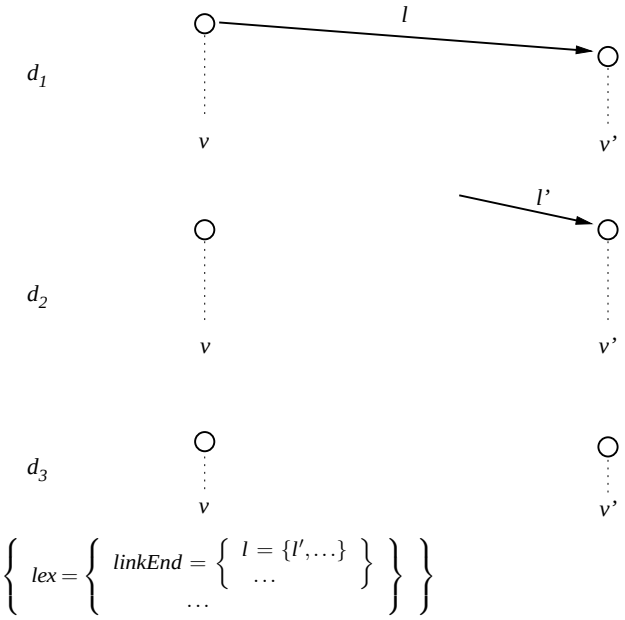


Figure 4.4.: LinkingEnd illustration

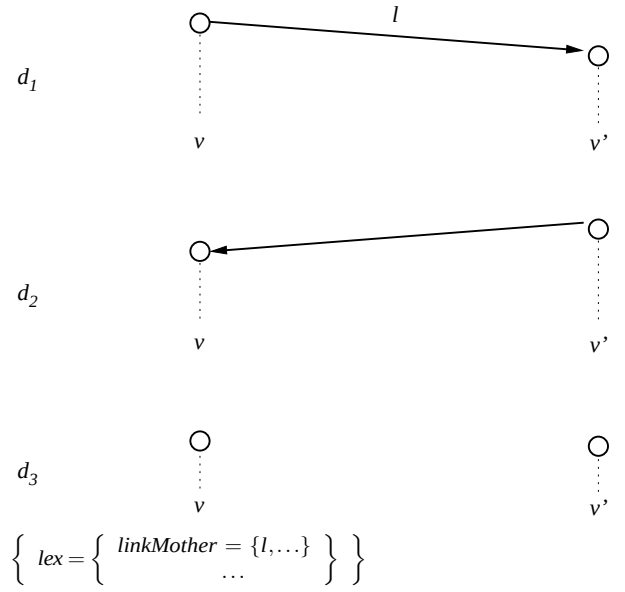


Figure 4.5.: LinkingMother illustration

4.8. Summary

In this chapter, we have shown how to reformulate the key concepts of dependency grammar as XDG principles. These principles, and their use on multiple dimensions, will form the basis of our investigation of the expressivity and computational complexity of XDG in the following chapters, and then of our modeling of natural language in part III.

5. Expressivity

In this chapter, we investigate the expressivity of XDG. We begin with the relation between XDG and Context-Free Grammar (CFG). We prove that it is possible to transform every CFG, given that it does not generate the empty string, into an equivalent XDG. In the next step, we show that by using multiple dimensions, XDG can also describe languages that fall outside context-freeness, including languages which are benchmarks for coping with natural language syntax.

5.1. XDG and Context-Free Grammar

We begin this chapter by looking at the relation XDG and CFG. At the end of this section stands a proof showing that for every CFG, we can construct an XDG which licenses the same string language, i.e., which is weakly equivalent. In principle, this is nothing new: the first proofs showing that restricted versions of dependency grammar are weakly equivalent to CFG date back to (Hays 1964), (Gaifman 1965) and (Gross 1964). Nevertheless, the proof is new for XDG, and shall show that XDG is at least as expressive as CFG.

5.1.1. Context-Free Grammar

Definition 21 (Context-Free Grammar)**.** *A CFG G is defined by a set V of non-terminal symbols, a set Σ of terminal symbols, a set $R \subseteq V \times (V \cup \Sigma)^*$ of production rules and a start symbol $S \in V$:*

$$G = (V, \Sigma, R, S) \tag{5.1}$$

We write single uppercase Roman letters for non-terminal symbols, single lowercase Roman letters for terminal symbols, and lowercase Greek letters for sequences of terminal and non-terminal symbols. We write $A \to \alpha$ for $(A, \alpha) \in R$, and call the left component of a rule Left Hand Side (LHS), and the right Right Hand Side (RHS). Here is an example grammar describing language ANBN (section 4.5.3) of *n a*s followed by *n b*s.

$$G = (\{S, B\}, \{a, b\}, \{S \to aSB, S \to aB, B \to b\}, S) \tag{5.2}$$

5.1.2. Derivations and Derivation Trees

The string language $L(G)$ of a CFG G is the set of all strings derivable from the start symbol. In each derivation step, written $\alpha \Rightarrow \beta$, a non-terminal A is replaced by the RHS of a rule with

A on its LHS. We show an example derivation of the string *aabb* under the example grammar (5.2) below:

$$S \Rightarrow aSB \Rightarrow aaBB \Rightarrow aabB \Rightarrow aabb \tag{5.3}$$

Derivations impose a tree structure on the derived string called syntax tree or derivation tree. Figure 5.1 shows an example derivation tree, which represents the derivation in (5.3).

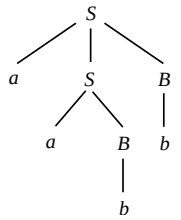


Figure 5.1.: Derivation tree for derivation (5.3)

5.1.3. Lexicalized Context-Free Grammar

In our transformation of CFGs into XDGs, we restrict ourselves to *Lexicalized Context-Free Grammar* (*LCFG*).

Definition 22 (Lexicalized Context-Free Grammar)**.** *In an LCFG, the RHS of each rule contains precisely one terminal symbol (*$1 \leq k \leq n$*):*

$$A \quad \rightarrow \quad B_1 \ldots B_k a B_{k+1} \ldots B_n \tag{5.4}$$

Every CFG G which does not generate the empty string can be brought into a weakly equivalent LCFG G', i.e., $L(G) = L(G')$. One method is to convert G to G' in *Greibach Normal Form* (*GNF*)[1]. However, the method of conversion is not our concern here.

5.1.4. Constructing an XDG from an LCFG

Using e.g. GNF, we can transform CFGs into weakly equivalent LCFGs. In this subsection, we proceed by showing that for every LCFG, we can construct a weakly equivalent XDG. We can then combine the two transformations to construct a weakly equivalent XDG from any CFG which does not generate the empty string. We first present the ideas behind the construction and an example, before we prove its correctness.

We construct the XDG from the LCFG using a grammar with one dimension called *derivation dimension* (abbreviated DERI). The derivation trees of the LCFG stand in the following correspondence to the models on DERI:

[1] GNF requires that the RHS of each rule starts with a terminal symbol, and is followed by a sequence of non-terminal symbols ($n \geq 0$):

$$A \quad \rightarrow \quad a B_1 \ldots B_n \tag{5.5}$$

- the non-terminal nodes in the derivation tree correspond to the nodes on DERI
- the labels of the non-terminal nodes in the derivation tree are represented by the incoming edge labels of the corresponding nodes on DERI[2]
- the terminal nodes in the derivation tree correspond to the words on DERI

Figure 5.2 shows an example DERI model, corresponding to the derivation tree displayed in Figure 5.1. For example, the non-root *S* node in the derivation tree corresponds to node 2 on DERI. The symbol *S* of the node in the derivation tree is represented by the incoming edge label on DERI, and the right *a* in the derivation tree corresponds to the word associated with node 2 on DERI.

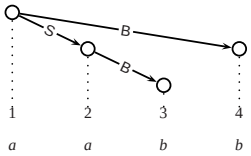
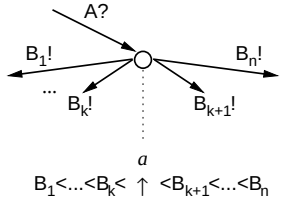


Figure 5.2.: DERI tree

The constructed XDG grammar uses the Tree, Projectivity, Valency and Order principles. We describe the lexical entries specifying the valency and order requirements by ordered fragments. Each rule $A \to B_1 \ldots B_k a B_{k+1} \ldots B_n$ ($1 \leq k \leq n$), given that A is the start symbol of the LCFG, corresponds to the following ordered fragment:

DERI

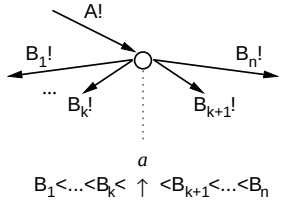


(5.6)

The anchor of the fragment is the terminal symbol a of the RHS of the LCFG rule. The fragment licenses at most one incoming edge labeled by the LHS of the rule, i.e., A. It requires precisely one outgoing edge for each non-terminal on the RHS of the rule, i.e., $B_1, \ldots, B_n$, and preserves the order of the non-terminals and the anchor on the RHS of the rule ($B_1 < \ldots < B_k < a < B_{k+1} < \ldots < B_n$).

If A is not the start symbol of the LCFG, then it can never be the root of the derivation tree, and hence it must have an incoming edge. This is expressed in the following ordered fragment:

DERI



(5.7)

[2]Except for the root.

(5.7) is equivalent to (5.6), except that it requires precisely one incoming edge labeled A (A!) instead of licensing at most one (A?).

However, there is a caveat to the construction presented so far: it only works for grammars where the RHSs of the rules do not contain multiple occurrences of the same non-terminal. A counter-example is $A \to BaB$, where B occurs twice on the RHS. At this point, we are left with two choices:

1. Change the construction of the XDG, e.g. augmenting the edge labels with the positions of the non-terminals.

2. Change the LCFG to get an LCFG where for each rule, the RHS contains only at most one occurrence of the same non-terminal.

We take the second choice: before we construct an XDG from the LCFG, we change the LCFG to contain at most one occurrence of each non-terminal on the RHSs of its rules. This is easy:

1. We replace each rule where the RHS contains multiple occurrences of the same non-terminals by a rule in which we replace the repeated non-terminals by fresh ones. For example, $A \to aBBCCC$ becomes $A \to aBB'CC'C''$.

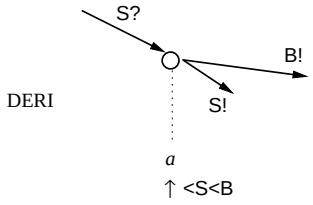
2. For each rule with one of the repeated non-terminals on its LHS, we introduce a new rule for each fresh non-terminal, where the fresh non-terminal replaces the repeated one. In our example, we introduce a new rule $B' \to \beta$ for each rule $B \to \beta$, and two new rules $C' \to \gamma$ and $C'' \to \gamma$ for each rule $C \to \gamma$.

As an example, we construct an XDG corresponding to the LCFG G in (5.2) above. The grammar contains no rule with more than one occurrence of the same non-terminal. Thus, we can directly proceed to construct the XDG. The set of words of the corresponding XDG grammar is $\{a, b\}$. The set of edge labels on DERI corresponds to the set of non-terminals:

$$\{\mathsf{S}, \mathsf{B}\} \tag{5.8}$$

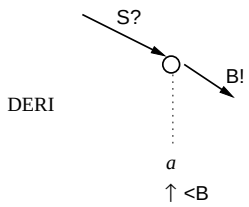
The three rules correspond to the following ordered fragments:

1. $S \to aSB$:



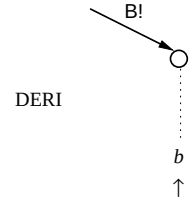
(5.9)

2. $S \to aB$:



(5.10)

3. $B \to b$:



(5.11)

For proving the correctness of the construction, we make use of McCawley's (1968) idea to view CFG as a description language for ordered, labeled trees.[3] McCawley describes the well-formedness conditions for derivation trees using so-called *node admissibility conditions*.

Definition 23 (Node Admissibility Conditions)**.** *Given an LCFG $G = (V, \Sigma, R, S)$, a node v satisfies G if either:*[4]

1. *v is a leaf node and is labeled with a terminal symbol.*
2. *v is an inner node with successors $v_1, \ldots, v_k, v', v_{k+1}, \ldots, v_n$ (in that order), and:*
   a) *v is labeled with A*
   b) *R contains rule $A \to B_1 \ldots B_k a B_{k+1} \ldots B_n$ ($1 \leq k \leq n$)*
   c) *v' is labeled with a*
   d) *each other successor v_i ($1 \leq i \leq n$) is labeled with B_i*

An ordered tree satisfies G if its root node is labeled with S and all of its nodes satisfy G.

McCawley's conditions carry over almost directly to our XDG construction. The differences between the CFG derivation trees and the XDG DERI trees are:

- DERI trees do not contain the terminal nodes of the derivation trees. Instead, each node is associated with the corresponding word by the node-word mapping.
- The edges of the DERI trees are labeled, not the nodes, as in the derivation trees. The node labels of the nodes in the derivation tree are modeled by the incoming edge label on DERI.

Proof. Considering these differences, we can adapt McCawley's node admissibility conditions for proving that our construction of XDGs from LCFGs is correct. Given an XDG G' constructed from an LCFG G, a node v on DERI satisfies G' if:

1. v is a terminal node associated to a word by the node-word mapping.
2. v is a node with successors $v_1, \ldots, v_n$ (in that order):
   a) if v is the root, it has no incoming edge, if it is not the root, its incoming edge is A

[3] This is also used as the starting point for the introduction of *Lexicalised Configuration Grammars* (*LCGs*) in (Grabowski, Kuhlmann & Möhl 2005).

[4] We have slightly adapted McCawley's conditions for CFG for LCFG.

b) if A is the start symbol of the underlying LCFG, the lexicon of G' contains the ordered fragment (5.6), otherwise if A is not the start symbol, the lexicon contains the ordered fragment (5.7)

c) v is associated with the anchor a of the fragment by the node-word mapping

d) the successors v_i ($1 \leq i \leq n$) have incoming edge label B_i ($1 \leq \mathsf{i} \leq n$)

A DERI analysis is always an ordered tree by the Tree principle and the Projectivity principle. A DERI analysis satisfies G' if all its nodes satisfy G'. □

5.2. Going Beyond Context-Freeness

Now that we know that XDG is at least context-free, we show that it is also perfectly able to handle languages which go beyond context-freeness. We begin with modeling the artificial language $a^n b^n c^n$, and proceed with two classical non-context-free benchmarks for grammar formalisms from natural language: *cross-serial dependencies* and *scrambling*.

5.2.1. $a^n b^n c^n$

The language of words formed by subsequent blocks of *a*s, *b*s and *c*s, is the prototypical example of a non-context-free language. We call it ANBNCN.

Language 3 (ANBNCN)**.**

$$\text{ANBNCN} \quad = \quad \{w \in a^n b^n c^n \mid n \geq 1\} \tag{5.12}$$

We model ANBNCN using two dimensions: Immediate Dominance (ID) and Linear Precedence (LP). The purpose of the ID dimension is to ensure that for each *a*, there is precisely one *b* and precisely one *c*. The models on ID are unordered trees, and the set of edge labels is $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. We relegate the ordering of the nodes to the LP dimension, whose models are ordered trees. More specifically, LP trees always have depth 1: the leftmost *a* is the root, which orders all the remaining nodes to its right. The set of edge labels on LP is $\{1, 2, 3\}$, where 1 corresponds to a, 2 to b and 3 to c. We show an example analysis in Figure 5.3.

The grammar uses the Tree and Valency principles on both ID and LP. The LP dimension in addition makes use of the Order principle. Thus, the lexicon of the grammar can be described using pairs of unordered and ordered fragments, where the unordered fragment specifies the lexical attributes of the Valency principle on ID, and the ordered fragment the lexical attributes of the Valency principle and the Order principle on LP. We call the pairs *fragment pairs*.

We start with the fragment pairs for nodes associated with word *a*. We make such nodes lexically ambiguous, behaving differently as a root and as a dependent. As a root, they are

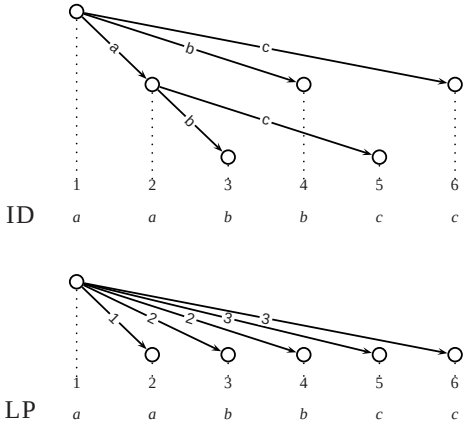
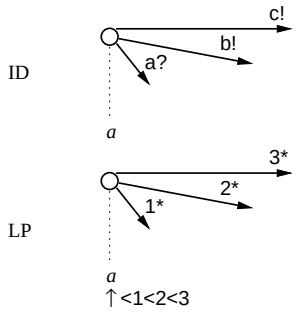


Figure 5.3.: ID/LP analysis

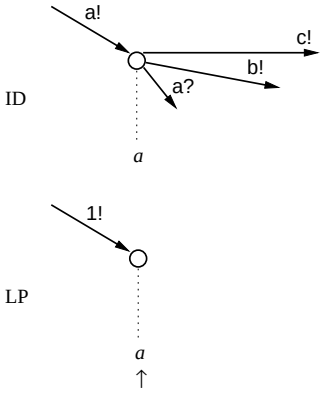
constrained by the following fragment pair:



(5.13)

This pair requires the node to be a root on both ID and LP as it does not license any incoming edges. As for the outgoing edges, on ID, it licenses at most one labeled a to the next *a*, and requires precisely one labeled b and one labeled c to ensure that there are equally many *a*s, *b*s and *c*s. On LP, it licenses arbitrary many outgoing edges labeled 1 (for the *a*s), 2 (for the *b*s) and 3 (for the *c*s). The root precedes all remaining *a*s, which in turn precede all *b*s which in turn precede all *c*s.

As a dependent, nodes with word *a* are constrained by the following fragment pair:

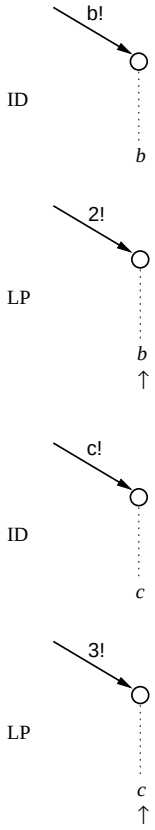


(5.14)

Here, on ID and LP, *a* must have precisely one incoming edge labeled a and 1, respectively. That is, the node cannot be the root. On ID, the outgoing edges are constrained as in the root

fragment pair (5.13) above to ensure an equal number of *a*s, *b*s and *c*s. On LP, it does not license any outgoing edges. As a result, all nodes whose mother is not the root node on ID must find a new mother on LP, and this new mother can only be the root *a*, since it is the only node on LP which licenses any outgoing edges.

For completeness, the fragment pairs for *b* and *c* are the following:



(5.15)

(5.16)

On both ID and LP, they require precisely one incoming edge labeled b and 2 (c and 3 for *c*). They do not license any outgoing edges, i.e., they must always be dependents of nodes associated with word *a*.

Notice that this grammar could easily be extended to languages with any finite number of letter blocks, e.g. $a^n b^n c^n d^n e^n$ etc., whereas interestingly, languages with more than four blocks cannot be modeled anymore using the *mildly context-sensitive* grammar formalisms of TAG and CCG (Shanker & Weir 1994).

5.2.2. Cross-Serial Dependencies

Cross-serial dependencies occur e.g. in Dutch (Bresnan, Kaplan, Peters & Zaenen 1983) and in Swiss German (Shieber 1985) subordinate sentences. The typical examples are so-called *hippo sentences* such as the following Dutch example:

(*omdat*) *ik Cecilia de nijlpaarden zag voeren*
(*that*) *I Cecilia the hippos saw feed* (5.17)
"(that) I saw Cecilia feed the hippos"

We show a dependency analysis of (5.17) in Figure 5.4. Here, the edge label det stands for “determiner” and vbse for “infinitival complement in base form”. As can be seen, hippo sentences are split into two parts:

1. The verbs on the right (here: *zag* and *voeren*) make up the so-called *verb cluster*. Here, each verbal head must precede its verbal dependents, hence in the example, *zag* must precede *voeren*.

2. The nominal dependents on the left make up the so-called *Mittelfeld*[5]. Here, each nominal head must follow the nominal dependents of the verbs higher up, so e.g. *nijlparden*, the object of *voeren*, must follow *ik* and *Cecilia*, the subject and object of *zag*, which is the mother of *voeren* and thus situated higher up.

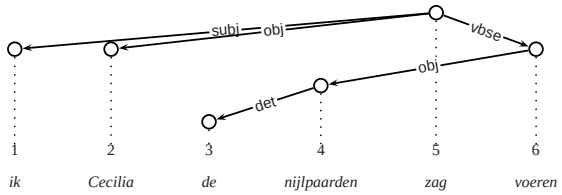


Figure 5.4.: Dependency analysis for *(omdat) ik Cecilia de nijlpaarden zag voeren*

To show how this phenomenon scales up, we give another example:

(*omdat*)	*ik*	*Cecilia*	*Henk*	*de*	*nijlpaarden*	*zag*	*helpen*	*voeren*
(*that*)	*I*	*Cecilia*	*Henk*	*the*	*hippos*	*saw*	*help*	*feed*

“(that) I saw Cecilia help Henk feed the hippos” (5.18)

of which we show a dependency analysis in Figure 5.5.

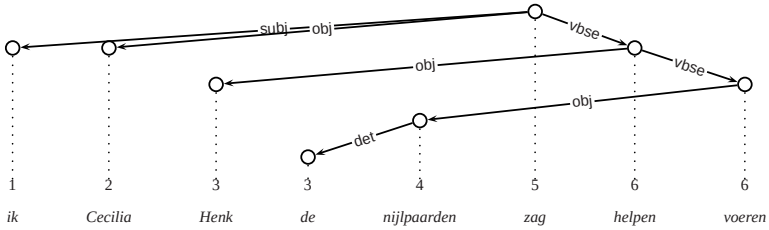


Figure 5.5.: Dependency analysis for *(omdat) ik Cecilia Henk de nijlpaarden zag helpen voeren*

The phenomenon gets its name from the series of crossing dependencies which it gives rise to, e.g. in Figure 5.5, the edge from *helpen* to *Henk* (crossing the projection edge of *zag*) and the edge from *voeren* to *de nijlpaarden* (crossing that of *zag* and *helpen*).

Now for simplicity, we assume that each verb has exactly one nominal argument and model cross-serial dependencies by the indexed language CSD.[6]

[5]The term is borrowed from German descriptive linguistics (Herling 1821), (Erdmann 1886).

[6]The indices are not part of the terminal alphabet, which is simply $\{n, v\}$.

Language 4 (CSD)**.**

$$\text{CSD} \quad = \quad \{n^{[1]} \dots n^{[k]} v^{[1]} \dots v^{[k]} \mid k \geq 1\} \tag{5.19}$$

The string language of CSD is $\{n^k v^k \mid k \geq 1\}$, i.e., k nouns followed by k verbs. Each index (in superscript) pairs exactly one n and one v, reflecting that the n is an argument of the v. CSD is not context-free (Shieber 1985), but can be handled by mildly context-sensitive grammar formalisms like TAG and CCG. In fact, cross-serial dependencies are one of the primary reasons for the introduction of such grammar formalisms with a higher expressivity than CFG.

In XDG, we model CSD using two dimensions, ID and LP, similarly as for ANBNCN: the models of ID are unordered trees, whereas the models of LP are ordered and projective trees. On ID, we ensure that for each verb, there is a corresponding noun. On LP, we order the nouns and verbs. For the verbs, we require that they follow the nouns and that verbal heads precede their verbal dependents. For the nouns, we require the additional constraint that each n-dependent of a verb node v must follow the n-dependents of the verbs above v. We realize this constraint by the *CSD principle*, where we instantiate d with ID, and show an example ID/LP analysis of *nnnvvv* in Figure 5.6.

Principle 12 (CSD)**.** *Given a dimension d, the CSD principle is defined as:*

$$\begin{aligned} &csd_d = \forall v, v' : \\ &v \xrightarrow{\mathsf{n}}_d v' \Rightarrow \forall v'', v''' : v'' \rightarrow_d^+ v \wedge v'' \xrightarrow{\mathsf{n}}_d v''' \Rightarrow v''' < v' \end{aligned} \tag{5.20}$$

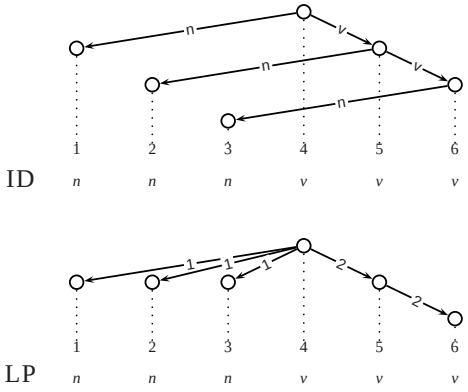


Figure 5.6.: ID/LP analysis for string *nnnvvv* (CSD grammar)

Contrary to ANBNCN, we need an additional constraint to synchronize the two dimensions. Otherwise, the dominance relations of the verbs in the verb cluster on ID are not preserved on LP, giving rise to ill-formed analyses. An example is shown in Figure 5.7, where on ID, the second noun (node 2) is a dependent of the third verb (node 6), and not, as it should, of the second verb (node 5). To rule out such analyses, we introduce a new principle called *Climbing principle*, which postulates that the dominance relation on LP must be a subset of that on ID. In our grammar, we instantiate d_1 with LP and d_2 with ID.

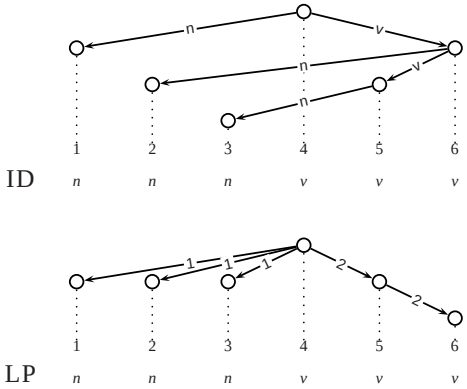


Figure 5.7.: Ill-formed ID/LP analysis for string *nnnvvv* (CSD grammar)

Principle 13 (Climbing)**.** *Given two dimensions d_1 and d_2, the Climbing principle is defined as:*

$$climbing_{d_1,d_2} = \forall v, v' : \; v \rightarrow^{+}_{d_1} v' \Rightarrow v \rightarrow^{+}_{d_2} v' \tag{5.21}$$

The principle gets its name from the metaphor that nodes, in this case the nouns, are allowed to "climb up" from their position on dimension d_2 (here: ID) to a higher position on d_1 (LP). For example, in Figure 5.6, the third noun (node 3) climbs up as follows: it is a dependent of the third verb (node 6) on ID, and climbs up to become a dependent of the first verb (node 4) on LP. Figure 5.7 is ruled out by the Climbing principle since the third verb (node 6) does not climb up from LP to ID, but migrates down to become a dependent of the second verb (node 5).

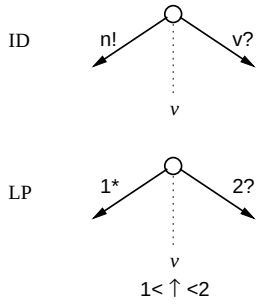
To sum up, the XDG for CSD makes use of the following principles:

- ID: Tree, Valency and CSD
- LP: Tree, Valency, Order
- ID and LP: Climbing

As for ANBNCN, we describe the lexical entries for the lexicalized Valency (on ID and LP) and Order principles (on LP only) by fragment pairs of an unordered fragment and an ordered fragment. Verbs (word *v*) are ambiguous between the following two lexical entries:

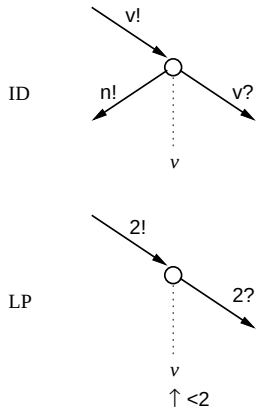
1. As a root, a verb *v* requires precisely one noun and at most one other verbal dependent on ID. On LP, *v* licenses arbitrary many nominal dependents (edge label 1) and at most one verbal dependent (2), where *v* must be positioned between the nominal dependents

on the left and the verbal dependent on the right:



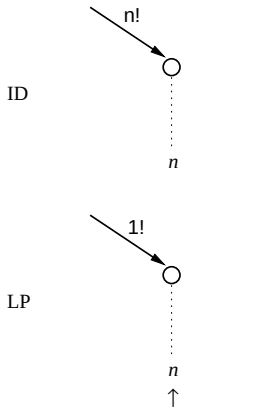
(5.22)

2. As a dependent (with incoming edge label v), a verb licenses the same outgoing edges as a root on ID. On LP (incoming edge label 2), it does not take any nominal dependents but only at most one verbal dependent, which must follow the verb:



(5.23)

Nouns must be dependents with incoming edge label *n* on ID and 1 on LP, and do not license any outgoing edges:



(5.24)

5.2.3. Scrambling

Subordinate sentences in standard German have a similar structure as in Dutch, but there are two differences:

1. The order of the verbs in the verb cluster is reversed: dependents precede their heads instead of following them.

2. The nominal dependents can occur in any permutation.

Here is an example:

(*dass*) *ein Mann Cecilia die Nilpferde füttern sah*
(*that*) *ein Mann Cecilia the hippos feed saw* (5.25)
"(that) a man saw Cecilia feed the hippos"

where interestingly, the other possible permutations of the nominal arguments in the Mittelfeld are also grammatical (although some are marginal). We show a dependency analysis of (5.25) in Figure 5.8, and of one of its permutations in Figure 5.9.

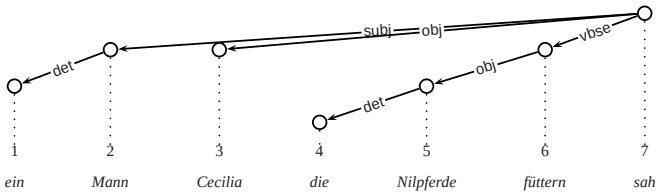


Figure 5.8.: Dependency analysis for *(dass) ein Mann Cecilia die Nilpferde füttern sah*

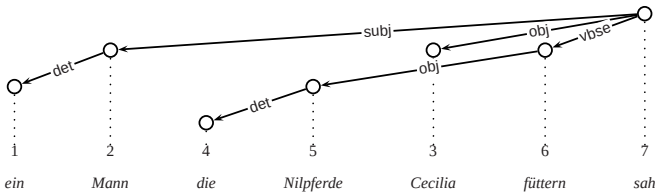


Figure 5.9.: Dependency analysis for *(dass) ein Mann die Nilpferde Cecilia füttern sah*

If we ignore the different ordering of the verbs for simplicity and leave it as in the cross-serial case (verbal dependents follow their heads), and assume that each verb has exactly one overt nominal argument, we can model scrambling with the indexed language SCR taken from (Becker, Rambow & Niv 1992).

Language 5 (SCR)**.**

$$\text{SCR} \quad = \quad \{\sigma(n^{[1]},\ldots,n^{[k]})v^{[1]}\ldots v^{[k]} \mid k \geq 1 \textit{ and } \sigma \textit{ a permutation}\} \tag{5.26}$$

The string language of SCR is the same as of CSD: $\{n^k v^k \mid k \geq 1\}$, and each index in SCR again pairs exactly one *n* and one *v*, reflecting the fact that *n* is an argument of the *v*.

For modeling SCR, we can reuse the same grammar as for CSD above, with the only exception that we leave out the CSD principle to free the order of the nominal arguments in the Mittelfeld. Becker et al. (1992) prove that no formalism in the class of *Linear Context-Free Rewriting Systems* (*LCFRS*) (Weir 1988) can model SCR, where LCFRS includes TAG, CCG and local *Multi-Component TAG* (*MC-TAG*) also introduced in (Weir 1988). So interestingly, what we did was to remove a constraint from the grammar for CSD, which is included in the LCFRS class, to get a grammar for SCR which is not included in LCFRS.

5.3. Summary

In this chapter, we have investigated the expressivity of XDG. We have proven that XDG is more expressive than context-free grammar by first translating CFGs into equivalent XDGs, and then showing that we can use XDG to model languages which go beyond context-freeness (ANBNCN, CSD and SCR). The XDG grammars for the benchmarks languages CSD and SCR demonstrated that XDG can handle complicated word order phenomena in natural language in an elegant way, which is substantiated by the elegant account of German word order in (Duchier & Debusmann 2001) and (Debusmann 2001), extended in (Bader et al. 2004). We have not found an upper bound to XDG's expressivity, but conjecture that it is at least mildly context-sensitive, i.e., that it at least includes TAG and CCG. Evidence for this is the encoding of TAG into XDG proposed (but not proven) in (Debusmann, Duchier, Kuhlmann & Thater 2004). We must leave a proof of this conjecture to future work.

6. Computational Complexity

After investigating the expressivity of XDG, we are interested in the price we have pay for it in terms of computational complexity. Therefore, in this chapter, we will prove the lower bound of the complexity of two kinds of recognition problems.

6.1. Recognition Problems

Following (Trautwein 1995), we distinguish two kinds of recognition problems: the *universal recognition problem* and the *fixed recognition problem*.

Definition 24 (Universal Recognition Problem)**.** *Given a pair (G,s) where G is a grammar and s a string, is s in $L(G)$?*

Definition 25 (Fixed Recognition Problem)**.** *Let G be a* fixed *grammar. Given a string s, is s in $L(G)$?*

6.2. Fixed Recognition Problem

We prove that the fixed membership problem is NP-hard by reduction of the NP-complete SAT problem.

6.2.1. Satisfiability Problem

SAT is the problem of deciding whether a formula in propositional logic has an assignment under which it evaluates to true.

Definition 26 (Propositional Formula)**.**

$$\begin{array}{rcll} f ::= & X,Y,Z,\ldots & & \textit{variable} \\ | & 0 & & \textit{false} \\ | & f_1 \Rightarrow f_2 & & \textit{implication} \end{array} \tag{6.1}$$

The reduction of SAT proceeds as follows.

6.2.2. Input Preparation

In three steps, we transform the propositional formula f into a string s which is suitable as an input to the fixed recognition problem. We call the function performing these steps *prep*. For example, given the formula

$$(X \Rightarrow 0) \Rightarrow Y \tag{6.2}$$

the transformation is defined as:

1. We transform the formula into prefix notation:

$$\Rightarrow \Rightarrow X\ 0\ Y \tag{6.3}$$

2. A propositional formula can contain an arbitrary number of variables, yet the domain of words of an XDG grammar must be finite. To overcome this limitation, we adopt a unary encoding for the variables: we encode the first variable from the left of the formula (here: *X*) as *var I*, the second (here: *Y*) *var I I* etc. (6.3) then becomes:

$$\Rightarrow \Rightarrow \textit{var I 0 var I I} \tag{6.4}$$

3. To clearly distinguish the input string from the original propositional formula, we replace all implication symbols with the word *impl*:

$$\textit{impl impl var I 0 var I I} \tag{6.5}$$

All three steps are polynomial.

6.2.3. Models

We model the structure of the propositional formula using a dimension called *Propositional Logic* (abbreviation: PL). The models on PL are ordered trees, which we enforce by the Tree and Projectivity principles. For example, Figure 6.1 shows a PL analysis of (6.5). Here, the edge labels are arg1 and arg2 for the antecedent and the consequent of an implication, respectively, and bar for connecting the bars (word *I*) of the unary variable encoding. Below the words of the nodes, we display their attributes, which have the following type:

$$\left\{ \begin{array}{l} \textit{truth} : \mathsf{B} \\ \textit{bars} : \mathsf{V} \end{array} \right\} \tag{6.6}$$

where *truth* represents the truth value of the node and *bars* the number of bars (nodes with word *I*) below the node plus 1. For example, in Figure 6.1, the *bars* value of node 3, which has one bar (node 4) below it, is 2. The *bars* value of node 6, which has two bars (nodes 7 and 8) below it, is 3. The purpose of the *bars* attribute will be to aid establishing coreferences between variables. Its type is V for two reasons:

1. There are always less (or equally many) variables in a formula than nodes, since every encoded formula contains less (or equally many) variables than words, and hence, V always suffices to distinguish them.

2. We require the precedence predicate, which is only defined on the type V, to implement incrementation.

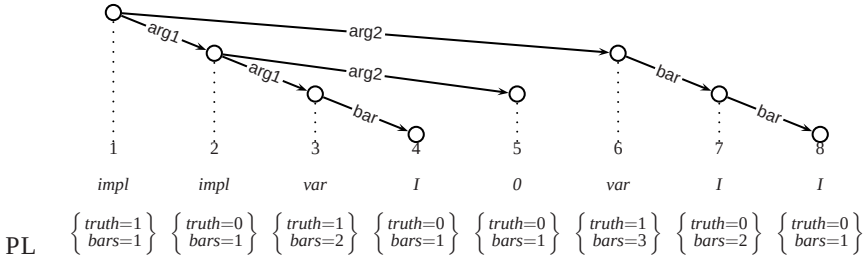
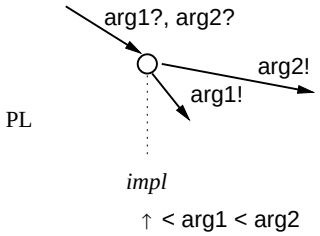


Figure 6.1.: PL analysis of the propositional formula $(X \Rightarrow 0) \Rightarrow Y$

6.2.4. Ordered Fragments

PL is additionally constrained by the Valency principle and the Order principle. We describe their lexical specifications with the following ordered fragments.

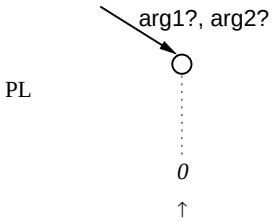
Implications. Implications, i.e., nodes with word *impl*, correspond to the following ordered fragment:



(6.7)

That is, an implication can have at most one incoming edge labeled arg1 or arg2. As for the outgoing edges, an implication requires precisely one labeled arg1 and one labeled arg2 for its own antecedent and consequent. The implication precedes its antecedent, and the antecedent in turn precedes the consequent.

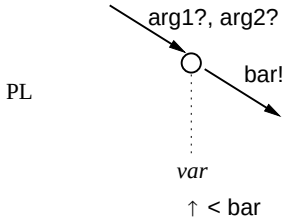
Zeros. Zeros are nodes with word *0*. They correspond to the following ordered fragment:



(6.8)

That is, a zero can either be the antecedent or the consequent of an implication, and must not have any outgoing edges.

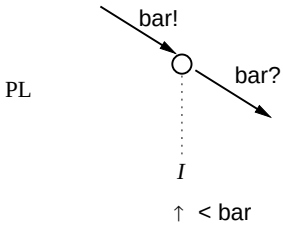
Variables. The following ordered fragment corresponds to variables, i.e., nodes with word *var*:



(6.9)

That is, a variable can either be the antecedent or the consequent of an implication, and requires precisely one outgoing edge labeled bar for the first bar below it. The variable must precede its bar-daughter.

Bars. Bars (nodes with word *I*) correspond to the following ordered fragment:



(6.10)

That is, a bar must have an incoming edge labeled bar, and can have at most one outgoing edge labeled bar. It must precede this potential bar-daughter.

6.2.5. Attributes

In this section, we constrain the attributes on PL, i.e., *truth* and *bars*. We capture these constraints using XDG predicates.

Roots. The truth value of the root of a PL analysis corresponds to the truth value of the analyzed formula. Thus, to model an assignment that evaluates to true, we must ensure that the *truth* attribute of the root node has value 1. We express this constraint in XDG with the following predicate:

$$\begin{array}{l} \mathit{plRoots} = \forall v : \\ \quad \neg\exists v' : v' \rightarrow_{\text{PL}} v \quad \Rightarrow \quad (\text{PL}\ v).\mathit{truth} \doteq 1 \end{array} \tag{6.11}$$

Implications. The *truth* value of implications equals the implication of the truth value of its arg1-daughter (the antecedent) and its arg2-daughter (the consequent). The *bars* value is irrelevant and hence we can pick an arbitrary value and set it to 1:

$$\begin{array}{l} \mathit{plImpls} = \forall v, v', v'' : \\ \quad (v \xrightarrow{\text{arg1}}_{\text{PL}} v' \wedge v \xrightarrow{\text{arg2}}_{\text{PL}} v'' \Rightarrow \\ \quad (\text{PL}\ v).\mathit{truth} \doteq ((\text{PL}\ v').\mathit{truth} \Rightarrow (\text{PL}\ v'').\mathit{truth})) \quad \wedge \\ \quad (\text{PL}\ v).\mathit{bars} \doteq 1 \end{array} \tag{6.12}$$

Zeros. The *truth* value of a zero is 0. Their *bars* value is irrelevant, i.e., we can arbitrarily set it to 1:

$$\begin{aligned} &plZeros = \forall v : \\ &\quad (W\ v) \doteq 0 \Rightarrow \\ &\quad (\text{PL}\ v).truth \doteq 0 \quad \wedge \\ &\quad (\text{PL}\ v).bars \doteq 1 \end{aligned} \tag{6.13}$$

Variables. The *truth* value of variables cannot be constrained a priori. Their *bars* value is the same as that of their bar daughter.

$$\begin{aligned} &plVars = \forall v, v' : \\ &\quad (W\ v) \doteq var \Rightarrow \\ &\quad v \xrightarrow{\text{bar}}_{\text{PL}} v' \Rightarrow (\text{PL}\ v).bars \doteq (\text{PL}\ v').bars \end{aligned} \tag{6.14}$$

Bars. The *truth* value of bars (word *I*) is irrelevant, and hence we can safely set it to an arbitrary value, here: 0. Their *bars* value is either 1 for the leaf bars (which do not have a daughter), or else the *bars* value of its daughter plus one:

$$\begin{aligned} &plBars = \forall v : \\ &\quad (W\ v) \doteq I \Rightarrow \\ &\quad (\text{PL}\ v).truth \doteq 0 \quad \wedge \\ &\quad \neg\exists v' : v \rightarrow_{\text{PL}} v' \Rightarrow (\text{PL}\ v).bars \doteq 1 \quad \wedge \\ &\quad (\forall v' : v \xrightarrow{\text{bar}}_{\text{PL}} v' \Rightarrow (\text{PL}\ v').bars < (\text{PL}\ v).bars \wedge \\ &\quad \neg\exists v'' : (\text{PL}\ v').bars < v'' \wedge v'' < (\text{PL}\ v).bars) \end{aligned} \tag{6.15}$$

Notice that the latter constraint actually increments the bar value, even though XDG does not provide us with any direct means to do that. The trick is to emulate incrementing using the precedence predicate.

6.2.6. Coreference

We can now establish coreferences between the variable occurrences. To this end, we stipulate that for each pair of variables (i.e., nodes v and v', both with word *var*) that if they have the same *bars* values, then their truth values must also be the same:

$$\begin{aligned} &plCoref = \forall v, v' : \\ &\quad (W\ v) \doteq var \wedge (W\ v') \doteq var \Rightarrow \\ &\quad (\text{PL}\ v).bars \doteq (\text{PL}\ v').bars \Rightarrow (\text{PL}\ v).truth \doteq (\text{PL}\ v').truth \end{aligned} \tag{6.16}$$

6.2.7. PL Principle

The *PL principle* ties the predicates defined in section 6.2.5 and section 6.2.6 together.

Principle 14 (PL)**.**

$$pl = plRoots \wedge plImpls \wedge plZeros \wedge plVariables \wedge plBars \wedge plCoref \tag{6.17}$$

6.2.8. Proof

Now we have gathered all the necessary ingredients for our NP-hardness proof.

Proof. Given a formula f according to definition 26 and the XDG grammar G defined in sections 6.2.3–6.2.7, f is satisfiable if and only if *prep* $f \in L(G)$. That is, SAT is reducible to the fixed recognition problem for XDG. As the reduction is polynomial, the fixed recognition problem for XDG is NP-hard. □

6.3. Universal Recognition Problem

The proof that the universal recognition problem is NP-hard as well falls out of the previous result.

Proof. The fixed recognition problem is an instance of the universal recognition problem where the grammar G is fixed. Hence, the universal recognition problem is at least as difficult as the fixed recognition problem, and as the latter is NP-hard, the universal recognition problem must also be NP-hard. □

A similar result has been obtained in (Koller & Striegnitz 2002), where they prove that the universal recognition problem for TDG, an instance of XDG, is NP-complete.

6.4. Summary

We have proven a lower bound for the complexity of the two kinds of recognition problems (fixed and universal) for XDG. Both are NP-hard. If we restrict the principles to the first order fragment of XDG, as is the case for all principles used in this thesis, the upper bound of model checking and thus of XDG recognition is in PSPACE. If we restrict ourselves to principles which can be tested in polynomial time, the overall complexity of the XDG recognition problems is NP-complete. For the principles used in this thesis, this is certainly the case, as we have implemented all of them as polynomially testable constraints in Mozart/Oz. We cannot see applications of XDG to natural language where this would not be the case. With even stronger restrictions, we hope that we can bring down the complexity to be polynomial, as e.g. for the grammar formalisms of TAG and CCG. We must leave finding these restrictions to future research.

Part II.

Implementation

7. The XDK—A Development Kit for XDG

We turn to the implementation of XDG, the XDG Development Kit (XDK) (Debusmann & Duchier 2006). In this chapter, we introduce its architecture and the XDK description language, which serves as a metagrammar for the description of grammars.

7.1. Architecture

The XDK consists of three main modules: the *metagrammar compiler*, the *constraint parser* and the *visualizer*, which are held together by the XDK description language and the *lattice functors*. This is illustrated in Figure 7.1.

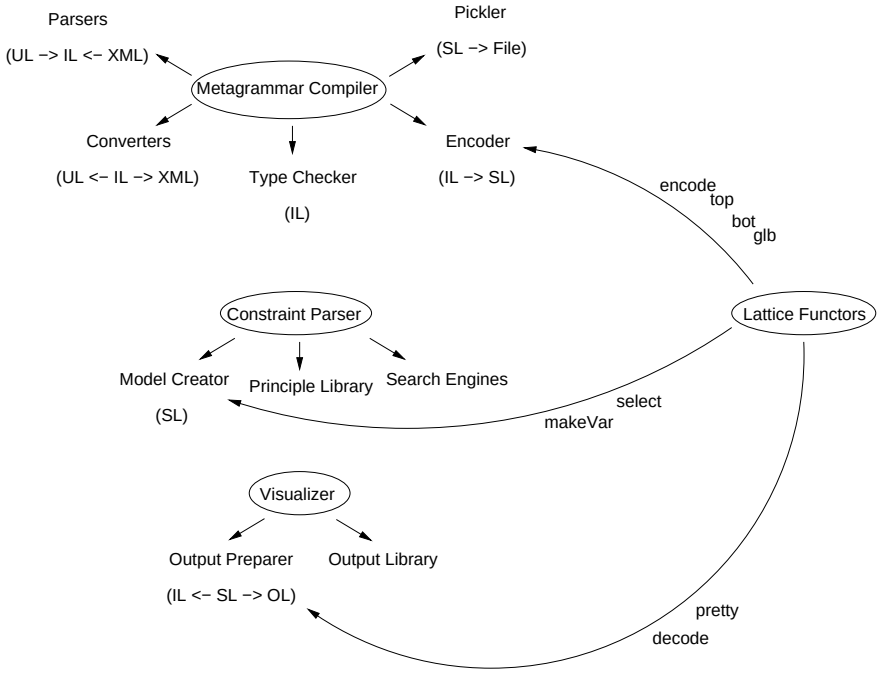


Figure 7.1.: Architecture of the XDK

7.1.1. Metagrammar Compiler

The purpose of the metagrammar compiler is to transform grammars in one of three supported concrete input syntaxes of the XDK description language into the *Solver Language* (*SL*) for

further processing in the constraint parser. The three syntaxes are:

- the *User Language* (*UL*), a custom syntax for handcrafted grammar development
- the *XML Language* (*XML*), based on XML, for automated grammar development in general
- the *Intermediate Language* (*IL*), based on Mozart/Oz syntax, for automated grammar development in Mozart/Oz, and for internal use in the XDK

For example, we show the definition of the lexical class `"fin"`, repeated from (2.11), in UL syntax in Figure 7.2, in XML syntax in Figure 7.3, and in IL syntax in Figure 7.4. The examples clearly show that contrary to the UL, due to their verbosity, the XML and IL syntaxes are not usable for writing grammars by hand—they are instead geared towards automated grammar development.

```
defclass "fin" Word Agrs {
  dim lex {word: Word}
  dim syn {in: {root?}
           out: {subj!}
           order: <subj "^" obj vinf adv>
           agrs: Agrs
           agree: {subj}}}
```

Figure 7.2.: Lexical class `"fin"` in UL syntax

The XDK implements parsers for UL and XML grammars into IL syntax for further internal use in the XDK, and converters to transform grammars from IL into either UL or XML syntax. The type checker performs static type checking on IL grammars for precise and early error detection, and the encoder encodes type checked IL grammars into the SL. Using the pickler, compiled SL grammars can be written into files. A detailed presentation of the metagrammar compiler can be found in appendix B.

7.1.2. Constraint Parser

Given a compiled SL grammar and an input string, the *model creator* of the constraint parser sets up a CSP, and augments it with the principles used in the grammar, which are taken from the extensible *principle library* of predefined principles. Constraint parsing amounts to searching for solutions of the CSP using one of the *search engines* of Mozart/Oz, e.g. the *Oz Explorer* (Schulte 1997), displayed in Figure 2.7, or *IOzSeF* (Tack 2002). The constraint parser will be explained in detail in chapter 8.

7.1.3. Visualizer

The visualizer transforms solutions (also partial ones) from the constraint parser into IL or *Output Language* (*OL*) syntax using the *output preparer*. The extensible *output library* provides functionality for actually visualizing the solutions, e.g. by displaying them as IL or OL terms, graphically using Tcl/Tk (as displayed in Figure 2.8), or by generating LaTeX code for them. We present the visualizer in detail in appendix C.

```
<classDef id="fin">
    <variable data="Word"/>
    <variable data="Agrs"/>
    <classConj>
        <classDimension idref="lex">
            <record>
                <feature data="word">
                    <variable data="Word"/>
                </feature>
            </record>
        </classDimension>
        <classDimension idref="syn">
            <record>
                <feature data="in">
                    <set>
                        <constantCard data="root" card="opt"/>
                    </set>
                </feature>
                <feature data="out">
                    <set>
                        <constantCard data="subj" card="one"/>
                    </set>
                </feature>
                <feature data="order">
                    <order>
                        <constant data="subj"/>
                        <constant data="^"/>
                        <constant data="obj"/>
                        <constant data="vinf"/>
                        <constant data="adv"/>
                    </order>
                </feature>
                <feature data="agrs">
                    <variable data="Agrs"/>
                </feature>
                <feature data="agree">
                    <set>
                        <constant data="subj"/>
                    </set>
                </feature>
            </record>
        </classDimension>
    </classConj>
</classDef>
```

Figure 7.3.: Lexical class `"fin"` in XML syntax

7.1.4. Lattice Functors

Lattice functors are Abstract Data Types (ADTs) corresponding to the types of the XDK description language. They include methods to obtain lattice top (`top`), lattice bottom (`bot`) and greatest lower bound (`glb`) of a type, methods to encode IL into SL syntax (`encode`), and to convert SL into IL (`decode`) or OL syntax (`pretty`). The lattice operations and the encode method are used in the metagrammar compiler, and the decode and pretty methods in the visualizer. The constraint parser makes use of the additional methods for the creation of constraint variables (`makeVar`) and for the efficient selection of values from a set of alternatives (`select`). The lattice functors are explained in detail in appendix A.

7.2. The XDK Description Language

The XDK is controlled by the XDK description language used for:

1. writing metagrammars:

```
elem(tag:classdef
     id:elem(tag:constant
             data:'fin')
     vars:[elem(tag:variable
                data:'Word')
           elem(tag:variable
                data:'Agrs')]
     body:elem(tag:conj
               args:[elem(tag:'class.dimension'
                          idref:elem(tag:constant
                                     data:'lex')
                          arg:elem(tag:record
                                   args:[elem(tag:constant
                                              data:'word')#
                                         elem(tag:variable
                                              data:'Word')]))
                     elem(tag:'class.dimension'
                          idref:elem(tag:constant
                                     data:'syn')
                          arg:elem(tag:record
                                   args:[elem(tag:constant
                                              data:'in')#
                                         elem(tag:set
                                              args:[elem(tag:constant
                                                         data:'root')#
                                                    elem(tag:'card.wild'
                                                         arg:'?' )])
                                         elem(tag:constant
                                              data:'out')#
                                         elem(tag:set
                                              args:[elem(tag:constant
                                                         data:'subj')#
                                                    elem(tag:'card.wild'
                                                         arg:'!' )])
                                         elem(tag:constant
                                              data:'order')#
                                         elem(tag:order
                                              args:[elem(tag:constant
                                                         data:'subj')
                                                    elem(tag:constant
                                                         data:'^')
                                                    elem(tag:constant
                                                         data:'obj')
                                                    elem(tag:constant
                                                         data:'vinf')
                                                    elem(tag:constant
                                                         data:'adv')])
                                         elem(tag:constant
                                              data:'agrs')#
                                         elem(tag:variable
                                              data:'Agrs')
                                         elem(tag:constant
                                              data:'agree')#
                                         elem(tag:set
                                              args:[elem(tag:constant
                                                         data:'subj')])])))])
```

Figure 7.4.: Lexical class `"fin"` in IL syntax

- metagrammar type definitions
- lexicon description
- principle instantiations

2. writing principles: principle type definitions
3. modeling multigraphs

We will develop the XDK description language using the UL concrete syntax for clarity.

7.2.1. Types

We begin by defining the types of the XDK description language, and showing how they are applied in the type definitions of metagrammars and principle definitions.

Definition 27 (Types)**.** *Given a set* A *of atoms,* DV *of dimension variables, and* TV *of type variables, we define the types* Ty *of the XDK description language as follows:*

$$
\begin{array}{lll}
\mathtt{a} \in \mathtt{A} & & \\
\mathtt{D} \in \mathtt{DV} & & \\
\mathtt{X} \in \mathtt{TV} & & \\
\mathtt{T} \in \mathtt{Ty} & ::= \quad \{\mathtt{a}_1 \ldots \mathtt{a}_n\} & \textit{finite domain } (n \geq 0) \\
& \quad | \quad \mathtt{string} & \textit{string} \\
& \quad | \quad \mathtt{int} & \textit{integer} \\
& \quad | \quad \mathtt{list}(\mathtt{T}) & \textit{list} \\
& \quad | \quad \mathtt{tuple}(\mathtt{T}_1 \ldots \mathtt{T}_n) & \textit{tuple } (n \geq 0) \\
& \quad | \quad \{\mathtt{a}_1 : \mathtt{T}_1 \ldots \mathtt{a}_n : \mathtt{T}_n\} & \textit{record } (n \geq 0) \\
& \quad | \quad \mathtt{set}(\mathtt{T}) & \textit{set } (\text{accumulative lattice}) \\
& \quad | \quad \mathtt{iset}(\mathtt{T}) & \textit{set } (\text{intersective lattice}) \\
& \quad | \quad \mathtt{card} & \text{cardinality} \\
& \quad | \quad \mathtt{label}(\mathtt{D}) & \textit{edge labels} \\
& \quad | \quad \mathtt{tv}(\mathtt{X}) & \textit{type variable}
\end{array}
\tag{7.1}
$$

Contrary to the types of XDG defined in section 3.2.1, the types of the XDK description language do not include functions, nor do they include types for booleans and nodes. In addition to the types of XDG, they include types for strings, integers, lists, tuples, three types of sets (`set(T)`, `iset(T)`, `card`)[1], edge labels and type variables. That is, the XDK description language is only equipped for the description of data. Functions, and hence also principles cannot be expressed. This is a deliberate design decision: we think that a grammar writer should not be bothered with the non-trivial issues surrounding the development of new principles using Mozart/Oz constraint programming, but should instead just pick them out from a library of predefined ones. Thus, the XDK is designed as a "toolkit" for grammar development, where the predefined principles act as "building blocks". Since the library is extensible, it can still be augmented by new principles if this is really needed.

Definition 28 (Notational Conveniences for Types)**.** *We introduce notational conveniences for:*

- *unions:*

$$\{\mathtt{a}_1 \ldots \mathtt{a}_k\} \,|\, \{\mathtt{a}_{k+1} \ldots \mathtt{a}_n\} \stackrel{\text{def}}{=} \{\mathtt{a}_1 \ldots \mathtt{a}_n\} \tag{7.2}$$

  for $0 \leq \mathtt{k} \leq \mathtt{n}$

- *vectors:*

$$\mathtt{vec}(\{\mathtt{a}_1 \ldots \mathtt{a}_n\}\ \mathtt{T}) \stackrel{\text{def}}{=} \{\mathtt{a}_1 : \mathtt{T} \ldots \mathtt{a}_n : \mathtt{T}\} \tag{7.3}$$

  for $\mathtt{n} \geq 0$

[1] As shown in appendix A, each type corresponds to a lattice. The three types do not differ on the level of types, but only in the lattices that correspond to them.

- *valencies:*

$$\texttt{valency(T)} \stackrel{\text{def}}{=} \texttt{map(T card)} \tag{7.4}$$

Definition 29 (Interpretation of Types)**.** *Given a set* A *of atoms and* D *of dimensions, we interpret the types as follows:*

- $\{\texttt{a}_1 \ldots \texttt{a}_n\}$ *as the set* $\{\texttt{a}_1, \ldots, \texttt{a}_n\} \uplus \{\top, \bot\}$*, where* $\top$ *and* $\bot$ *are added to act as top and bottom of the lattice corresponding to the type*
- `string` *as the set of all atoms plus* $\top$ *and* $\bot$*:* $\texttt{A} \uplus \{\top, \bot\}$*, i.e., the interpretation of strings can be infinite (if* A *is infinite), contrary to the interpretation of finite domains*
- `int` *as the set of all integers plus* $\top$ *and* $\bot$
- `list(T)` *for all* $n > 0$ *as the set of all n-tuples whose projections are elements of the interpretation of* T*, plus* $\top$ *and* $\bot$
- `set(T)` *and* `iset(T)` *as the power set of the interpretation of* T
- `card` *as the power set of the set of integers*
- $\texttt{tuple}(\texttt{T}_1 \ldots \texttt{T}_n)$ *as the set of all* n*-tuples whose* i*th projection is an element of the interpretation of* $\texttt{T}_i$ *(for* $1 \leq \texttt{i} \leq \texttt{n}$*)*
- $\{\texttt{a}_1 : \texttt{T}_1 \ldots \texttt{a}_n : \texttt{T}_n\}$ *as the set of all functions* f *with:*
  1. *Dom* $f = \{\texttt{a}_1, \ldots, \texttt{a}_n\}$
  2. *for all* $1 \leq \texttt{i} \leq \texttt{n}$*,* $f\ \texttt{a}_i$ *is an element of the interpretation of* $\texttt{T}_i$
- `label(D)` *as, given a binding of dimension variable* D *to dimension* d*, the type of edge labels on* d*.*
- `tv(X)` *as, given a binding of type variable* X *to type* T*, the interpretation of* T*.*

where the `label(D)` *and* `tv(X)` *can only be used in principle type definitions, not in metagrammar type definitions.*

Metagrammar Type Definitions. In the metagrammar type definitions, we use the types to specify for each dimension the types of edge labels (`deflabeltype`), lexical attributes (`defentrytype`) and non-lexical attributes (`defattrstype`). For convenience, in the metagrammar type definitions, a type T can be named a by writing:

$$\texttt{deftype a T} \tag{7.5}$$

The type can be referenced by just writing `a`. An example metagrammar type definition is shown below, repeated from (2.8):

```
deftype "syn.label" {root subj part obj vinf adv}
deftype "syn.label1" "syn.label" | {"^"}
deftype "syn.person" {"1" "2" "3"}
deftype "syn.number" {sg pl}
deftype "syn.agr" tuple("syn.person" "syn.number")

deflabeltype "syn.label"
defentrytype {in: valency("syn.label")
              out: valency("syn.label")
              order: set(tuple("syn.label1" "syn.label1"))
              agrs: iset("syn.agr")
              agree: set("syn.label")}
defattrstype {agr: "syn.agr"}
```
(7.6)

Principle Type Definitions. Each principle in the XDK principle library is accompanied with a *principle definition*. As principles are parametrized, principle definitions specify among other things described in chapter 8 below, the dimensions, the arguments and the types of the arguments that the principle abstracts over (principle type definition). For example, consider the following definition of the Valency principle (cf. principle 6 in chapter 4):

```
defprinciple "principle.valency" {
  dims {D}
  ...
  args {In: valency(label(D))
        Out: valency(label(D))}
  ...}
```
(7.7)

The principle abstracts over one dimension with the *dimension variable* `D`. It has two arguments, represented by the *argument variables* `In` and `Out`. The type of the two arguments is given by the expression `valency(label(D))`, which denotes a valency over the edge labels on the dimension denoted by dimension variable `D`.

As another example, consider the following definition of the Agreement principle (cf. principle 9 in chapter 4):

```
defprinciple "principle.agreement" {
  dims {D}
  args {Agr1: tv(X)
        Agr2: tv(X)
        Agree: set(label(D))}
  ...}
```
(7.8)

It abstracts over dimension `D` and has the three arguments `Agr1`, `Agr2` and `Agree`. The type of `Agr1` is not known beforehand—the only known fact is that it has the same type as `Agr2`. This is expressed using the same type variable `tv(X)` for both `Agr1` and `Agr2`.

7.2.2. Terms

In this section, we define the terms of the XDK description language and show how to apply them for the description of the lexicon and for the instantiation of principles.

Definition 30 (Terms)**.** *Given a set* A *of atoms, a set* N *of integers and a set* V *of variables, the terms* Te *of the XDK description language are defined as follows:*

$$
\begin{array}{lll}
\mathtt{a} \in \mathtt{A} & & \\
\mathtt{i} \in \mathtt{N} & & \\
\mathtt{v} \in \mathtt{V} & & \\
\mathtt{t} \in \mathtt{Te} & ::= & \mathtt{a} \qquad\qquad\qquad\qquad\qquad\qquad \textit{atom} \\
& | & \mathtt{i} \qquad\qquad\qquad\qquad\qquad\qquad \textit{positive integer} \\
& | & \{\mathtt{t}_1 \ldots \mathtt{t}_n\} \qquad\qquad\qquad\qquad\qquad \textit{set} \\
& | & \{\mathtt{i}_1 \ldots \mathtt{i}_n \ \ldots\} \qquad\qquad\qquad\qquad \textit{infinite set of integers} \\
& | & [\mathtt{t}_1 \ldots \mathtt{t}_n] \qquad\qquad\qquad\qquad\qquad \textit{list or tuple} \\
& | & \mathtt{v} \qquad\qquad\qquad\qquad\qquad\qquad \textit{variable} \\
& | & \{\mathtt{a}_1 : \mathtt{t}_1 \ldots \mathtt{a}_n : \mathtt{t}_n\} \qquad\qquad\qquad \textit{record specification} \\
& | & \{:\} \qquad\qquad\qquad\qquad\qquad\qquad \textit{empty record} \\
& | & \mathtt{c} \qquad\qquad\qquad\qquad\qquad\qquad \textit{cardinality} \\
& | & \{\mathtt{a}_1 \ \mathtt{c}_1 \ldots \mathtt{a}_n \ \mathtt{c}_n\} \qquad\qquad\qquad \textit{valency} \\
& | & \mathtt{top} \qquad\qquad\qquad\qquad\qquad\qquad \textit{lattice top} \\
& | & \mathtt{bot} \qquad\qquad\qquad\qquad\qquad\qquad \textit{lattice bottom} \\
& | & \mathtt{t}_1 \,\&\, \mathtt{t}_2 \qquad\qquad\qquad\qquad\qquad \textit{lattice greatest lower bound} \\
& | & \mathtt{t}_1 \,|\, \mathtt{t}_2 \qquad\qquad\qquad\qquad\qquad \textit{alternation} \\
& | & \$\,\mathtt{g} \qquad\qquad\qquad\qquad\qquad\qquad \textit{set generator} \\
& | & \langle \mathtt{t}_1 \ldots \mathtt{t}_n \rangle \qquad\qquad\qquad\qquad\qquad \textit{order} \\
& | & \mathtt{t}_1 \,@\, \mathtt{t}_2 \qquad\qquad\qquad\qquad\qquad \textit{concatenation} \\
& | & \mathtt{p} \qquad\qquad\qquad\qquad\qquad\qquad \textit{feature path} \\
& | & \mathtt{t} :: \mathtt{T} \qquad\qquad\qquad\qquad\qquad\qquad \textit{type annotation} \\
& | & (\mathtt{t}) \qquad\qquad\qquad\qquad\qquad\qquad \textit{brackets}
\end{array}
\tag{7.9}
$$

Cardinalities are a special syntax to describe sets of integers:

$$
\begin{array}{lll}
\mathtt{c} & ::= & ! \qquad\qquad\qquad \textit{precisely one } (\{1\}) \\
& | & ? \qquad\qquad\qquad \textit{zero or one } (\{0\ 1\}) \\
& | & * \qquad\qquad\qquad \textit{zero or more } (\{0\ 1\ 2\ \ldots\}) \\
& | & + \qquad\qquad\qquad \textit{one or more } (\{1\ 2\ \ldots\}) \\
& | & \#\{\mathtt{i}_1 \ldots \mathtt{i}_n\} \qquad \textit{set } (\{\mathtt{i}_1\ \ldots\ \mathtt{i}_n\}) \\
& | & \#[\mathtt{i}_1\ \mathtt{i}_2] \qquad \textit{interval } (\{\mathtt{i}_1\ \ldots\ \mathtt{i}_2\})
\end{array}
\tag{7.10}
$$

Set generators *describe sets of tuples whose projections are finite domain types:*

$$
\begin{array}{lll}
\mathtt{g} & ::= & \mathtt{a} \qquad\qquad\qquad \textit{atom} \\
& | & \mathtt{g}_1 \,\&\, \mathtt{g}_2 \qquad \textit{conjunction} \\
& | & \mathtt{g}_1 \,|\, \mathtt{g}_2 \qquad \textit{disjunction} \\
& | & (\mathtt{g}) \qquad\qquad\qquad \textit{brackets}
\end{array}
\tag{7.11}
$$

Feature paths *denote paths to the lexical or non-lexical attributes of a node:*

$$
\begin{array}{lll}
\mathtt{p} & ::= & \_.\mathtt{D}.\mathtt{entry}.\mathtt{a}_1.\ \ldots\ .\mathtt{a}_n \qquad \textit{lexical feature path (daughters)} \\
& | & \hat{}.\mathtt{D}.\mathtt{entry}.\mathtt{a}_1.\ \ldots\ .\mathtt{a}_n \qquad \textit{lexical feature path (mothers)} \\
& | & \_.\mathtt{D}.\mathtt{attrs}.\mathtt{a}_1.\ \ldots\ .\mathtt{a}_n \qquad \textit{non-lexical feature path (daughters)} \\
& | & \hat{}.\mathtt{D}.\mathtt{attrs}.\mathtt{a}_1.\ \ldots\ .\mathtt{a}_n \qquad \textit{non-lexical feature path (mothers)}
\end{array}
\tag{7.12}
$$

In addition to the usual expressions (atoms, integers, sets etc.), the terms of the XDK description language include a number of extensions:

- variables, which will be used for abstraction in the lexicon description
- record specifications, which allow to specify records partially by omitting any number of attributes. Upon interpreting the terms, the omitted attributes are set to the default value of the respective type, defined by the top value of its corresponding lattice (see appendix A). For example, given the following record type:

```
{ in: set({subj obj})
  out: set({subj obj}) }
```
(7.13)

  The record specification {out : {subj}} represents the following record:

```
{ in: top
  out: {subj} }
```
(7.14)

  where top of the type set({subj obj}) stands for the empty set, and thus (7.14) for:

```
{ in: {}
  out: {subj} }
```
(7.15)

- cardinalities and valencies, which are notational conveniences for sets of integers and for records whose values are cardinalities, allowing to abbreviate for instance:

```
{ subj: {1} obj: {1} adv: {0 1 2 ...} }
```
(7.16)

  as:

```
{ subj! obj! adv* }
```
(7.17)

- lattice operations (top, bot, &)
- alternations: $t_1 \mid t_2$ stands for the non-deterministic choice "either t_1 or t_2"
- set generators for economically describing sets of agreement tuples (cf. section 4.6). For example, the set generator $ sg, whose type must be any set of tuples of domains where one of the projections includes sg, e.g.:

```
set(tuple({"1" "2" "3"} {sg pl}))
```
(7.18)

  denotes the set of tuples with sg at their second projection, i.e.:

```
{ ["1" sg] ["2" sg] ["3" sg] }
```
(7.19)

  and the set generator $ ("1" | "3") & sg denotes the set of tuples with either "1" or "3" at their first projection and sg at their second:

```
{ ["1" sg] ["3" sg] }
```
(7.20)

- orders to abbreviate sets of tuples which represent strict partial orders, e.g.:

```
<subj "^" obj vinf adv>
```
(7.21)

  abbreviates the following set:

```
{ [subj "^"] [subj obj] [subj vinf] [subj adv]
  ["^" obj] ["^" vinf] ["^" adv]
  [obj vinf] [obj adv] [vinf adv] }
```
(7.22)

- concatenations of atoms of type `string`
- lexical and non-lexical feature paths to access the lexical and non-lexical attributes of a node. As feature paths must be dynamically resolved during parsing, they can only be used in principle instantiations but not in the lexicon description, which must be completely static. We will give examples for feature paths below.
- type annotations to annotate terms with types

For the constraint parser, we will transform most of these extensions into *core terms* in the interpretation step of the encoder of the lattice functors (appendix A).

Definition 31 (Core Terms)**.** *Given a set* $\mathtt{A}$ *of atoms and a set* $\mathtt{N}$ *of integers, the terms* $\mathtt{CTe}$ *of the XDK description language are defined as follows:*

$$
\begin{array}{lll}
\mathtt{a} \in \mathtt{A} & & \\
\mathtt{i} \in \mathtt{N} & & \\
\mathtt{t} \in \mathtt{Te} & ::= \quad \mathtt{a} & \textit{atom} \\
& \mid \quad \mathtt{i} & \textit{positive integer} \\
& \mid \quad \{\mathtt{t}_1 \ldots \mathtt{t}_n\} & \textit{set} \\
& \mid \quad \{\mathtt{i}_1 \ldots \mathtt{i}_n \ldots\} & \textit{infinite set of integers} \\
& \mid \quad [\mathtt{t}_1 \ldots \mathtt{t}_n] & \textit{list or tuple} \\
& \mid \quad \{\mathtt{a}_1 : \mathtt{t}_1 \ldots \mathtt{a}_n : \mathtt{t}_n\} & \textit{totally specified record} \\
& \mid \quad \mathtt{p} & \textit{feature path}
\end{array}
\tag{7.23}
$$

where $\mathtt{p}$ *is defined as above in (7.12).*

Lexicon Description. The terms of the XDK description language are mainly used for the lexicon description of metagrammars, where the lexicon is described using *lexical classes*. A lexical class is a representation of a set of lexical entries, and can additionally abstract over any number of variables, making them similar to templates in other grammar formalisms such as PATR-II (Shieber 1984) and LFG.

In the lexicon description, we distinguish between *lexical class definitions*, where a lexical class is named and the variables it abstracts over are defined, and lexical classes per se.

Definition 32 (Lexical Class Definitions)**.** *Given a set of atoms* $\mathtt{A}$ *and a set of variables* $\mathtt{V}$*, a lexical class* $\mathtt{l}$ *named* $\mathtt{a} \in \mathtt{A}$ *and abstracting over variables* $\mathtt{v}_1 \ldots \mathtt{v}_n \in \mathtt{V}$ *in* $\mathtt{l}$ *is defined as follows:*

$$\mathtt{defclass}\ \mathtt{a}\ \mathtt{v}_1 \ldots \mathtt{v}_n\ \{\mathtt{l}\} \tag{7.24}$$

Definition 33 (Lexical Classes)**.** *Given a set* $\mathtt{A}$ *of atoms and* $\mathtt{V}$ *of variables, a lexical class is defined as follows:*

$$\begin{array}{lll} \mathtt{a} \in \mathtt{A} & & \\ \mathtt{v} \in \mathtt{V} & & \\ \quad \mathtt{l} \;::=\; & \mathtt{dim}\;\mathtt{a}\;\mathtt{t} & \textit{dimension specification} \\ \quad\quad \mid & \mathtt{a}\,\{\mathtt{v}_1 : \mathtt{t}_1 \ldots \mathtt{v}_n : \mathtt{t}_n\} & \textit{class reference} \\ \quad\quad \mid & \mathtt{l}_1 \,\&\, \mathtt{l}_2 & \textit{greatest lower bound} \\ \quad\quad \mid & \mathtt{l}_1 \mid \mathtt{l}_2 & \textit{alternation} \end{array} \tag{7.25}$$

where the ampersand for greatest lower bound can be omitted for convenience.

A dimension specification `dim a t` stands for the record specification $\{\mathtt{a} : \mathtt{t}\}$, which describes the lexical attributes for dimension `a`. A class reference $\mathtt{a}\,\{\mathtt{v}_1 : \mathtt{t}_1 \ldots \mathtt{v}_n : \mathtt{t}_n\}$ refers to the lexical class definition `defclass a` $\mathtt{v}_1 \ldots \mathtt{v}_n$ $\{\mathtt{l}\}$ with the same name `a`, and represents $\mathtt{l}^{\mathtt{t}_1/\mathtt{v}_1 \ldots \mathtt{t}_n/\mathtt{v}_n}$, i.e., the result of substituting each variable $\mathtt{v}_i$ in `l` by the term $\mathtt{t}_i$ for $0 \leq \mathtt{i} \leq \mathtt{n}$. Greatest lower bound and alternation are lattice operations as for terms.

After defining the lexical classes describing the lexicon, lexical entries must be explicitly generated by writing, given a lexical class `l`:

```
defentry {l}
```
(7.26)

This generates all lexical entries described by `l`.

Here is an example. We first define the lexical classes `"verb"`, `"intrans"`, `"trans"` and `"fin"`, repeated from (2.12), (2.13), (2.14) and (2.11):

```
defclass "verb" {
  dim syn {out: {adv*}}
  dim sem {in: {root! th*}}}
```
(7.27)

```
defclass "intrans" {
  dim sem {out: {ag!}}
  dim synsem {arg: {ag: {subj}}}}
```
(7.28)

```
defclass "trans" {
  "intrans"
  dim syn {out: {obj!}}
  dim sem {out: {pat!}}
  dim synsem {arg: {pat: {obj}}}}
```
(7.29)

```
defclass "fin" Word Agrs {
  dim lex {word: Word}
  dim syn {in: {root?}
           out: {subj!}
           order: <subj "^" obj vinf adv>
           agrs: Agrs
           agree: {subj}}}
```
(7.30)

where the possibility of partially specifying records is heavily used, e.g., only the `out` attribute of the `syn` dimension is specified in (7.27). Then, we explicitly generate the lexical entries for the word *eat* by making use of the classes:

```
defentry {
  "verb"
  ("intrans" | "trans")
  "fin" {Word: "eat"
         Agrs: $ (("1"|"2") |  ("3" & sg))}}
```
(7.31)

This results in the two lexical entries shown below, one intransitive (using the lexical class `"intrans"`) and one transitive (`"trans"`):

```
dim lex {word: "eat"}
dim syn {in: {root?}
         out: {subj! adv*}
         order: <subj "^" obj vinf adv>
         agrs: $ (("1"|"2") |  ("3" & sg))
         agree: {subj}}
dim sem {in: {root! th*}
         out: {ag!}}
dim synsem {arg: {ag: {subj}}}
```

(7.32)

```
dim lex {word: "eat"}
dim syn {in: {root?}
         out: {subj! obj! adv*}
         order: <subj "^" obj vinf adv>
         agrs: $ (("1"|"2") |  ("3" & sg))
         agree: {subj}}
dim sem {in: {root! th*}
         out: {ag! pat!}}
dim synsem {arg: {ag: {subj}
                  pat: {obj}}}
```

(7.33)

where (7.33), for example, represents the following core term, where the valencies and cardinalities (`in` and `out` attributes), orders (`order`) and set generators (`agrs`) are compiled out:

```
{lex: {word: "eat"}
 syn: {in: {root: {0 1}}
       out: {subj: {1} obj: {1} adv: {0 1 2 ...}}
       order: {[subj "^"] [subj obj] [subj vinf] [subj adv]
               ["^" obj] ["^" vinf] ["^" adv]
               [obj vinf] [obj adv] [vinf adv]}
       agrs: {["1" sg] ["2" sg] ["1" pl] ["2" pl] ["3" pl]}
       agree: {subj}}
 sem: {in: {root: {1} th: {0 1 2 ...}}
       out: {ag: {1} pat: {1}}}
 synsem: {arg: {ag: {subj}
                pat: {obj}}}}
```

(7.34)

Principle Instantiations. The second use of the terms of the XDK description language is in principle instantiations. Upon instantiation, a principle binds the dimension variables of its principle definitions to actual dimensions, and the argument variables to terms. For example, here is an instantiation of the Valency principle, which was defined in (7.7):

```
useprinciple "principle.valency" {
  dims {D: syn}
  args {In: {root?}
        Out: {subj! adv*}}
  ...}
```

(7.35)

where the dimension variable `D` is bound to dimension `syn`, and the argument variables `In` and `Out` to valencies. As the `In` and `Out` arguments are interpreted for all nodes, this principle instantiation stipulates that all nodes have the same licensed incoming and outgoing edges.

Clearly, this is not what we generally want. Instead, what we want is a lexicalized instantiation of the Valency principle, where the licensed incoming and outgoing edges are specified

by the lexical entry of each node. This is precisely the purpose of the lexical feature paths in the following principle instantiation:

```
useprinciple "principle.valency" {
  dims {D: syn}
  args {In: _.D.entry.in
        Out: _.D.entry.out}
  ...}
```
(7.36)

where the feature path `_.D.entry.in` represents lexical attribute `in`, and `_.D.entry.out` the lexical attribute `out` on the dimension represented by dimension variable `D`, i.e., `syn`.

For principles which quantify over edges instead of nodes, the feature paths need to distinguish the mother of the edge from the daughter. For example, here is the instantiation of the Agreement principle, where for each edge, the value of `Agr1` is determined by the non-lexical attribute `agr` of the mother (^), and `Agr2` by the value of `agr` of the daughter (_). `Agree` is not lexicalized and set to {`subj`}:

```
useprinciple "principle.agreement" {
  dims {D: syn}
  args {Agr1: ^.D.attrs.agr
        Agr2: _.D.attrs.agr
        Agree: {subj}}
  ...}
```
(7.37)

7.3. Summary

In this chapter, we have presented the overall architecture of the XDK and then turned our attention to the XDK description language. We put its types to use in the type definitions of metagrammars and principle definitions, and then its terms in the lexicon description and the instantiation of principles.

8. Constraint Parser

This chapter describes the constraint parser, which is at the heart of the XDK, as can be seen in Figure 8.1. We show how multigraphs can be modeled in terms of finite sets of integers, and how this idea is implemented in the actual constraint parser and the principles of the XDK principle library. After a short excursion to generation, we close by discussing the runtime of the parser.

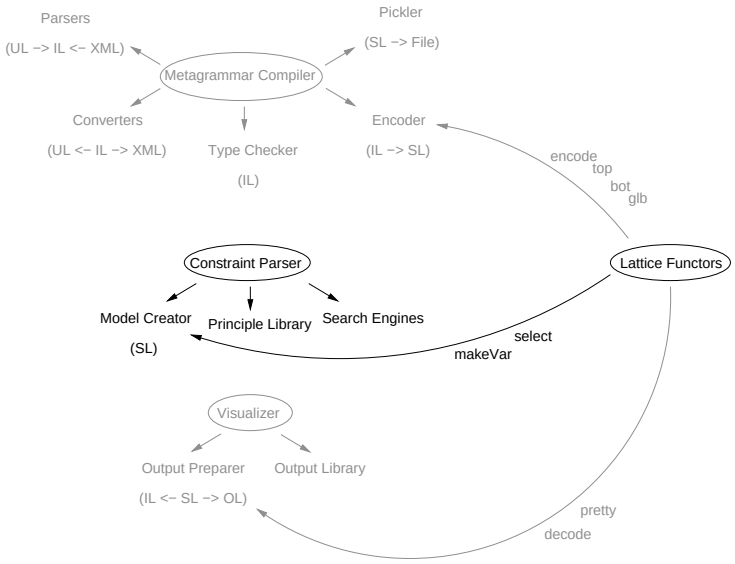


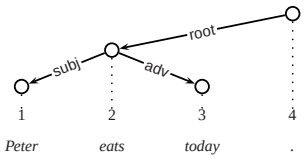
Figure 8.1.: The constraint parser in the XDK architecture

8.1. Modeling Multigraphs

The XDK constraint parser is based on the idea of modeling multigraphs in terms of *finite sets of integers*, and making use of the support for *finite set constraint programming* implemented in Mozart/Oz (Schulte 2002). We begin by showing how to model individual dependency graphs, how to add attributes, and how to extend the modeling to multigraphs.

8.1.1. Modeling Dependency Graphs

A dependency graph is a labeled directed graph whose nodes are identified by indices and words, as in the example dependency graph for the sentence *Peter eats today* below:



(8.1)

The graph consists of four nodes, including the additional fourth node for the full stop, which is connected to the actual root of the analysis (the finite verb *eats*) by an edge labeled root. *eat* has two daughters: the subject *Peter* and the adverb *today*.

We model graphs using sets of records, one for each node. Each node contains a representation of its outgoing edges in *daughter sets*. For example, the second node (*eats*) corresponds to the record below:

$$\left\{\begin{array}{rl} \mathit{index} = 2 & \\ \mathit{word} = \mathit{eats} & \\ \mathit{nodeSet} = \{1,2,3,4\} & \\ \mathit{model} = \left\{ \mathit{daughtersL} = \left\{ \begin{array}{l} \mathsf{adv} = \{3\} \\ \mathsf{root} = \{\} \\ \mathsf{subj} = \{1\} \end{array} \right\} \right\} & \end{array}\right\} \tag{8.2}$$

where the attribute *index* represents the index of the node, and *word* the word. *nodeSet* represents the entire set of nodes of the graph. Given an edge label, the daughter sets (attribute *daughtersL* in the *model* subrecord) denote the sets of indices of the daughters with that edge label. In the example, the set of adv daughters of node 2 contains node 3, the set of root daughters is empty, and the set of subj daughters contains node 1. Using Mozart/Oz syntax in form of the *Solver Language* (*SL*), we can represent (8.2) as the following record called *node record*:

```
o(index: 2
  word: eats
  nodeSet: {1 2 3 4}#4
  model: o(daughtersL: o(adv: {3}#1
                         root: {}#0
                         subj: {1}#1)))
```

(8.3)

where the os are dummy record labels required because each record in Oz must be labeled, and sets are represented together with their cardinality: e.g. $\{3\}\#1$ stands for the set $\{3\}$ with cardinality 1.

In practice, the XDK constraint parser makes use of many more sets, mainly to ease the statement of constraints and to improve constraint propagation. The sets are determined by the freely extensible and also replaceable *Graph principle* from the XDK principle library.[1] The current version of the Graph principle makes use of the following sets, given a node *v*:

[1] The existence of the Graph principle in the principle library is one of the few divergences of the XDK from the formalization of XDG in part I: in XDG, graphs were hardwired into the formalization. In the XDK, they are modularized into a principle, such that its implementation can easily be replaced, e.g. by a more efficient one.

- `mothers`: the set of mothers of *v*
- `daughters`: the set of daughters of *v*
- `up`: the set of nodes above *v*
- `down`: the set of nodes below *v*
- `eq`: the set of nodes including only *v* itself
- `equp`: the set of nodes equal or above *v*
- `eqdown`: the set of nodes equal or below *v*
- `labels`: the set of edge labels of the incoming edges of *v*
- `mothersL`: the set of mothers of *v* sorted according to their edge label
- `daughtersL`: the set of daughters of *v* sorted according to their edge label
- `upL`: the set of nodes above *v* sorted according to their edge label when entering *v*
- `downL` the set of nodes below *v* sorted according to their edge label when emanating *v*

For node 2 in (8.1), these sets are instantiated as follows:

```
o(index: 2
  word: eats
  nodeSet: {1 2 3 4}#4
  model: o(mothers: {4}#1
           daughters: {1 3}#2
           up: {4}#1
           down: {1 3}#2
           index: 2
           eq: {2}#1
           equp: {2 4}#2
           eqdown: {1 2 3}#3
           labels: {5}#1
           mothersL: o(adv: {}#0
                       root: {4}#1
                       subj: {}#0)
           daughtersL: o(adv: {3}#1
                         root: {}#0
                         subj: {1}#1)
           upL: o(adv: {}#0
                  root: {4}#1
                  subj: {}#0)
           downL: o(adv: {3}#1
                    root: {}#0
                    subj: {1}#1)))
```
(8.4)

where the labels in the set `labels` are encoded as described in section A.1. Here, the edge label `root` is represented by the integer `5`.

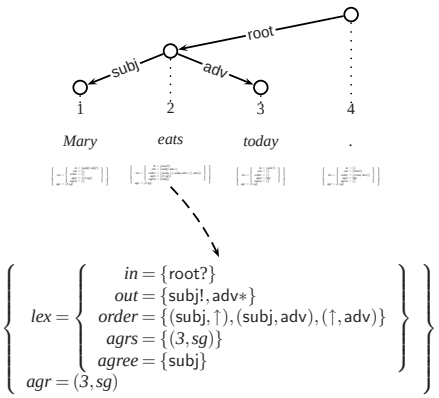


Figure 8.2.: Dependency graph with attributes

8.1.2. Modeling Attributes

In the next step, we extend our modeling of dependency graphs with attributes, as in Figure 8.2, where we display the graph (8.1) with attributes and highlight the attributes of node 2.

We model attributes using the `attrs` subrecord representing the non-lexical attributes, and the `entry` subrecord representing the lexical attributes. For example, the record corresponding to node 2 then becomes:[2]

```
o(index: 2
  word: eats
  nodeSet: {1 2 3 4}#4
  entryIndex: 1
  model: o(daughtersL: o(adv: {3}#1
                         root: {}#0
                         subj: {1}#1))
  attrs: o(agr: 6)
  entry: o('in': o(adv: {0}#1
                   root: {0 1}#2
                   subj: {0}#1)
           out: o(adv: {0 1 2 3}#4
                  root: {0}#1
                  subj: {1}#1)
           order: {2 36 37}#3
           agrs: {6}#1
           agree: {6}#1))
```
(8.5)

where:

- the value of the non-lexical `agr` attribute encodes the tuple $(3, sg)$ as the integer 6, cf. section A.1
- the value of the lexical `order` attribute encodes the set of tuples

$$\{(\mathsf{subj}, \uparrow), (\mathsf{subj}, \mathsf{adv}), (\uparrow, \mathsf{adv})\} \tag{8.6}$$

  where 2 represents the tuple $(\uparrow, \mathsf{adv})$, 36 the tuple $(\mathsf{subj}, \uparrow)$ and 37 the tuple $(\mathsf{subj}, \mathsf{adv})$

[2] The attribute `'in'` is a Mozart/Oz keyword and is thus has to be enclosed in single quotes.

- the value of the lexical `agrs` attribute encodes the set of agreement tuples $\{(3, sg)\}$
- the value of the lexical `agree` attribute encodes the set of edge labels $\{\text{subj}\}$, where subj is encoded as the integer 6

In addition, the attribute `entryIndex` represents the selected lexical entry for the node. In the example, the first lexical entry is selected.

8.1.3. Multigraphs

We now lift our encoding of dependency graphs to multigraphs. To this end, we package the components of the multigraph into subrecords. For example, here is how we model node 2 of the multigraph displayed in Figure 8.3:

```
o(index: 2
  word: eats
  entryIndex: 3
  syn: o(model: o(daughtersL: o(adv: {3}#1
                                root: {}#0
                                subj: {1}#1)
         attrs: o(agr: 6)
         entry: o('in': o(adv: {0}#1
                          root: {0 1}#2
                          subj: {0}#1)
                   out: o(adv: {0 1 2 3}#4
                          root: {0}#1
                          subj: {1}#1)
                   order: {2 36 37}#3
                   agrs: {6}#1
                   agree: {6}#1))
  sem: o(model: o(daughtersL: o(ag: {1}#1
                                root: {}#0
                                th: {}#0))
         attrs: o
         entry: o('in': o(ag: {0}#1
                          root: {1}#1
                          th: {0 1 2 3}#4)
                   out: o(ag: {1}#1
                          root: {0}#1
                          th: {0}#1)))
  synsem: o(attrs: o
            entry: o(arg: o(ag: {3}#1
                            root: {}#0
                            th: {}#0)
                     'mod': {}#0)))
```
(8.7)

As we assume that the models on the `synsem` dimension are graphs without edges, as in the example grammar in section 2.2.4, we can omit the representation of edges using daughter sets for simplicity and efficiency. In fact, we can also omit an implementation of the *Edgeless principle*: it suffices for all dimensions without edges to *not* use the Graph principle.

8.2. Constraint Parsing

The constraint parser itself is realized as an *Oz script*. Oz scripts are programs that can compute one or all solutions of a Constraint Satisfaction Problem (CSP), and are run on *search*

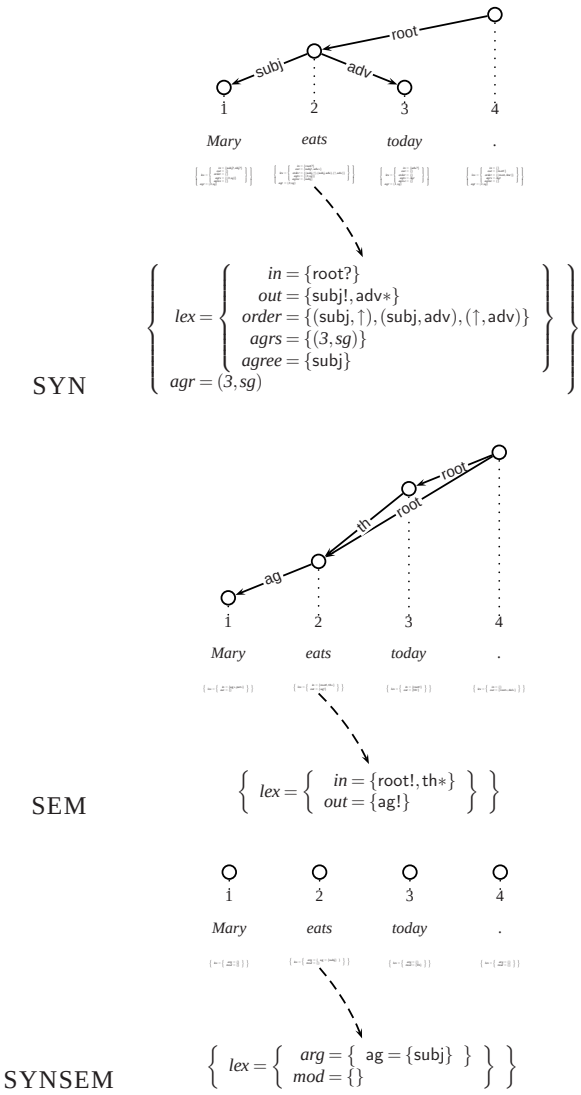


Figure 8.3.: Multigraph

engines implementing the propagate and distribute method. The XDK supports the search engines *Search*, the *Oz Explorer* (Schulte 1997) and *IOzSeF* (Tack 2002).

The constraint parser Oz script is generated by the function `Make` displayed in Figure 8.4. Given a list of words `WordAs`[3] and a compiled grammar `G`, it proceeds in three steps which we elucidate in the following subsections:

1. create node records (lines 7–14)
2. do lexicalization[4] (lines 16–23)

[3]We make use of a convention to suffix Oz variables with type information, similar to e.g. the Hungarian notation for C++, which is explained in the XDK manual (Debusmann & Duchier 2006). For example, `A` stands for an atom, `As` for a list of atoms, `I` for an integer and `M` for a set.

[4]Note that contrary to the formalization XDG in part I, where lexicalization was realized as a principle (cf.

3. post principles (line 28)

```
(  1) fun {Make WordAs G}
(  2)    proc {$ Nodes}
(  3)       NodeSetM = {FS.value.make 1#{Length WordAs}}
(  4)       !Nodes =
(  5)       {List.mapInd WordAs
(  6)        fun {$ IndexI WordA}
(  7)           Node = {G.nodeLat.makeVar}
(  8)
(  9)           Node.index = IndexI
( 10)           Node.word = WordA
( 11)           Node.nodeSet = NodeSetM
( 12)
( 13)           Entries = G.lexicon.WordA
( 14)           Node.entryIndex = {FD.int 1#{Length Entries}}
( 15)
( 16)           for DIDA in G.dIDAs do
( 17)              EntryLat = {G.dIDA2EntryLat DIDA}
( 18)              DIDAEntries = {Map Entries
( 19)                             fun {$ Entry} Entry.DIDA end}
( 20)           in
( 21)              Node.DIDA.entry =
( 22)              {EntryLat.select DIDAEntries Node.entryIndex}
( 23)           end
( 24)        in
( 25)           Node
( 26)        end}
( 27)    in
( 28)       {G.principles.post Nodes G}
( 29)    end
( 30) end
```

Figure 8.4.: The script generator realizing the constraint parser

8.2.1. Creating Node Records

For each word in the list of words `WordAs`, the script creates the node record `Node` in line 7 of the script, using the `makeVar` method of the lattice functors (explained in detail in section A.3). Essentially, for the attributes in the node record, `makeVar` creates corresponding *constraint variables*.

In lines 9–11, the constraint variables of the `index`, `word` and `nodeSet` attributes are instantiated: `index` is set to the index `IndexI` of the word in the list of words, `word` to the word `WordA`, and `nodeSet` to the entire set of indices required for the list of words.

8.2.2. Lexicalization

The lexicon of a compiled grammar is a record which maps each word to a list of lexical entries for it. In line 13 of the script, we obtain the list of entries `Entries` for word `WordA`

section 4.3), it is hardwired in the constraint parser of the XDK. This is the result of a design decision, taken because the existence of a lexicon is a central assumption of the XDK.

from the lexicon. Then, we instantiate the attribute `entryIndex` with a finite domain variable ranging from 1 to the number of lexical entries for `WordA` (line 14). `entryIndex` represents the selected lexical entry for the node, which is shared by all dimensions and thus synchronizes their lexical selection.

Lexical entry selection itself is then implemented in lines 16–23. For each dimension identifier `DIDA` in the list of all dimensions of the grammar `G.dIDAs`, first the appropriate lattice for the record of lexical attributes `EntryLat` is obtained (line 17), and second the list of entries `DIDAEntries` (lines 18–19). Then, in lines 21–22, the lexical attributes on dimension `DIDA` in the `entry` record are instantiated with the lexical attributes of the entry selected from the lexicon using the entry index. It is here that we make use of the `select` method of the lattice functors (explained in detail in section A.3). The method utilizes the *selection constraint* (Duchier 1999, Duchier 2003), which significantly improves constraint propagation and therefore also the treatment of lexical ambiguity. The genius behind the constraint is that it makes the commonalities of the lexical entries of a word available for propagation as soon as possible, long before the lexical entry is eventually selected.

8.2.3. Posting Principles

The final step consists of posting the principles of the grammar for nodes `Nodes` and grammar `G` in line 28. The modeling of these principles is the topic of the next section.

8.3. Modeling Principles

Most of the actual functionality of the constraint parser is factored out into the principles. A principle consists of:

- a *principle definition*
- a set of *node constraint functors*
- a set of *edge constraint functors*

The principles are arranged in the extensible *principle library* of the XDK.

8.3.1. Principle Definitions

A principle definition is an XDK term defining the following:

- the identifier of the principle
- a set of *dimension variables*, one for each dimension referred to by the principle
- the types of the arguments of the principle
- default values for the arguments

- the type of the *model record* introduced by the principle
- the set of *node constraint functors* implementing the principle, coupled with a priority $\neq 100$, which determines when the constraint functor is posted (the higher the earlier)
- the set of *edge constraint functors* implementing the principle, coupled with the dimension, which determines which dimensions' edges shall be constrained. Edge constraint functors always have priority 100, i.e., they are posted after the node constraints with priority > 100 and before those < 100.

where the purpose of the constraint priorities is to enable optimization of the constraint solver by determining the order in which they are posted. As an example, we show the principle definition of the *Graph principle* below:

```
defprinciple "principle.graph" {
  dims {D}
  args {:}
  defaults {:}
  model {mothers: set(int)
         daughters: set(int)
         up: set(int)
         down: set(int)
         index: int
         eq: set(int)
         equp: set(int)
         eqdown: set(int)
         labels: set(label(D))
         mothersL: vec(label(D) set(int))
         daughtersL: vec(label(D) set(int))
         upL: vec(label(D) set(int))
         downL: vec(label(D) set(int))}
  constraints {"GraphMakeNodes": 130
               "GraphConditions": 120
               "GraphDist": 90}
  edgeconstraints {"GraphMakeEdges": D}}
```
(8.8)

The identifier of the principle is `"principle.graph"`. It constrains only one dimension represented by the dimension variable `D` (`dims`). The principle neither has arguments (`args`) nor defaults (`defaults`). The model record (`model`) defines the types of the attributes introduced in section 8.1.1. The principle is implemented by the node constraint functors `"GraphMakeNodes"` (priority 130), `"GraphConditions"` (120), and `"GraphDist"` (90)[5] (`constraints`), and the edge constraint functor `"GraphMakeNodes"` (for edges on dimension `D`) (`edgeconstraints`).

[5] As we will see soon, this constraint functor does not implement constraints but controls distribution in the Mozart/Oz search engine running the constraint parser script.

As a second example, we present the principle definition of the *Valency principle* (cf. principle 6 in chapter 4):

```
defprinciple "principle.valency" {
  dims {D}
  args {In: valency(label(D))
        Out: valency(label(D))}
  defaults {In: _.D.entry.in
            Out: _.D.entry.out}
  model {:}
  constraints {"In": 130
               "Out" 130}
  edgeconstraints {:}}
```
(8.9)

The Valency principle has two arguments, where `In` stands for the valency specification for the incoming edges, and `Out` for the outgoing edges. The default for `In` is the feature path `_.D.entry.in` representing the lexical attribute `in` on dimension `D`, and the feature path for `Out` represents the lexical attribute `out` on dimension `D`.

As a third example, here is the principle definition of the *Agreement principle* (cf. principle 9 in chapter 4), which was already partially given in (7.8):

```
defprinciple "principle.agreement" {
  dims {D}
  args {Agr1: tv(X)
        Agr2: tv(X)
        Agree: set(label(D))}
  defaults {Agr1: ^.D.attrs.agr
            Agr2: _.D.attrs.agr
            Agree: ^.D.entry.agree}
  model {:}
  constraints {:}
  edgeconstraints {"Agreement": D}}
```
(8.10)

The principle abstracts over dimension variable `D` and has three arguments: `Agr1`, `Agr2` and `Agree`. Given an edge, the default for `Agr1` is the feature path denoting the non-lexical attribute `agr` of the mother, for `Agr2` the non-lexical attribute `agr` of the daughter, and for `Agree` the lexical attribute `agree` of the mother. It is implemented by the edge constraint functor `"Agreement"` on dimension `D`.

8.3.2. Node Constraint Functors

Node constraint functors have the purpose of constraining the nodes of the analysis. They directly implement Oz procedures (functions with no return value) called `Constraint`, and have four arguments:

1. `Nodes`: the list of node records of the analysis
2. `G`: the grammar
3. `GetDim`: a function mapping dimension variables to dimensions
4. `GetArg2`: a function mapping two arguments (hence the 2), namely, an argument variable and a node record, to an argument

where the purpose of GetDim is to obtain the dimensions, and of GetArg2 to obtain the arguments of the principle, given a node record. Because the arguments can also be feature paths, they have to be resolved, dynamically at runtime.

As a first example, Figure 8.5 shows the node constraint functor "GraphMakeNodes", which is a part of the Graph principle. In line 2, it obtains the dimension represented by dimension variable 'D', and in line 3 the set of all nodes of the analysis NodeSetM. Then, in lines 5–24, it posts the following constraints on all nodes Node: the sets mothers, daughters, up and down of the model record of the node are all subsets of the set of all nodes (lines 8–11), the index equals the index of the node (line 13), and eq is the singleton set containing only the index (line 14). The set equp is the set of nodes equal or above the node (line 16), and eqdown equal or below (line 17). The set mothers is the disjoint union of the sets in the mothersL record (line 19), i.e., is a partition of this set, and analogously for daughters (line 20). Finally, up is the union of the sets in the upL record (line 22), and analogously for downL (line 23).

```
(  1) proc {Constraint Nodes G DVA2DIDA}
(  2)    DIDA = {DVA2DIDA 'D'}
(  3)    NodeSetM = Nodes.1.nodeSet
(  4) in
(  5)    for Node in Nodes do
(  6)       Model = Node.DIDA.model
(  7)    in
(  8)       {FS.subset Model.mothers NodeSetM}
(  9)       {FS.subset Model.daughters NodeSetM}
( 10)       {FS.subset Model.up NodeSetM}
( 11)       {FS.subset Model.down NodeSetM}
( 12)
( 13)       Model.index = Node.index
( 14)       Model.eq = {FS.value.make Model.index}
( 15)
( 16)       Model.equp = {FS.union Model.eq Model.up}
( 17)       Model.eqdown = {FS.union Model.eq Model.down}
( 18)
( 19)       Model.mothers = {FS.partition Model.mothersL}
( 20)       Model.daughters = {FS.partition Model.daughtersL}
( 21)
( 22)       Model.up = {FS.unionN Model.upL}
( 23)       Model.down = {FS.unionN Model.downL}
( 24)    end
( 25) end
```

Figure 8.5.: "GraphMakeNodes" node constraint functor

Figure 8.5 shows the node constraint functor "GraphConditions", also a part of the Graph principle. It obtains the dimension represented by dimension variable 'D' in line 2, and the list LAs of edge labels on that dimension in lines 3–5. In lines 7–9, it creates lists of the model records, the eqdown sets, and the equp sets of the nodes, before it quantifies over the model records in lines 11–22, where it makes repeated use of the *selection union constraint* Select.union introduced in (Duchier 2003), whose declarative semantics is the following for $1 \leq i \leq n$:

$$\{\texttt{Select.union}\ [M_1 \ldots M_n]\ M\} \quad = \quad \bigcup_{i \in M} M_i \tag{8.11}$$

For all nodes, the set of nodes below the node equals the union of the `eqdown` sets of the daughters (line 12), and the set of nodes above the node equals the union of the `equp` sets of the mothers (line 13). Similarly, for all edge labels `LA` in `LAs`, the `LA downL` set is the union of the `eqdown` sets of the `LA` daughters (lines 15–17), and the `LA upL` set is the union of the `equp` sets of the `LA` mothers (lines 18–20).

```
(  1) proc {Constraint Nodes G DVA2DIDA}
(  2)    DIDA = {DVA2DIDA 'D'}
(  3)    DIDA2LabelLat = G.dIDA2LabelLat
(  4)    LabelLat = {DIDA2LabelLat DIDA}
(  5)    LAs = LabelLat.constants
(  6)
(  7)    Models = {Map Nodes fun {$ Node} Node.DIDA.model end}
(  8)    EqdownMs = {Map Models fun {$ Model} Model.eqdown end}
(  9)    EqupMs = {Map Models fun {$ Model} Model.equp end}
( 10) in
( 11)    for Model in Models do
( 12)       Model.down = {Select.union EqdownMs Model.daughters}
( 13)       Model.up = {Select.union EqupMs Model.mothers}
( 14)
( 15)       for LA in LAs do
( 16)          Model.downL.LA = {Select.union EqdownMs Model.daughtersL.LA}
( 17)       end
( 18)       for LA in LAs do
( 19)          Model.upL.LA = {Select.union EqupMs Model.mothersL.LA}
( 20)       end
( 21)    end
( 22) end
```

Figure 8.6.: `"GraphConditions"` node constraint functor

As another example, Figure 8.7 shows the node constraint functor `"In"`, which implements the first half of the Valency principle, dealing with the incoming edges of each node. In lines 7–13, the constraint functor quantifies over all nodes `Node` and all edge labels `LA` to constrain the set of mothers of `Node` according to the valency specification denoted by the argument variable `'In'`, which is obtained using `GetArg2` (line 11). If `'In'` denoted the feature path `_.D.entry.in`, as in the defaults of the Valency principle in (8.9), the function call `{GetArg2 'In' Node}` in line 11 would dynamically resolve it to the value of `Node.DIDA.entry.in`, i.e., the lexical attribute `in` on dimension `DIDA` of node `Node`.

8.3.3. Edge Constraint Functors

Edge constraint functors have the purpose to constrain edges of the analysis. They have four arguments, similar to node constraint functors:

1. `Nodes`: the list of node records of the analysis
2. `G`: the grammar
3. `GetDim`: a function mapping dimension variables to dimensions
4. `GetArg3`: a function mapping three arguments (hence the 3), namely, an argument variable and two node records to arguments

```
(  1) proc {Constraint Nodes G GetDim GetArg2}
(  2)    DIDA = {GetDim 'D'}
(  3)    DIDA2LabelLat = G.dIDA2LabelLat
(  4)    LabelLat = {DIDA2LabelLat DIDA}
(  5)    LAs = LabelLat.constants
(  6) in
(  7)    for Node in Nodes do
(  8)       for LA in LAs do
(  9)          {FS.include
( 10)           {FS.card Node.DIDA.model.mothersL.LA}
( 11)           {GetArg2 'In' Node}}
( 12)       end
( 13)    end
( 14) end
```

Figure 8.7.: "In" node constraint functor

where the purpose of GetArg3 is to obtain the arguments of the principle, given two node records (one for the mother and one for the daughter of the edge).

Contrary to node constraint functors, which directly implement constraints on the multigraph, edge constraint functors return procedures implementing constraints on labeled edges, which still need to be executed to actually post the constraints. As an example, we show the edge constraint functor "GraphMakeEdges" in Figure 8.8. The functor returns a procedure with the arguments Node1, Node2 and LA, which does nothing (skip).

```
(  1) fun {Constraint Nodes G GetDim GetArg3}
(  2)    Proc = proc {$ Node1 Node2 LA} skip end
(  3) in
(  4)    Proc
(  5) end
```

Figure 8.8.: "GraphMakeEdges" edge constraint functor

As another example, we show the edge constraint functor implementing the Agreement principle in Figure 8.9. It implements the constraint (lines 10–12) that if the integer LI encoding the edge label LA is in the set denoted by the 'Agree' argument variable of the principle, then the value denoted by the 'Agr1' argument variable must equal that of 'Agr2'. Assuming the defaults of the Agreement principle defined in (8.10) above,

```
{GetArg3 'Agree' Node1 Node2}
```
(8.12)

in line 10 corresponds to Node1.DIDA.entry.agree,

```
{GetArg3 'Agr1' Node1 Node2}
```
(8.13)

in line 12 to Node1.DIDA.attrs.agr, and

```
{GetArg3 'Agr2' Node1 Node2}
```
(8.14)

also in line 12 to Node2.DIDA.attrs.agr.

Where in the XDK constraint parser are the procedures returned by the edge constraint functors executed? Edge constraints are executed by a special functor called *edge functor*, displayed in Figure 8.10. The edge functor has the following arguments:

```
(  1) fun {Constraint Nodes G GetDim GetArg3}
(  2)    DIDA = {GetDim 'D'}
(  3)    DIDA2LabelLat = G.dIDA2LabelLat
(  4)    LabelLat = {DIDA2LabelLat DIDA}
(  5)
(  6)    Proc =
(  7)    proc {$ Node1 Node2 LA}
(  8)       LI = {LabelLat.a2I LA}
(  9)    in
( 10)       {FS.reified.include LI {GetArg3 'Agree' Node1 Node2}}
( 11)       =<:
( 12)       ({GetArg3 'Agr1' Node1 Node2}=:{GetArg3 'Agr2' Node1 Node2})
( 13)    end
( 14) in
( 15)    Proc
( 16) end
```

Figure 8.9.: `"Agreement"` edge constraint functor

1. `Nodes`: the list of node records
2. `G`: the grammar
3. `DIDA`: the dimension whose edges shall be constrained
4. `Procs`: the procedures of all edge constraint functors for dimension `DIDA`

For each edge from mother `Node1` to daughter `Node2` labeled `LA`, lines 23–28 launch a thread containing a *deep guard*, which implements the following: either the edge is contained in the graph, or it is not. If it is, then:

- the index of the daughter must be an element of the set of daughters of the mother labeled `LA` (line 24)
- the label `LA` encoded as an integer (`LI`) must be an element of the set of incoming edge labels of the daughter (line 25)
- the procedures of all edge constraint functors for dimension `DIDA` are posted (line 26)

If the edge is not contained in the graph, then the index of the daughter must not be an element of the set of daughters of the mother labeled `LA` (line 27). The idea of using deep guards for edge constraints was introduced in (Duchier 1999). The key advantage is that if any of the constraints of the edge constraint functors in `Procs` is inconsistent for an edge, constraint propagation can immediately infer that the edge is not contained in the graph.

The purpose of returning a function instead of directly implementing the edge constraint is to enable us to collect all edge constraints, and then embed them in one piece in the deep guards launched by the edge functor. That means that we can post all edge constraint functors using only one thread per possible edge, instead of having to launch one thread for each edge constraint functor.

```
(  1) proc {Edge Nodes G DIDA Procs}
(  2)    DIDA2LabelLat = G.dIDA2LabelLat
(  3)    LabelLat = {DIDA2LabelLat DIDA}
(  4) in
(  5)    for Node1 in Nodes do
(  6)       Model1 = Node1.DIDA.model
(  7)    in
(  8)       for Node2 in Nodes do
(  9)          Model2 = Node2.DIDA.model
( 10)       in
( 11)          {FS.reified.include Model2.index Model1.down}=:
( 12)          {FS.reified.include Model1.index Model2.up}
( 13)
( 14)          {FS.reified.include Model2.index Model1.daughters}=:
( 15)          {FS.reified.include Model1.index Model2.mothers}
( 16)
( 17)          for LA in LabelLat.constants do
( 18)             LI = {LabelLat.a2I LA}
( 19)          in
( 20)             {FS.reified.include Model2.index Model1.daughtersL.LA}=:
( 21)             {FS.reified.include Model1.index Model2.mothersL.LA}
( 22)
( 23)             thread
( 24)                or {FS.include Model2.index Model1.daughtersL.LA}
( 25)                   {FS.include LI Model2.labels}
( 26)                   for Proc in Procs do {Proc Node1 Node2 LA} end
( 27)                [] {FS.exclude Model2.index Model1.daughtersL.LA}
( 28)                end
( 29)             end
( 30)          end
( 31)       end
( 32)    end
( 33) end
```

Figure 8.10.: Edge functor

8.3.4. Distribution

Distribution, i.e., non-deterministic choice, is necessary to ensure completeness of constraint parsing, as constraint propagation alone is not complete. In the XDK, distribution is not realized by the constraint solver script but by the node constraint functors of the principle which requires distribution. The reason for this is that only the principles themselves (but not the script) know what attributes they are using and which of these attributes must be distributed.

In practice, distribution is almost solely necessary to ensure completeness of the Graph principle, whose principle definition was displayed in (8.8) above. Here, distribution is realized by the node constraint functor `'GraphDist'` displayed in Figure 8.11 with priority `90`.[6] `'GraphDist'` distributes over the sets of mothers of each node (lines 4–6) and the sets of daughters sorted by their edge label (lines 8–11).

Factoring out distribution from the constraint solver script into node constraint functors enables us to easily obtain a second Graph principle `"principle.graphConstraints"` without

[6] The priority of distribution node constraint functors should be less than the lowest priority of the other node constraint functors: this way, constraint propagation is granted some time before distribution ensues.

```
(  1) proc {Constraint Nodes G GetDim GetArg}
(  2)    DIDA = {GetDim 'D'}
(  3)
(  4)    MothersMs = {Map Nodes
(  5)                 fun {$ Node} Node.DIDA.model.mothers end}
(  6)    {Distributor.distributeMs MothersMs}
(  7)
(  8)    DaughtersLMRecs = {Map Nodes
(  9)                       fun {$ Node} Node.DIDA.model.daughtersL end}
( 10) in
( 11)    {Distributor.distributeMRecs DaughtersLMRecs}
( 12) end
```

Figure 8.11.: "GraphDist" node constraint functor

distribution by simply adapting the principle definition:

```
defprinciple "principle.graphConstraints" {
  ...
  constraints {"GraphMakeNodes": 130
               "GraphConditions": 120}
  edgeconstraints {"GraphMakeEdges": D}}
```
(8.15)

The effect is that the graph models the dimension on which the principle is used are not enumerated. The genius of this is that it gives us *underspecification* (e.g. of PP-attachment, scope etc.) for free without further stipulation: even only partial analyses already contain information about dominance, encoded directly in the attributes down and downL, for example. We will make use of this in chapter 10 below for modeling scope underspecification, and for the interface to CLLS in appendix E.

8.4. Example Principles

In this section, we present three additional example principles for further illustration: the LinkingEnd principle demonstrates a constraint on multiple dimensions, and the Order principle and the Projectivity principle show how constraints on the order of nodes are expressed.

8.4.1. LinkingEnd

The *LinkingEnd principle* demonstrates how multiple dimensions can be constrained. It abstracts over three dimensions (D1, D2 and D3) and the argument LinkEnd, whose type is a vector used to map edge labels on D1 to sets of edge labels on D2. The principle, whose declarative semantics are given in principle 10 in chapter 4, is implemented by the edge constraint functor "LinkingEnd" over edges on dimension D1:

```
defprinciple "principle.linkingEnd" {
  dims {D1 D2 D3}
  args {LinkEnd: vec(label(D1) set(label(D2)))}
  defaults {LinkEnd: ^.D3.entry.linkEnd}
  model {:}
  constraints {:}
  edgeconstraints {"LinkingEnd": D1}}
```
(8.16)

The edge constraint functor `"LinkingEnd"` is displayed in Figure 8.12. By the principle definition in (8.16), it constrains the edges on dimension `D1`. It first obtains the value of the argument `'LinkEnd'` in line 6 as `LinkEndM`, and then stipulates that if `LinkEndM` is non-empty (line 8), then there exists an edge label in the set of incoming edge labels of the daughter `Node2` on dimension `D2` which is an element of `LinkEndM` (lines 9–10).

```
(  1) fun {Constraint Nodes G GetDim GetArg3}
(  2)    D2DIDA = {GetDim 'D2'}
(  3)
(  4)    Proc =
(  5)    proc {$ Node1 Node2 LA}
(  6)       LinkEndM = {GetArg3 'LinkEnd' Node1 Node2}
(  7)
(  8)       ({FS.reified.equal LinkEndM FS.value.empty}=:0)=<:
(  9)       {FS.reified.include
( 10)        {FS.include $ Node2.D2DIDA.model.labels} LinkEndM}
( 11)    end
( 12) in
( 13)    Proc
( 14) end
```

Figure 8.12.: `"LinkingEnd"` edge constraint functor

8.4.2. Order

The XDK provides two implementations of the *Order principle*, one reflecting precisely the declarative semantics of the Order principle given in principle 7 in chapter 4, and a non-lexicalized and optimized implementation based on (Duchier 2003). Since it is more straightforward to explain and more consistent with the declarative semantics, we explain the former.

The Order principle abstracts over a dimension (`D`) and has one argument (`Order`): a set of pairs of edge labels on `D` plus the special anchor label `"^"`. The set represents a strict partial order on the edge labels of `D` and the anchor label `"^"` standing for the node itself. The principle is implemented by the node constraint functor `"Order"` with priority `120`.

```
defprinciple "principle.order" {
  dims {D}
  args {Order: set(tuple((label(D)|{"^"}) (label(D)|{"^"})))
  defaults {Order: _.D.entry.order}
  model {:}
  constraints {"Order": 120}
  edgeconstraints {:}}
```
(8.17)

We show the node constraint functor `"Order"` in Figure 8.13. What does it do? After obtaining the list of edge labels `LAs` on dimension `'D'` (lines 2–5), lines 7–8 create a lattice for the domain of edge labels plus the anchor label `'^'`, and line 9 creates a lattice for pairs of this domain. Line 11 obtains the set of all nodes `NodeSetM`. Then, the node constraint functor loops over all node records `Node` (line 13), obtains the value of the argument variable `'Order'` for `Node` (line 14), and the model record `Model` (line 15). Then, for all labels `LA1` and `LA2`, encoded as an integer in line 19, the functor creates the list `Ms` as follows:

- if both LA1 and LA2 equal the anchor label '^', then Ms is empty—in this case, nothing needs to be ordered (lines 22–23)
- if LA1 equals the anchor label, then the list orders the node Node itself (i.e., its eq set) before the daughters of the node with edge label LA2 (lines 24–25)
- if LA2 equals the anchor label, then the list orders the daughters with edge label LA1 before the node itself (lines 26–27)
- else the daughters with edge label LA1 are ordered before the daughters with edge label LA2 (lines 28–29)

Ms is then transformed into the list Ms1 in lines 32–42.[7] For each set in Ms, if the integer I encoding the tuple [LA1 LA2] is in the set OrderM, then M is contained in Ms1, otherwise, it is replaced by the empty set (lines 36–40). Then, the crucial final constraint is in line 44, stipulating that for all elements M1 and M2 in the list Ms1, if M1 precedes M2 in Ms1, then all elements of M1 must precede all elements of M2.

8.4.3. Projectivity

The *Projectivity principle* (cf. principle 4 in chapter 4) abstracts over a dimension (D) and is implemented by the node constraint functor "Projectivity" with priority 130:

```
defprinciple "principle.projectivity" {
  dims {D}
  args {:}
  defaults {:}
  model {:}
  constraints {"Projectivity": 130}
  edgeconstraints {:}}
```
(8.18)

The node constraint functor "Projectivity" is displayed in Figure 8.14. For all nodes Node, it stipulates that the set of nodes below or equal the node must be convex, i.e., a set without holes (line 5).

8.5. Generation

The constraint solver was so far only geared towards parsing. It is however easy to make it reversible and use it also for generation. To this end, we only need to:

1. introduce the new model record attribute pos representing the eventual position of the node
2. state all constraints on the order of nodes on the positions instead of the indices

We realize this idea by creating reversible versions of the Order principle and the Projectivity principle.

[7] The code in lines 32–44 could be less awkward if Mozart/Oz supported a reified version of the constraint FS.int.seq.

```
(  1) proc {Constraint Nodes G GetDim GetArg2}
(  2)    DIDA = {GetDim 'D'}
(  3)    DIDA2LabelLat = G.dIDA2LabelLat
(  4)    LabelLat = {DIDA2LabelLat DIDA}
(  5)    LAs = LabelLat.constants
(  6)
(  7)    LAs1 = '^'|LAs
(  8)    Label1Lat = {Domain.make LAs1}
(  9)    Label1PairLat = {Tuple1.make [Label1Lat Label1Lat]}
( 10)
( 11)    NodeSetM = Nodes.1.nodeSet
( 12) in
( 13)    for Node in Nodes do
( 14)       OrderM = {GetArg2 'Order' Node}
( 15)       Model = Node.DIDA.model
( 16)    in
( 17)       for LA1 in LAs1 do
( 18)          for LA2 in LAs1 do
( 19)             I = {Label1PairLat.as2I [LA1 LA2]}
( 20)
( 21)             Ms =
( 22)             if LA1=='^' andthen LA2=='^' then
( 23)                nil
( 24)             elseif LA1=='^' then
( 25)                [Model.eq Model.daughtersL.LA2]
( 26)             elseif LA2=='^' then
( 27)                [Model.daughtersL.LA1 Model.eq]
( 28)             else
( 29)                [Model.daughtersL.LA1 Model.daughtersL.LA2]
( 30)             end
( 31)
( 32)             Ms1 = {Map Ms
( 33)                    fun {$ M}
( 34)                       M1 = {FS.subset $ NodeSetM}
( 35)                    in
( 36)                       {FS.reified.include I OrderM}=<:
( 37)                       {FS.reified.equal M M1}
( 38)
( 39)                       ({FS.reified.include I OrderM}=:0)=<:
( 40)                       {FS.reified.equal M1 FS.value.empty}
( 41)                       M1
( 42)                    end}
( 43)          in
( 44)             {FS.int.seq Ms1}
( 45)          end
( 46)       end
( 47)    end
( 48) end
```

Figure 8.13.: `"Order"` node constraint functor

8.5.1. Reversible Order Principle

To the principle definition (8.17) of the Order principle, we add the model record attribute `pos` whose type is `int`, and the additional node constraint functor `ROrderDist` for distributing on

```
(  1) proc {Constraint Nodes G GetDim GetArg2}
(  2)    DIDA = {GetDim 'D'}
(  3) in
(  4)    for Node in Nodes do
(  5)       {FS.int.convex Node.DIDA.model.eqdown}
(  6)    end
(  7) end
```

Figure 8.14.: "Projectivity" node constraint functor

it. ROrder is the reversible version of the node constraint functor Order:

```
defprinciple "principle.rOrder" {
  dims {D}
  args {Order: set(tuple((label(D)|{"^"}) (label(D)|{"^"})))
  defaults {Order: _.D.entry.order}
  model {pos: int}
  constraints {"ROrder": 120
               "ROrderDist": 90}
  edgeconstraints {:}}
```

(8.19)

Figure 8.15 shows the distribution functor "ROrderDist", and Figure 8.16 the modifications of the node constraint functor "Order" of Figure 8.13, which yield the reversible "ROrder" node constraint functor.

```
(  1) proc {Constraint Nodes G GetDim GetArg}
(  2)    DIDA = {GetDim 'D'}
(  3)
(  4)    PosDs = {Map Nodes
(  5)             fun {$ Node} Node.DIDA.model.pos end}
(  6) in
(  7)    {Distributor.distributeDs PosDs}
(  8) end
```

Figure 8.15.: "ROrderDist" node constraint functor

The reversible node constraint functor defines the function IndexM2PosM mapping sets of indices to sets of positions (lines 6). The function is defined using the list PosMs created in lines 2–5, which encodes a mapping from indices to sets of positions: the *i*th list element denotes the set containing only the position of the node with index *i*. Given a set of indices IndexM, IndexM2PosM uses the selection union constraint to efficiently obtain the union of all positions corresponding to M.

8.5.2. Reversible Projectivity Principle

Making the Projectivity principle now works analogously, i.e., it also makes use of the function IndexM2PosM.

8.5.3. Reversible Constraint Parser

When we leave the positions the nodes underspecified before solving, the constraint solver does all the work for us, and finds the right positions of the words automatically. By equating the position of each node with its index, we can easily get the old parsing behavior back.

```
(  1) proc {Constraint Nodes G GetDim GetArg2}
(  2)    PosMs = {Map Nodes
(  3)             fun {$ Node}
(  4)                {FS.value.make Node.DIDA.model.pos}
(  5)             end}
(  6)    fun {IndexM2PosM IndexM} {Select.union PosMs IndexM} end
(  7)
(  8)    DIDA = {GetDim 'D'}
(  9)    ...
( 10)       for LA1 in LAs1 do
( 11)          for LA2 in LAs1 do
( 12)             I = {Label1PairLat.as2I [LA1 LA2]}
( 13)
( 14)             Ms =
( 15)             if LA1=='^' andthen LA2=='^' then
( 16)                nil
( 17)             elseif LA1=='^' then
( 18)                [{IndexM2PosM Model.eq}
( 19)                 {IndexM2PosM Model.daughtersL.LA2}]
( 20)             elseif LA2=='^' then
( 21)                [{IndexM2PosM Model.daughtersL.LA1}
( 22)                 {IndexM2PosM Model.eq}]
( 23)             else
( 24)                [{IndexM2PosM Model.daughtersL.LA1}
( 25)                 {IndexM2PosM Model.daughtersL.LA2}]
( 26)             end
( 27)    ...
( 28) end
```

Figure 8.16.: `"ROrder"` node constraint functor

The reversed constraint parser can be used e.g. for debugging: by generating all possible linearizations for a multiset of words, the grammar writer can quickly spot overgeneration. It can also be applied for generation from a set of semantic literals, but here, it is not at all clear how many words are required to realize the literals before constraint solving. First attempts to cope with this can be found in (Debusmann 2004*b*) and (Pelizzoni & das Gracas Volpe Nunes 2005). Another smart approach based on TAG is described in (Koller & Striegnitz 2002).

8.6. Runtime

In chapter 6, we have shown that XDG is NP-hard. However, in practice, the implementation of XDG as the XDK constraint parser fares better than expected, at least for handcrafted grammars.

8.6.1. Handcrafted Grammars

Handcrafted grammars can already be parsed reasonably fast. For example, using a test set of 60 sentences ranging from 4-44 words, we have profiled the grammar `diss.ul` from the XDK distribution, which implements the grammar of part III of the thesis, with all its ten dimensions (ID, LP, ID/LP, PA, SC, PA/SC, PS, IS, ID/PA and PS/IS). In the table below, we show the minima, maxima and averages of the number of words, the time required for solving

("Time (s)"), the number of solutions, failures and the search tree depth ("Sol/Fail/Depth"), the number of lexical entries per word, i.e., their lexical ambiguity ("Amb"), and the number of constraint variables ("Vars") and propagators ("Props") introduced by the XDK constraint parser on an AMD Athlon with 1.2 GHz and 512 MBytes of RAM:

	Words	Time (s)	Sol/Fail/Depth	Amb	Vars	Props
min	4	0.480	1/0/1	1	13547	63950
max	44	32.880	2/2/3	36	1342027	3501430
average	9.22	2.440	1.05/0.2/1.25	3.00	78899.7	279879.0

(8.20)

TDG grammars, using only two dimensions (ID/LP), can be parsed more efficiently. For example, the grammar developed in (Debusmann 2001), which is called `Diplom.ul` in the XDK distribution, has the following profile:

	Words	Time (s)	Sol/Fail/Depth	Amb	Vars	Props
min	3	0.020	0/0/1	1	603	2318
max	64	8.360	6/2/6	9	12803	338184
average	7.89	0.184	1.14/0.36/1.48	2.12	1582.71	14436.9

(8.21)

Optimizing the XDK constraint parser was not in the focus of the research for thesis. Hence, the parser is almost unoptimized, and there is ample room for optimization, which we see as our next steps. Our ideas include extensive profiling of the parser, the advent of global constraints, and the use of the new and more efficient *Gecode* constraint library (Schulte & Stuckey 2004).

8.6.2. Automatically Induced Grammars

We have also applied the XDK constraint parser to grammars induced from treebanks. Bojar (2004) describes a series of experiments of inducing a large-scale grammar from *Prague Dependency Treebank* (*PDT*) (Böhmová, Hajič, Hajičová & Hladká 2001) for Czech. His grammars heavily overgenerated, which lead, in combination with exhaustive search of the XDK parser, to a combinatorial explosion. Möhl (2004) induced grammars from the *TIGER treebank* (Brants 1999) for German, using an induction technique developed in (Korthals 2003), but the resulting grammars could also only be parsed inefficiently by the XDK parser, and suffered from undergeneration.

A major problem of the approaches of Bojar and Möhl was the lack of any statistical support, e.g. by *guided search*. To find out whether guided search can improve the efficiency of XDK large-scale parsing, Narendranath (2004) experimented with grammars induced from the *Penn Treebank* (*PTB*) (Marcus, Santorini & Marcinkiewicz 1993) for English, employing for the first time the ideas for guided search developed for XDG in (Dienes, Koller & Kuhlmann 2003). Her grammars heavily overgenerated, like Bojar's, but she could successfully show that guided search can considerably prune the search space in comparison to exhaustive search: for unseen sentences, the time for enumerating the solutions could be reduced by factor 5, the number of failures by factor 50, and the number of solutions by factor 1000. For already seen sentences, the effect was even more positive: 15 times less solutions, 100 times less failures, and 1000 times less solutions. We conjecture that the addition of other

statistical techniques such as *supertagging* (Joshi & Bangalore 1994, Clark & Curran 2004) could further boost the efficiency of XDK large-scale parsing.

8.7. Summary

This chapter introduced the constraint parser of the XDK. We illustrated how to model multigraphs using finite sets of integers, and how the CSP for the constraint parser is set up by an Oz script making use of both the functionality of the lattice functors and the principles from the extensible principle library of the XDK. The principles are realized using node and edge constraint functors. The constraint parser can be adapted to act in a reversible way, i.e., also for generation. The parser is already reasonably fast on smaller, handcrafted grammars, but could not be shown to scale up to large-scale parsing. This is not surprising given that the parser is yet almost unoptimized, and lacks statistical support. The multitude of possibilities for optimization makes us optimistic that large-scale parsing is possible with XDG, and attempting this will be one of our next steps.

Part III.

Application

9. Syntax

In this part of the thesis, we finally apply XDG to natural language. We present an example XDK metagrammar for a fragment of English, which covers the linguistic aspects of syntax, semantics and phonology. This grammar clearly demonstrates the modularity of XDG with respect to grammar development, allowing us to develop the dimensions of syntax (this chapter), semantics (chapter 10) and phonology (chapter 11) as independent modules, whose relation we establish subsequently through the syntax-semantics interface and the phonology-semantics interface (chapter 12). We will show that by this modularity, the phenomena covered by the grammar need not be explicitly specified, but rather emerge from the intersective demands of its dimensions. The grammar covers control and raising constructions, auxiliaries, passives, questions, topicalization, subordinate sentences and relative clauses. We have deliberately left out coordination for simplicity. An account of coordination without ellipsis in XDG can be found in (Bader et al. 2004). We must leave an account for coordination including ellipsis to future work.

This chapter introduces the dimensions of syntax, whose position in the overall architecture of the grammar is displayed in Figure 9.1. Following the account of German syntax in TDG described in (Duchier & Debusmann 2001, Debusmann 2001), we model syntax using the following three dimensions:

1. *Immediate Dominance* (ID)
2. *Linear Precedence* (LP)
3. ID/LP

where the ID dimension models the hierarchical syntactic structure by an unordered tree labeled by *grammatical functions*, and the LP dimension models word order by ordered projective trees labeled by topological fields. The ID/LP dimension acts as the interface of the ID and LP dimensions.[1]

9.1. Immediate Dominance Dimension

The models of the Immediate Dominance (ID) dimension are unordered trees whose edge labels represent grammatical functions like subject and object. We call an ID analysis ID *tree*,

[1] In the original TDG account, the relation between the ID and LP dimensions is constrained without the definition of an additional ID/LP dimension, which we introduce here for modularity.

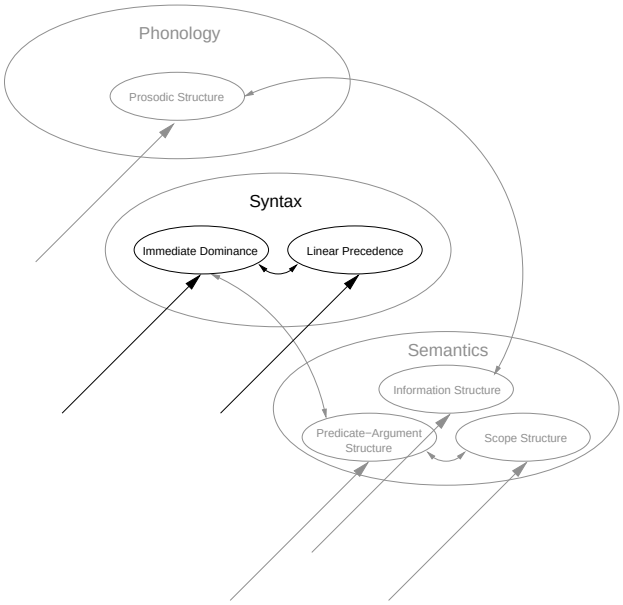


Figure 9.1.: Syntax in the overall architecture of the example grammar

and show an example ID tree of the sentence below in Figure 9.2:[2]

Peter admires the woman who smiles. (9.1)

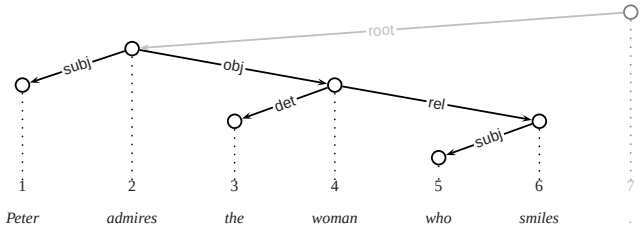


Figure 9.2.: ID tree of *Peter admires the woman who smiles.*

As in chapter 2, the ID tree is equipped with an additional root node corresponding to the end-of-sentence marker (here: the full stop), which is connected to the finite verb (here: *admires*) by an edge labeled root. *Peter* is the subject of *admires*, and *woman* the object. *the* is the determiner of *woman*, and *woman* is modified by the relative clause (edge label rel) *who smiles*. In the relative clause, *smiles* is the head and the subject is *who*.

Figure 9.3 shows another example ID tree, this time of the question

Who does he say Mary thinks smiles? (9.2)

where the finite verb *does* has the subject *he* and the base form infinitival complement (edge label vbse) *say*. *say* in turn is the head of the subordinate clause headed by *thinks*, which is the

[2]For visualization, we have to fix an order on the nodes. For clarity, we choose the order of the corresponding words in the sentence.

head of another subordinate clause headed by *smiles*. The subject of *thinks* is *Mary* and that of *smiles* is the wh-pronoun *who*. This example demonstrates that since ID trees are unordered, no compromises have to be made to bring word order in line with the intuitive analysis of the sentence in terms of grammatical functions. *Unbounded dependencies* such as the dependency between *smiles* and its subject *who* are not considered "unbounded" at all, since order and thus the distance between the words is simply irrelevant on the ID dimension.

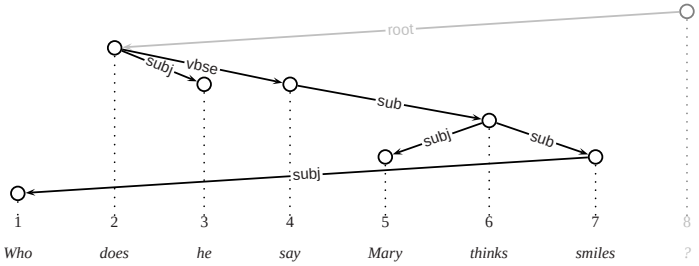


Figure 9.3.: ID tree of *Who does he say Mary thinks smiles?*

As a third example, we show the ID analysis of the sentence

Peter persuades Mary to smile. (9.3)

in Figure 9.4. This is an example of a subject-to-object control construction, where the object *Mary* of the control verb *persuades* is regarded as the "deep subject" of the embedded verb *smile*. As the analysis shows, control relations are not represented on the ID dimension. We think that they belong on the "deeper" dimension of *predicate-argument structure* (PA) instead (see section 10.1), and not on the more "surface-oriented" ID dimension. Another reason is that if we modeled control on the ID dimension, we would have to give up the invariant that ID analyses are trees, which would severely complicate the interface between the ID and LP dimensions.

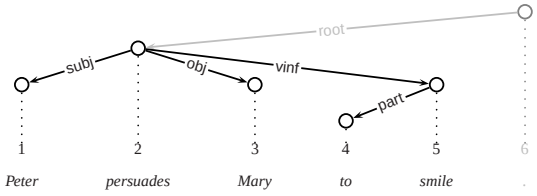


Figure 9.4.: ID tree of *Peter persuades Mary to smile.*

9.1.1. Types

We continue the explanation of the ID dimension by introducing its types of edge labels and attributes.

Edge Labels. We define the type of edge labels on the ID dimension as follows:

```
deftype "id.label" {adj adv comp det iobj obj part pmod pobj1 pobj2 prepc
                    rel root sub subj vbse vinf vprt}
deflabeltype "id.label"
```
(9.4)

and show an overview of the edge labels and their corresponding grammatical functions in Figure 9.5. They consist of:

- standard grammatical functions: adjective (`adj`), adverb (`adv`), determiner (`det`), indirect object (`iobj`), direct object (`obj`), and subject (`subj`)
- `comp`, the complementizer of a subordinate clause (e.g. *that* in *Peter says that Mary laughs.*)
- edge labels concerned with prepositions. We distinguish prepositional objects (edge labels `pobj1` or `pobj2`) and prepositional modifiers (`pmod`). The complement of a preposition has label `prepc` (e.g. *Peter* in *to Peter*). We distinguish `pobj1` and `pobj2` for examples like *A book is given to Peter by Mary*, where two prepositional objects must be distinguished (*to Peter* and *by Mary*).
- `rel` and `sub`, the incoming edge labels of finite verbs heading a relative clause and a subordinate clause, respectively
- `vbse`, `vinf` and `vprt`, the labels of non-finite verbs (`vbse`: base form infinitive, `vinf` full infinitive with particle *to*, `vprt`: past participle), and `part`, the label of particles
- `root`, the incoming edge label of the finite verb heading the sentence

Attributes. We define the attributes of the ID dimension with respect to the type of *agreement tuples* `"id.agr"` consisting of person (first, second or third), number (singular or plural), gender (masculine, feminine, neuter) and case (nominative or accusative):

```
deftype "id.person" {first second third}
deftype "id.number" {sg pl}
deftype "id.gender" {masc fem neut}
deftype "id.case" {nom acc}
deftype "id.agr" tuple("id.person" "id.number" "id.gender" "id.case")
```
(9.5)

Furthermore, we define the type `"id.pagr"` of preposition types, which consists of the prepositions covered by the grammar:

```
deftype "id.pagr" {at by in of on to with}
```
(9.6)

The non-lexical attributes consist of the two attributes `agr` (of type `"id.agr"`) and `pagr` (of type `"id.pagr"`). `agr` denotes the agreement tuple and `pagr` the preposition type selected for the node:

```
defattrstype {agr: "id.agr"
              pagr: "id.pagr"}
```
(9.7)

edge label	grammatical function
adj	adjective
adv	adverb
comp	complementizer
det	determiner
iobj	indirect object
obj	object
part	particle
pmod	prepositional modifier
pobj1	prepositional object 1
pobj2	prepositional object 2
prepc	complement of a preposition
rel	relative clause
root	root
sub	subordinate clause
subj	subject
vbse	base form infinitive
vinf	full infinitive
vprt	past participle

Figure 9.5.: ID edge labels and corresponding grammatical functions

The lexical attributes include the attributes `in` and `out` (representing the in and out valencies of the word), `agrs` (the set of licensed agreement tuples), `pagrs` (the set of licensed preposition types), and `pobj1` and `pobj2` (the preposition types licensed for `pobj1` and `pobj2` dependents), respectively:

```
defentrytype {in: valency("id.label")
              out: valency("id.label")
              agrs: iset("id.agr")
              pagrs: iset("id.pagr")
              pobj1: iset("id.pagr")
              pobj2: iset("id.pagr")}
```
(9.8)

9.1.2. Principles and Lexical Classes

The ID dimension is further characterized by a set of principles and lexical classes.

Models. We start by constraining the models on the ID dimension to be trees using the *Graph principle* and the *Tree principle*:

```
useprinciple "principle.graph" { dims {D: id} }
useprinciple "principle.tree" { dims {D: id} }
```
(9.9)

Subcategorization, Modification and Categorization. With the *Valency principle*, we model *categorization*, *subcategorization* and *modification*. We apply the principle as follows, using by the lexical attributes `in` and `out`:

```
useprinciple "principle.valency" {
  dims {D: id}
  args {In: _.D.entry.in
        Out: _.D.entry.out}}
```
(9.10)

Subcategorization determines the number of syntactic dependents of a node using the lexical attribute `out`. For example, the lexical class `"id_fin"` states that finite verbs always require a subject:

```
defclass "id_fin" {
  dim id {out: {subj!}}}
```
(9.11)

Modification is also modeled using the Valency principle. For example, the following lexical classes state that main verbs (`"id_main"`) can be modified by arbitrary many adverbs and prepositional modifiers, and that auxiliary verbs (`"id_aux"`) cannot be modified:

```
defclass "id_main" {
  dim id {out: {adv* pmod*}}}

defclass "id_aux" {
  dim id {out: {}}}
```
(9.12)

Categorization states constraints on the incoming edge labels of the nodes using the lexical attribute `in`. For example, a finite verb can either be the root of a sentence, the head of a subordinate clause or the head of a relative clause, which we capture in the following lexical classes:

```
defclass "id_fin_root" {
  "id_fin"
  dim id {in: {root?}}}

defclass "id_fin_sub" {
  "id_fin"
  dim id {in: {sub?}
          out: {comp?}}}

defclass "id_fin_rel" {
  "id_fin"
  dim id {in: {rel?}}}
```
(9.13)

where as the root of a sentence, the finite verb must have incoming edge label `root`, as the head of a subordinate clause `sub`[3], and as the head of a relative clause `rel`.

Agreement. We realize the morphological *agreement* of heads and dependents in terms of person, number, gender and case using the *Agr principle* and the *Agreement principle* (principles 8 and 9 in chapter 4):

```
useprinciple "principle.agr" {
  dims {D: id}
  args {Agr: _.D.attrs.agr
        Agrs: _.D.entry.agrs}}
useprinciple "principle.agreement" {
  dims {D: id}
  args {Agr1: ^.D.attrs.agr
        Agr2: _.D.attrs.agr
        Agree: {det subj}}}
```
(9.14)

By the Agr principle, the value of the non-lexical attribute `agr` must be an element of the lexical attribute `agrs`. By the Agreement principle, for all edges labeled `det` and `subj`, the head must agree with its dependent, i.e., the values of the non-lexical attribute `agr` of the head and its dependent must be the same to exclude e.g. *a researchers* or *most researcher*. Similarly, subjects must agree with their verbal heads to exclude e.g. *He sleep* or *They sleeps*.

[3] As the head of a subordinate clause, it can also have an optional complementizer.

Government. *Government* is also concerned with agreement. In XDG, we define government as describing the fact that some heads "govern" the agreement of their dependents.[4] For instance, finite verbs govern the case of their subject to be nominative. We model government using the *Government principle*, which has the declarative semantics that for each edge from v to v' labeled l, the agreement tuple of the dependent v' (given by the non-lexical attribute *agr*) must be an element of the set of agreement tuples licensed by the head v for label l (given by the lexical attribute *govern*).

Principle 15 (Government)**.**

$$government_d = \forall v, v' : \forall l : (d\ v').agr \in (d\ v).lex.govern.l \tag{9.15}$$

In the XDK, we can specify the value of *govern* non-lexically to minimize the lexical description, stating that all subjects are governed to have nominative agreement, and all objects and complements of a preposition to have accusative agreement. No other dependents are constrained.

```
useprinciple "principle.government" {
  dims {D: id}
  args {Agr2: _.D.attrs.agr
        Govern: {subj: ($ nom)
                 obj: ($ acc)
                 prepc: ($ acc)}}}
```
(9.16)

Our grammar reuses the idea of government to make verbs govern the preposition of their prepositional objects. For example, the ditransitive verb *give* only licenses the prepositional object *to* for its `pobj1` dependent, as indicated below:

Peter gives a book to Mary .
**Peter gives a book at Mary .* (9.17)

We model this using the Government principle a second time, in addition to a second use of the Agr principle to select for each node a preposition type from the set of licensed preposition types:

```
useprinciple "principle.agr" {
  dims {D: id}
  args {Agr: _.D.attrs.pagr
        Agrs: _.D.entry.pagrs}}
useprinciple "principle.government" {
  dims {D: id}
  args {Agr2: _.D.attrs.pagr
        Govern: {pobj1: ^.D.entry.pobj1
                 pobj2: ^.D.entry.pobj2}}}
```
(9.18)

Here, the lexical attribute `pobj1` determines the licensed preposition types for `pobj1` dependents, and `pobj2` for `pobj2` dependents. To model the contrast (9.17) above, *give* would thus set its lexically attribute `pobj1` to {`to`} to state that it only accepts `pobj1` dependents with preposition type *to*.

[4] Government is not uniformly defined in the literature. Other definitions can be found e.g. for GB in (Chomsky 1981), or for MTT in (Mel'čuk 1988).

9.2. Linear Precedence Dimension

We describe word order using the Linear Precedence (LP) dimension, whose models are ordered and projective trees, and whose edges are labeled by *topological fields*. We call LP analyses LP *trees*. Topological fields stem from German descriptive linguistics (Herling 1821, Erdmann 1886), and have recently been rediscovered in frameworks such as HPSG (Penn 1999, Kathol 2000) and MTT (Gerdes & Kahane 2001). In the theory, sentences are subdivided into sequences of substrings, and these substrings are called topological fields. For German, the basic topological field structure is the following:

Vorfeld	left bracket	Mittelfeld	right bracket	Nachfeld

(9.19)

where the *Vorfeld* ("pre-field") typically contains the subject, the *Mittelfeld* ("mid-field") the other nominal complements such as indirect and direct objects, and the *Nachfeld* ("post-field") subordinate clauses or extraposed relative clauses. The Mittelfeld is surrounded by the finite verb, often called the *left bracket*, and its non-finite verbal dependents in the *right bracket*. In the Mittelfeld, the nominal complements can be freely permuted.[5]

In (Duchier & Debusmann 2001) and (Debusmann 2001), topological fields theory serves as the basis for an elegant analysis of German word order phenomena on the LP dimension of TDG. Figure 9.6 shows an example TDG LP analysis of the following German sentence:

Maria	*hat*	*dem*	*Mann*	*heute*	*einen*	*Korb*	*gegeben,*	*der*	*lacht.*
Mary	*has*	*the*	*man*	*today*	*a*	*basket*	*given,*	*who*	*laughs.*

"Mary has given the man who laughs a basket today." (9.20)

where the finite verb *hat* is in the left bracket. Its subject *Maria* is in the Vorfeld (edge label vf), and the Mittelfeld (mf) is filled by the indirect object *dem Mann*, the adverb *heute* and the direct object *einen Korb*. The right bracket (rbf) is filled by the past participle *gegeben* and the Nachfeld (nf) by the extraposed relative clause *der lacht*.

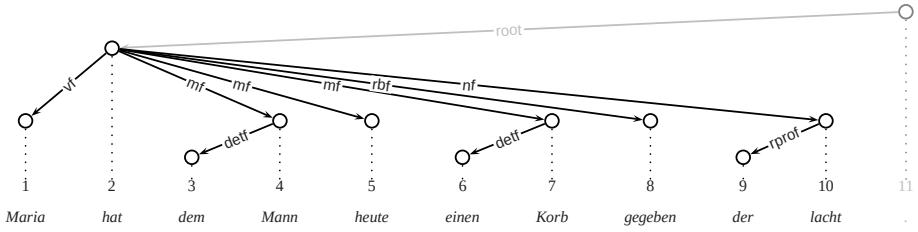


Figure 9.6.: TDG LP tree of *Maria hat dem Mann heute einen Korb gegeben, der lacht.*

In this thesis, we show that topological fields theory can also be transferred to English. As an example, Figure 9.7 shows the LP analysis of the translation of (9.20), where the finite verb *has* is in the left bracket, and its subject *Mary* in its Vorfeld, as in the German example. The past participle *given* is however not in the right bracket but is also positioned in the left bracket (edge label lbf). The indirect object *the man* and the direct object *a basket* are both in

[5] This is a simplification: generally, the elements of the Mittelfeld can be freely permuted, but there are exceptions, e.g. the order of pronouns, which is fixed.

the Mittelfeld (mf1 and mf2). As relative clauses cannot be extraposed in English, the relative clause *who laughs* directly follows the modified noun *man*. The adverb *today* cannot be part of the Mittelfeld. It is positioned at the end of the sentence into the field for adverbs of time (tadvf).

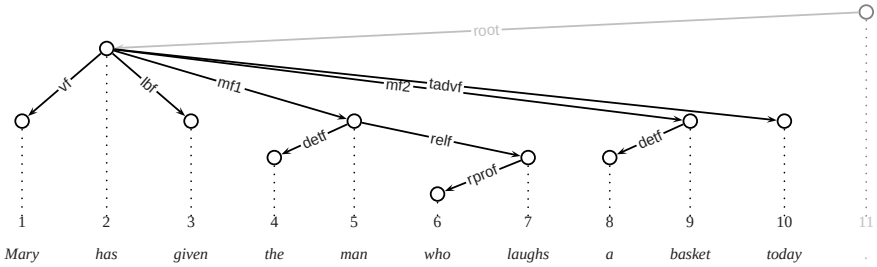


Figure 9.7.: LP tree of *Mary has given the man who laughs a basket today.*

9.2.1. Types

Edge Labels. The type of edge labels on the LP dimension is defined as follows:

```
deftype "lp.label" {adjf compf detf fadvf lbf mf1 mf2 nf padjf padvf prepcf
                    rbf relf root rprof tadvf vf vvf}
deflabeltype "lp.label"
```
(9.21)

We show an overview of the edge labels and their corresponding topological fields in Figure 9.8. They consist of:

- fields corresponding directly to the fields of topological fields theory: `vf` (Vorfeld), `lbf` (left bracket field), `mf1` and `mf2` (Mittelfeld), `rbf` (right bracket field) and `nf` (Nachfeld). Contrary to German, where the words in the Mittelfeld can be freely permuted, English permits less word order variation: indirect objects must always precede direct objects. We capture this by splitting the Mittelfeld into two fields: `mf1` for indirect objects and `mf2` for direct objects. The left bracket field (`lbf`) is the landing site for base form infinitives and past participles, the right bracket field (`rbf`) for full infinitives, and the Nachfeld (`nf`) for subordinate clauses.

- the *Vor-Vorfeld* (“ pre-pre-field”) (`vvf`), to the left of the Vorfeld, for fronted material such as wh-pronouns (e.g. *who* in *Who does Mary like?*) and for particles (e.g. *to* in *to believe*), a field for complementizers in subordinate clauses (`compf`), and a field for relative pronouns in relative clauses (`rprof`)

- three fields for adverbs and prepositional modifiers: `fadvf` for adverbs of frequency (e.g. *often* in *Peter often sleeps*), `padvf` for adverbs of place or manner (e.g. *carefully* in *Peter reads the book carefully.*), and `tadvf` for adverbs of time (e.g. *now* in *Peter reads the book carefully now*).

- four fields for noun phrases: the determiner field (`detf`), the adjective field (`adjf`), the field (`padjf`) for prepositional modifiers of nouns, which we call *prepositional adjectives*, and the relative clause field (`relf`)
- a field for the complement of a preposition (`prepcf`)
- `root`, the label of the edge from the root (e.g. the full stop) to the finite verb

edge label	topological field
`adjf`	adjective field
`compf`	complementizer field
`detf`	determiner field
`fadvf`	adverbs of frequency field
`lbf`	left bracket field
`mf1`	Mittelfeld 1
`mf2`	Mittelfeld 2
`nf`	Nachfeld
`padvf`	adverbs of place or manner field
`padjf`	prepositional adjective field
`prepcf`	complement of a preposition field
`rbf`	right bracket field
`relf`	relative clause field
`root`	root
`rprof`	relative pronoun field
`tadvf`	adverbs of time field
`vf`	Vorfeld
`vvf`	Vor-Vorfeld

Figure 9.8.: LP edge labels and corresponding topological fields

Attributes. The LP dimension defines the following lexical attributes:

```
deftype "lp.label1" "lp.label" | {"^"}
defentrytype {in: valency("lp.label")
              out: valency("lp.label")
              order: set(tuple("lp.label1" "lp.label1"))}
```
(9.22)

where `in` and `out` stipulate the licensed incoming and outgoing edges and `order` a strict partial order on the outgoing edges and the special anchor label `"^"`.

9.2.2. Principles and Lexical Classes

Models. We constrain the models of the LP dimension to be projective trees:

```
useprinciple "principle.graph" { dims {D: lp} }
useprinciple "principle.tree" { dims {D: lp} }
useprinciple "principle.projectivity" { dims {D: lp} }
```
(9.23)

Topological Valency and Order. We use the *Valency principle* and the *Order principle* (principle 7 in chapter 4) to constrain the topological structure induced by the nodes. The Valency principle is lexicalized by the lexical attributes `in` and `out`, and the Order principle by the lexical attribute `order`:

```
useprinciple "principle.valency" {
  dims {D: lp}
  args {In: _.D.entry.in
        Out: _.D.entry.out}}
useprinciple "principle.order" {
  dims {D: lp}
  args {Order: _.D.entry.order}}
```
(9.24)

In the following table, we show the topological structure induced by finite verbs and full infinitives:

Vor-Vorfeld			Vorfeld	left bracket		Mittelfeld		right bracket	Nachfeld		
compf	rprof	vvf	vf	fadvf	lbf	mf1	mf2	rbf	padvf	tadvf	nf

(9.25)

where the Vor-Vorfeld contains at most one fronted node: a complementizer in the `compf` of a subordinate clause, a relative pronoun in the `rprof` of a relative clause, a particle in the `vvf` of a full infinitive, or any other fronted node in the `vvf` of a matrix clause. The Vorfeld `vf` and the Mittelfeld (`mf1` and `mf2`) contain subjects, indirect objects and direct objects, respectively. The left bracket can be filled by arbitrary many adverbs of frequency in the `fadvf`, followed by at most one base form infinitive or past participle in the left bracket field `lbf`. The right bracket contains arbitrary many full infinitives or prepositional objects in the right bracket field `rbf`. The Nachfeld contains arbitrary many adverbs of place or manner in the field `padvf`, followed by arbitrary many adverbs of time in the `tadvf`, and followed by at most one subordinate clause in the `nf`.

To realize this topological structure, we first define the following lexical class:

```
defclass "lp_fin" {
  dim lp {out: {lbf? fadvf* mf1? mf2? rbf* padvf* tadvf* nf?}}}
```
(9.26)

where we state that finite verbs may have at most one dependent in the left bracket field (`lbf`), arbitrary many dependents in the fields for adverbs (`fadvf`, `padvf` and `tadvf`), at most one in `mf1` and at most one in `mf2`, arbitrary many in the right bracket field (`rbf`) and at most one in the Nachfeld (`nf`).

Depending on their context, we further specify the topological structure of finite verbs by the following lexical classes, where `"lp_fin_root"` describes heads of matrix clauses, which have incoming edge label `root` and license at most one dependent in the Vor-Vorfeld and precisely one (the obligatory subject) in the Vorfeld. `"lp_fin_sub"` describes finite verbs heading a subordinate clause, which can be fronted into the Vor-Vorfeld or extraposed into the Nachfeld, and which license at most one complementizer and at most one dependent in the Vorfeld. Contrary to matrix clauses, the Vorfeld is not obligatory since it could also be extracted, as in example (9.2), where the subject of *smiles* is extracted. Finally, `"lp_fin_rel"` describes finite verbs heading a relative clause. They require one dependent in the relative

pronoun field and license at most one in the Vorfeld:

```
defclass "lp_fin_root" {
  "lp_fin"
  dim lp {in: {root?}
          out: {vvf? vf!}}}

defclass "lp_fin_sub" {
  "lp_fin"
  dim lp {in: {vvf? nf?}
          out: {compf? vf?}}}

defclass "lp_fin_rel" {
  "lp_fin"
  dim lp {in: {relf?}
          out: {rprof! vf?}}}
```
(9.27)

Full infinitives (lexical class `"lp_vinf"`) can only land in the right bracket field. Their topological structure is very similar to that of finite verbs, except that they do not license a Vorfeld dependent. Base form infinitives (`"lp_vbse"`) and past participles (`"lp_vprt"`) can only land in the left bracket field and license at most one outgoing edge to a dependent in their left bracket field:

```
defclass "lp_vinf" {
  dim lp {in: {rbf?}
          out: {vvf! lbf? fadvf* mf1? mf2? rbf* padvf* tadvf* nf?}}}

defclass "lp_vbse" {
  dim lp {in: {lbf?}
          out: {lbf?}}}

defclass "lp_vprt" {
  dim lp {in: {lbf?}
          out: {lbf?}}}
```
(9.28)

The order of the topological dependents of verbs is defined in the following three lexical classes for main verbs (`"lp_main"`), auxiliaries (`"lp_aux"`) and question auxiliaries (`"lp_qaux"`). Their only difference is the position of the verb (`"^"`) with respect to its dependents:

1. Main verbs must be positioned to the right of the field for adverbs of frequency `fadvf` and to the left of the Mittelfeld (Figure 9.9):

   Peter often admires Mary.
   **Peter admires often Mary.*
   **Peter Mary often admires.*
   (9.29)

   ```
   defclass "lp_main" {
     dim lp {order: <compf rprof vvf vf fadvf "^" lbf mf1 mf2 rbf
                     padvf tadvf nf>}}
   ```
   (9.30)

2. Auxiliaries must be positioned directly to the left of the `fadvf`. Their complement must end up in the left bracket field to the right of the `fadvf` (Figure 9.10):

   Peter has often admired Mary.
   **Peter has admired often Mary.*
   (9.31)

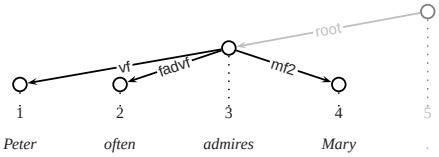


Figure 9.9.: LP tree of *Peter often admires Mary.*

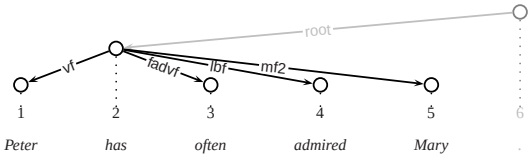


Figure 9.10.: LP tree of *Peter has often admired Mary.*

```
defclass "lp_aux" {
  dim lp {order: <compf rprof vvf vf "^" fadvf lbf mf1 mf2 rbf
                  padvf tadvf nf>}}
```
(9.32)

3. The position of question auxiliaries is even further to the left, between the Vor-Vorfeld and the Vorfeld (Figure 9.11):

Whom has Peter often admired?
**Whom Peter has often admired?*
**Whom Peter often has admired?* (9.33)

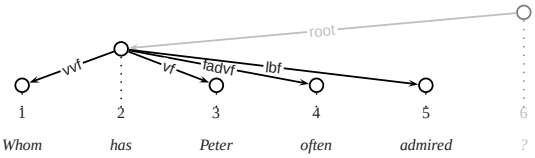


Figure 9.11.: LP tree of *Whom has Peter often admired?*

```
defclass "lp_qaux" {
  dim lp {order: <compf rprof vvf "^" vf fadvf lbf mf1 mf2 rbf
                  padvf tadvf nf>}}
```
(9.34)

We now turn to the topological structure induced by nouns, which is much simpler: at most one determiner is followed by arbitrary many adjectives, by the noun itself, at most one prepositional adjective and at most one relative clause. Here is an example, where *of the researcher* is the prepositional adjective and the relative clause *which hums* modifies the common noun *product* (Figure 9.12):[6]

a nice little product of the researcher which hums (9.35)

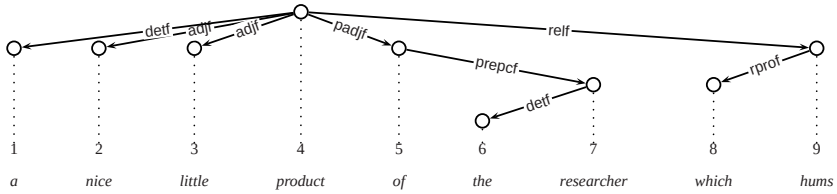


Figure 9.12.: LP tree of *a nice little product of the researcher which hums*

We realize this topological structure with the following lexical class, which also states that nouns can either be fronted into the Vor-Vorfeld, land in the Vorfeld or in the Mittelfeld, or be the complement of a preposition:

```
defclass "lp_noun" {
  dim lp {in: {vvf? vf? mf1? mf2? prepcf?}
          out: {detf? adjf* padjf? relf?}
          order: <detf adjf "^" padjf relf>}}
```
(9.36)

Prepositions induce an even simpler topological structure where the preposition must precede its complement in the `prepcf`:

```
defclass "lp_prep" {
  dim lp {out: {prepcf?}
          order: <"^" prepcf>}}
```
(9.37)

Prepositional objects can either land in the right bracket field, or they can be fronted into the Vor-Vorfeld or the relative pronoun field.

```
defclass "lp_pobj" {
  "lp_prep"
  dim lp {in: {rbf? vvf? rprof?}}}
```
(9.38)

9.3. ID/LP Dimension

The ID/LP dimension constitutes the interface between the ID and LP dimensions, constraining their relation. The models of the ID/LP dimension are graphs without edges.

9.3.1. Types

Attributes. We define only one lexical attribute: `blocks`, whose type is a set of ID edge labels:

```
defentrytype {blocks: set("id.label")}
```
(9.39)

[6] We will establish the partial agreement of the relative pronoun and its modified noun, which is responsible for ruling out the analysis where *which hums* modifies *researcher* instead of *product*, on the ID/PA dimension in chapter 12.

9.3.2. Principles and Lexical Classes

Climbing. The relation between the ID and LP dimensions is mainly one of flattening: LP trees must be flatter than ID trees. We express this using the *Climbing principle* (cf. principle 13 in chapter 5) to model the idea that nodes more deeply embedded on the ID dimension can be extracted and land higher up on the LP dimension:

```
useprinciple "principle.climbing" {
  dims {D1: lp
        D2: id}}
```
(9.40)

Without the Climbing principle, the relation between the ID and LP dimensions would be too loose: for example, for the sentence below:

Peter likes a nice woman. (9.41)

we would license the wrong ID/LP analysis shown in Figure 9.13, where the adjective *nice* modifies *Peter* instead of *woman* on the ID dimension. The reason for this is that *nice* does not not climb up to a transitive head (here: either *Peter* or *likes*) on the ID dimension.

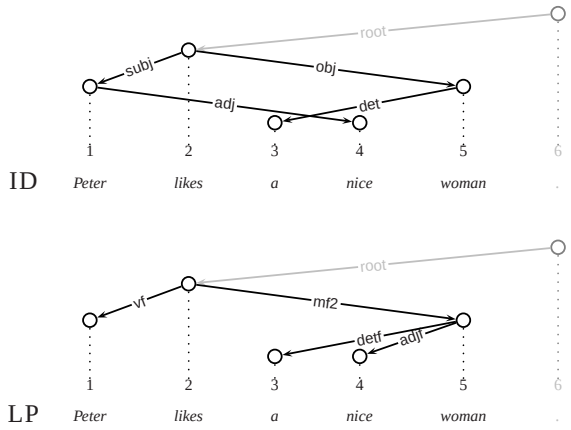


Figure 9.13.: Wrong ID/LP analysis ruled out by the Climbing principle

Barriers. Climbing alone is not sufficient to bring the ID and LP dimensions together. For example, we must to prevent adverbs from climbing out of subordinate clauses and relative clauses, as in the analysis given in Figure 9.14 for the sentence below:

Peter always likes Mary who smiles. (9.42)

where the adverb *always* has wrongly been extracted out of a relative clause into the field `fadvf` of the matrix verb *likes*.

We realize restrictions like these with the *Barriers principle*, which has the declarative semantics that for each node v, no node v'' between v and its transitive head v' on may "block" v from migrating up. In the example, the nodes between *always* and its transitive head *likes* are *Mary* and *smiles*, where the adverb *always* is "blocked" by the finite verb *smiles*.

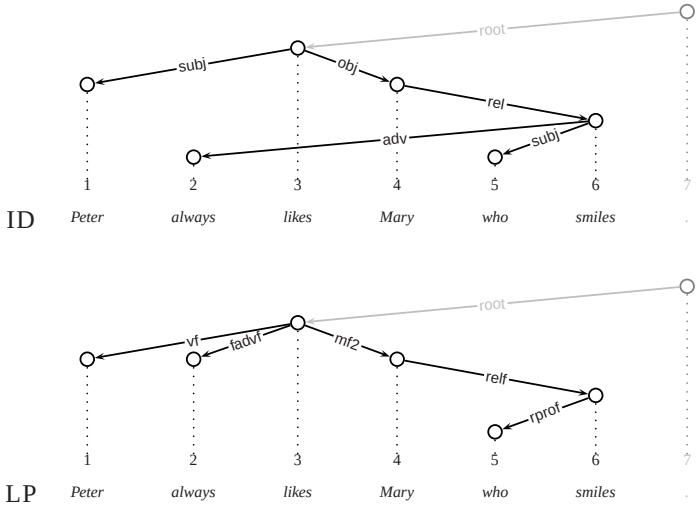


Figure 9.14.: Wrong ID/LP analysis ruled out by the Barriers principle

Principle 16 (Barriers)**.**

$$\begin{aligned} &barriers_{d_1,d_2,d_3} = \forall v, v' : v' \rightarrow_{d_1} v \Rightarrow \forall v'' : v' \rightarrow^{+}_{d_2} v'' \wedge v'' \rightarrow^{+}_{d_2} v \Rightarrow \\ &\forall v''' : \forall l : v''' \xrightarrow{l}_{d_2} v \Rightarrow l \notin (d_3\ v'').lex.blocks \end{aligned} \tag{9.43}$$

In our metagrammar, we apply the Barriers principle using the lexical attribute `blocks`:

```
useprinciple "principle.barriers" {
  dims {D1: lp
        D2: id
        D3: idlp}
  args {Blocks: _.D3.entry.blocks}}
```
(9.44)

Using the Barriers principle, we can rule out the analysis of Figure 9.14 with the lexical class `"idlp_fin"`, where we stipulate that finite verbs such as *smiles* in Figure 9.14 block adverbs, complementizers, prepositional modifiers, subordinate clauses and non-finite verbs:

```
defclass "idlp_fin" {
  dim idlp {blocks: {adv comp pmod sub vbse vinf}}}
```
(9.45)

As another example, nouns block all their dependents, including relative clauses, to model that their extraction is forbidden in English:

```
defclass "idlp_noun" { dim idlp {blocks: {det adj pmod rel}} }
```
(9.46)

Linking. Contrary to German, where grammatical functions can often be distinguished morphologically, English crucially relies on word order to do this. As a result, in German, the order of the remaining nominal complements in the Mittelfeld is free, and any nominal complement (i.e., a subject, an indirect or a direct object) can theoretically be positioned in the Vorfeld. In English, on the contrary, the lack of inflection leads to the following two restrictions:

1. the order of the indirect and direct object in the Mittelfeld is fixed: the indirect must precede the direct object

2. the Vorfeld of a finite verb is reserved for its subject

We realize the first restriction with the *LinkingEnd principle* (principle 10 in chapter 4) as follows:

```
useprinciple "principle.linkingEnd" {
  dims {D1: lp
        D2: id
        D3: idlp}
  args {End: {mf1: {iobj}
              mf2: {obj}}}}
```
(9.47)

That is, an `mf1` dependent on the LP dimension must be an indirect object on the ID dimension, and an `mf2` dependent a direct object.

For the second restriction, that the Vorfeld of a finite verb must be reserved for its subject, the LinkingEnd principle does not suffice: it can only be used to state that the Vorfeld must be filled by some subject, but this must not necessarily be its own. For example, consider the analysis in Figure 9.15 of the sentence below:

Who does he say smiles? (9.48)

where the Vorfeld of *does* on the LP dimension is filled by the wrong subject: not by its own subject *who* but by the subject *he* of *smiles* on the ID dimension.

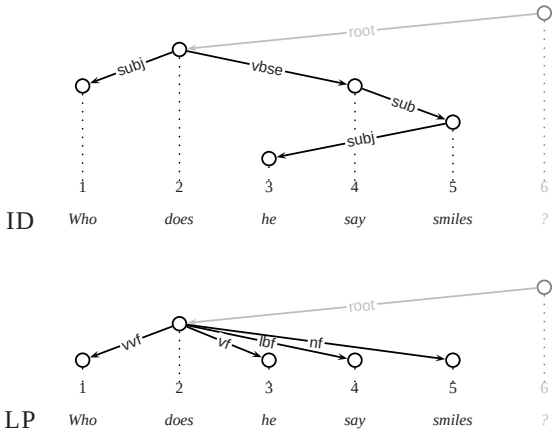


Figure 9.15.: Wrong ID/LP analysis ruled out by the LinkingDaughterEnd principle

We exclude such analyses using the *LinkingDaughterEnd principle*, which has the following declarative semantics. If for an edge from v to v' labeled l on d_1, the value of *linkDaughterEnd* for v and l on d_3 is non-empty, then for at least one edge label l' in this set, there must be an edge from v to v' on d_2 labeled l'.

Principle 17 (LinkingDaughterEnd)**.**

$$\begin{aligned} &\mathit{linkingDaughterEnd}_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} v' \wedge (d_3\ v).\mathit{lex}.\mathit{linkDaughterEnd}.l \neq \emptyset \Rightarrow \\ &\exists l' : l' \in (d_3\ v).\mathit{lex}.\mathit{linkDaughterEnd}.l \wedge v \xrightarrow{l'}_{d_2} v' \end{aligned} \tag{9.49}$$

In our grammar, we use the LinkingDaughterEnd principle as follows:

```
useprinciple "principle.linkingDaughterEnd" {
  dims {D1: lp
        D2: id
        D3: idlp}
  args {End: {vf: {subj}}}}
```
(9.50)

As a result, any edge labeled `vf` from any node v to any other node v' on the LP dimension must be accompanied by a corresponding edge from v to v' labeled `subj` on the ID dimension.

9.4. Emerging Phenomena

At the beginning of this chapter, we claimed that our modular account of syntax would lead to the emergence of a number of interesting syntactic phenomena without further stipulation. In this section, we substantiate this claim by demonstrating the emergence of the phenomena of *topicalization*, *wh questions*, and *pied piping* (Ross 1967).

9.4.1. Topicalization

The grammar allows nominal arguments of verbs to climb up into the Vor-Vorfeld of the matrix verb:

1. The migration of the nominal arguments is not blocked by the Barriers principle, as can be seen from the lexical classes in (9.45) and (9.46), where neither subjects, objects, indirect objects, nor prepositional objects are blocked.

2. The set of licensed incoming edge labels of nouns on LP includes `vvf` (9.36).

This leads to the emergence of the phenomenon of *topicalization*. As an example, consider the sentence below, analyzed in Figure 9.16, where the object *Mary* is topicalized, i.e., climbs up from being the object of *find* on the ID dimension into the Vor-Vorfeld of *tries* on the LP dimension:

Mary, Peter tries to find. (9.51)

9.4.2. Wh questions

Wh questions are analyzed analogously to topicalization. Below is an example, analyzed in Figure 9.17, where the object wh pronoun *whom* is fronted:

Whom does Mary say a man thinks she tries to find? (9.52)

The example also shows that the grammar covers arbitrarily nested unbounded dependencies.

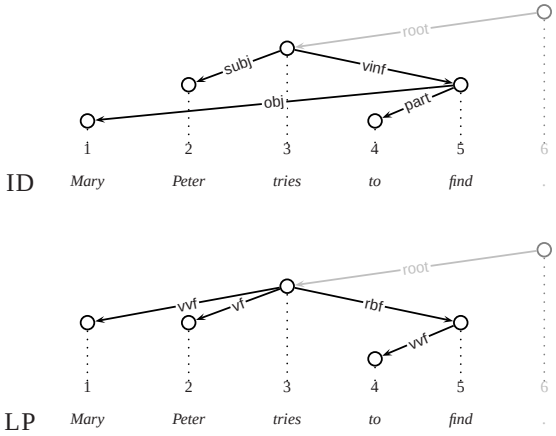


Figure 9.16.: ID/LP analysis of *Mary, Peter tries to find.*

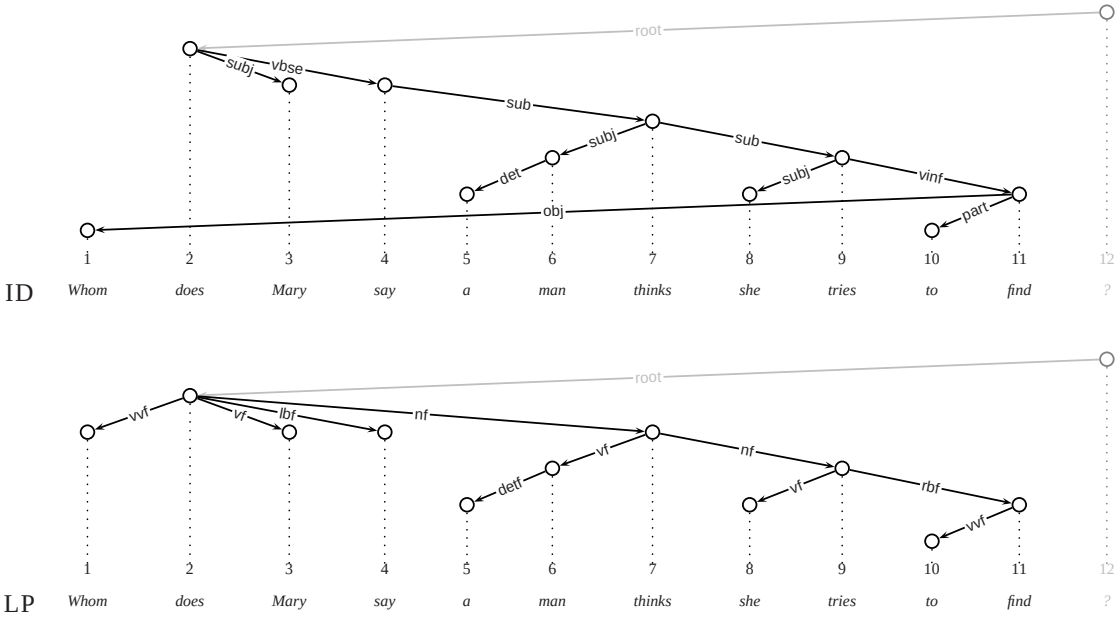


Figure 9.17.: ID/LP analysis of *Whom does Mary say a man thinks she tries to find?*

9.4.3. Pied Piping

Prepositional objects can also be fronted, leading to the emergence of the phenomenon of pied piping. As in relative clauses, prepositional objects can also be fronted, we also obtain relative clause pied piping. We give an example of this below and in the analysis in Figure 9.18:

Mary by whom Peter is persuaded to sleep smiles. (9.53)

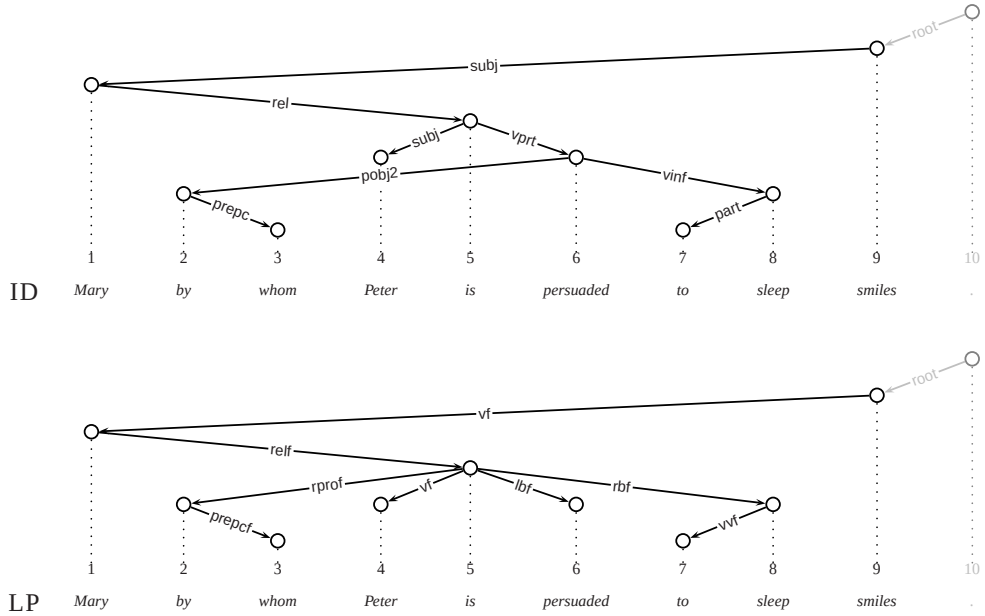


Figure 9.18.: ID/LP analysis of *Mary by whom Peter is persuaded to sleep smiles.*

9.5. Summary

In this chapter, we have modeled the syntax of a fragment of English. Our approach was based on TDG, where topological fields theory formed the basis of an elegant account of German word order. We have demonstrated that a similar analysis is also possible for English, where word order is less variable, but still far from trivial. As in TDG, we have modularized the dimensions of grammatical function and word order, which greatly simplified the description of syntax. In fact, phenomena such as topicalization and pied piping simply emerged from the intersective demands of the individual dimensions, and did not have to be explicitly specified. As we will see, the modularity of the grammar design will also prove beneficial for the specification of the syntax-semantics interface in chapter 12, where we will be able to exclusively concentrate on the ID dimension of grammatical functions, while not having to worry about word order at all.

10. Semantics

Turning to the semantics of natural language, we again adopt a very modular approach: we regard semantics not as a monolithic whole, but as modularized into three dimensions: *Predicate-Argument structure* (PA), *SCope structure* (SC), and *Information Structure* (IS). "Semantics" in the narrower sense, traditionally expressed using predicate logic or higher order logic (Montague 1974), is modeled by the PA and SC dimensions, where the PA dimension reflects the predicate-argument relations, and the SC dimension scopal relations. The mutual relation of the PA and SC dimensions is constrained by means of the PA/SC dimension. The IS dimension, represents theme/rheme and focus/background relationships, and thus corresponds to "semantics" in a broader sense, close to pragmatics.

The position of the semantic dimensions in the overall architecture of the grammar is displayed in Figure 10.1. The modularity of XDG allows us to formulate the account of semantics completely independently from syntax, which will significantly simplify the syntax-semantics interface in chapter 12.

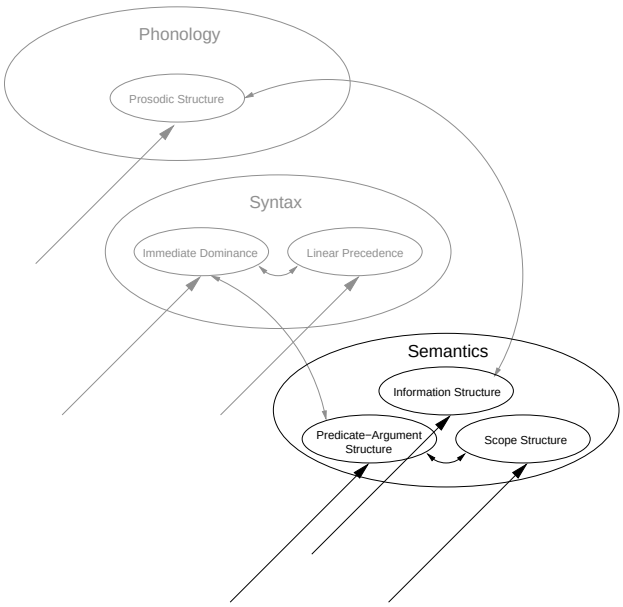


Figure 10.1.: Semantics in the overall architecture of the example grammar

10.1. Predicate-Argument Dimension

The PA dimension models predicate argument structure as a DAG called PA *DAG*, whose edges are labeled by *thematic roles* (Panenová 1974). We use a pragmatic, coarse-grained notion of thematic roles, whose only purpose is to distinguish multiple arguments of a node, and we do not make any claims towards their linguistic adequacy, which is problematic (Dowty 1989). In PA DAGs, all nodes not serving a semantic purpose are "deleted", i.e., collected by the root node with an edge labeled del. For example, we delete the prepositions of prepositional objects, since we consider them only as argument markers, and not as semantic predicates as e.g. Wechsler (1995). This is reflected in the PA DAG of the sentence below in Figure 10.2, where the preposition *to* is deleted:

Peter gives a book to Mary. (10.1)

In the PA DAG, *Peter* is the agent (edge label ag) of *gives*, *book* the patient (pat) and *Mary* the addressee (addr). *a* is the determiner (det) of *book*.

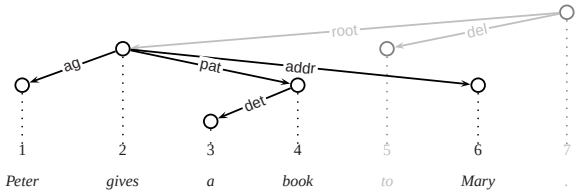


Figure 10.2.: PA DAG of the sentence *Peter gives a book to Mary.*

From another perspective, PA DAGs can be regarded as multisets of predicates and their arguments. For example, the PA DAG of Figure 10.2 can be regarded as the following multiset:

$$book(x), give(p,x,m) \tag{10.2}$$

where we regard the variable *x* as implicitly existentially quantified. The first argument of the predicate *give* is its agent, the second its patient, and the third its addressee.

As the PA dimension reflects only semantic but not syntactic considerations, contrary to the ID dimension, passive constructions are analyzed precisely as their active counterparts. An example is the PA analysis of the passive version (10.3) of (10.1) below in Figure 10.3, where again *Peter* is the agent, *book* the patient and *Mary* the addressee:

To Mary, a book is given by Peter. (10.3)

Contrary to the analyses of the ID dimension, PA analyses are DAGs and not trees, since we require multiple incoming edges per node e.g. for the modeling of control constructions. For example, consider the following sentence:

Peter persuades Mary to sleep. (10.4)

which we schematically analyze as following multiset to show that the argument *m* representing *Mary* is both an argument of the predicate *persuade* and of the predicate *sleep*:

$$persuade(p,m,sleep(m)) \tag{10.5}$$

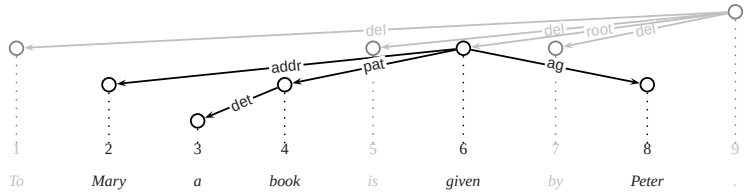


Figure 10.3.: PA DAG of the passive sentence *To Mary, a book is given by Peter.*

This is reflected in the PA DAG in Figure 10.4, where *Mary* has two incoming edges: one labeled pat from *persuades* and one labeled ag from *sleep*.

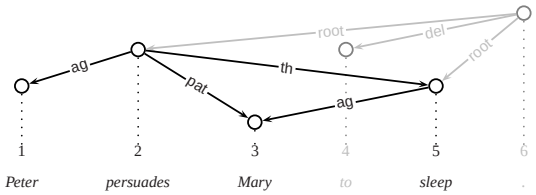


Figure 10.4.: PA DAG of *Peter persuades Mary to sleep.*

A second difference of PA to ID analyses is that the dependency relation between syntactic heads and their modifiers is reversed: on the PA dimension, modifiers take their syntactic heads as their dependents. This is reflected in the PA DAG shown in Figure 10.5 of the sentence below:

Peter loves a woman who often hums. (10.6)

where the adverb *often* takes the modified verb *hums* as its theme dependent (edge label thm for "theme of a modifier"). The PA DAG also shows that relative pronouns play a double role on the PA dimension:

1. As an argument of the finite verb heading the relative clause, e.g. *who* is the agent of *hums* in Figure 10.5.

2. As a modifier of their noun: *who* is connected to *woman* by an edge labeled agm (standing for "agent of a modifier") in Figure 10.5.

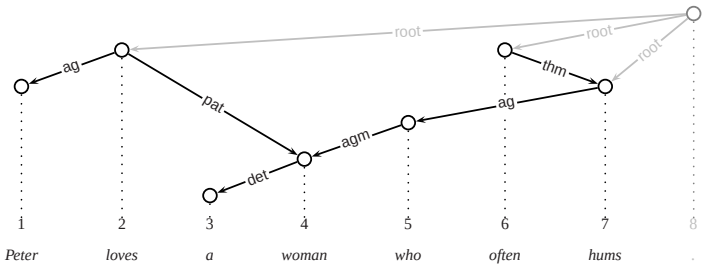


Figure 10.5.: PA DAG of *Peter loves a woman who often hums*

In the following, we call prepositional modifiers of nouns *prepositional adjectives*, and of verbs *prepositional adverbs*. They are modeled similarly: prepositional adjectives take their modified noun as a agm dependent and prepositional adverbs take their modified verb as a thm dependent. Both take their complement as a patm ("patient of a modifier") dependent, as illustrated in the analysis of the following sentence in Figure 10.6:

Every researcher of a company smiles with a woman. (10.7)

which models the following schematic multiset of predicates:

$$researcher(x), company(y), of(x,y), woman(z), with(smiles(x),z) \tag{10.8}$$

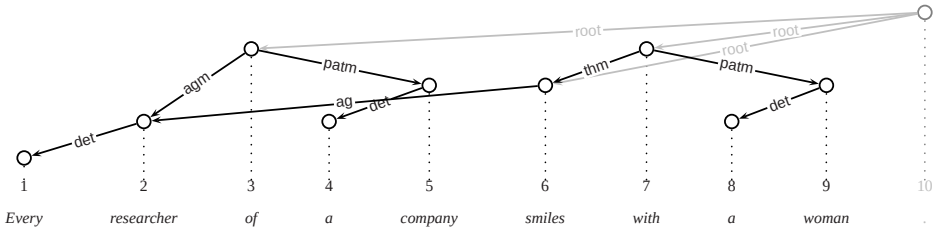


Figure 10.6.: PA DAG of *Every researcher of a company smiles with a woman.*

10.1.1. Types

Edge Labels. The type of edge labels on the PA dimension is defined below, and we give an overview of the edge labels and their corresponding thematic roles in Figure 10.7:

```
deftype "pa.label" {addr ag agm del det pat patm root th thm}
deflabeltype "pa.label"
```
(10.9)

The edge labels include the traditional thematic roles agent, patient and addressee, which we use to denominate nominal arguments of verbs. For verbal arguments, we use the role theme. `agm`, `patm` and `thm` denote agents, patients and themes of modifiers, and `det` is the edge label of determiners. `del` marks nodes without a semantic contribution as to be "deleted", and `root` marks predicates.

Attributes. The PA dimension defines the following lexical attributes:

```
defentrytype {in: valency("pa.label")
              out: valency("pa.label")
              lockDaughters: set("pa.label")}
```
(10.10)

where `in` and `out` represent valencies and `lockDaughters` is a set of PA edge labels.

10.1.2. Principles and Lexical Classes

Models. The models on the PA dimension are DAGs (cf. principle 1 in chapter 4):

```
useprinciple "principle.graph" { dims {D: pa} }
useprinciple "principle.dag" { dims {D: pa} }
```
(10.11)

edge label	thematic role
addr	addressee
ag	agent of a verb
agm	agent of a modifier
del	deleted node
det	determiner
pat	patient of a verb
patm	patient of a modifier
root	root
th	theme
thm	theme of a modifier

Figure 10.7.: PA edge labels and corresponding thematic roles

Valency. The PA dimension makes use of the *Valency principle* to constrain the incoming and outgoing edges of the nodes.

```
useprinciple "principle.valency" {
  dims {D: pa}
  args {In: _.D.entry.in
        Out: _.D.entry.out}}
```
(10.12)

The following four lexical classes constrain the incoming edges of nodes on the PA dimension:

1. Predicates are all main verbs, adverbs, adjectives and prepositional modifiers. They require an incoming edge labeled `root`:

```
defclass "pa_pred" {
  dim pa {in: {root!}}}
```
(10.13)

2. Words without a semantic contribution, i.e., auxiliary verbs[1], particles, complementizers and prepositional objects, require an incoming edge labeled `del`, i.e., they are "deleted":

```
defclass "pa_del" {
  dim pa {in: {del!}}}
```
(10.14)

3. Nouns can be the agent, patient or addressee of arbitrary many verbs, and can be the agent or patient of arbitrary many adjectives, prepositional adjectives or relative clauses:

```
defclass "pa_noun" {
  dim pa {in: {ag* pat* addr* agm* patm*}}}
```
(10.15)

4. Determiners require an incoming edge labeled `det`:[2]

```
defclass "pa_det" {
  dim pa {in: {det!}}}
```
(10.16)

[1] We can delete auxiliary verbs since our account does not cover tense, nor aspect for simplicity.

[2] Alternatively, we could delete determiners on the PA dimension. We have decided to keep them to simplify the interface to CLLS, cf. appendix E.

Turning to the out valencies of the words, root nodes can have arbitrary many predicate dependents (labeled `root`) and can collect arbitrary many deleted dependents (`del`):

```
defclass "pa_root" {
  dim pa {in: {}
          out: {root* del*}}}
```
(10.17)

Adverbs are predicates, can be modified by arbitrary many other adverbs or prepositional adverbs, and require a theme. Prepositional adverbs in addition require a patient:

```
defclass "pa_adv" {
  "pa_pred"
  dim pa {in: {thm*}
          out: {thm!}}}

defclass "pa_padv" {
  "pa_adv"
  dim pa {out: {patm!}}}
```
(10.18)

Similarly, adjectives are predicates and require an agent, and prepositional adjectives in addition also require a patient:

```
defclass "pa_adj" {
  "pa_pred"
  dim pa {out: {agm!}}}

defclass "pa_padj" {
  "pa_adj"
  dim pa {out: {patm!}}}
```
(10.19)

Common nouns require a determiner:

```
defclass "pa_cnoun" {
  "pa_noun"
  dim pa {out: {det!}}}
```
(10.20)

And finally, relative pronouns require an outgoing edge labeled `agm` to their modified noun:

```
defclass "pa_relpro" {
  "pa_noun"
  dim pa {out: {agm!}}}
```
(10.21)

Locking. In control constructions, either the agent (in case of subject control) or the patient (object control) of the control verb is simultaneously the agent of at least one subordinate verb. For example, in Figure 10.4 above, the patient of *persuade* is also the agent of the subordinate verb *sleep*.

However, the subordinate verb cannot know *which* of the dependents of the control verb it may take. As an example, consider the wrong analysis of sentence below in Figure 10.8, where the agent of *sleep* is *Peter*, not *Mary*.

Peter tries to persuade Mary to sleep. (10.22)

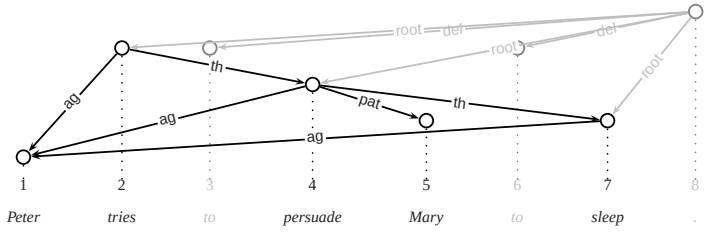


Figure 10.8.: Wrong PA DAG of *Peter tries to persuade Mary to sleep.*

To rule out such analyses, we must ensure that for object control verbs, only the patient may simultaneously be a dependent of a subordinate verb, but not the agent or the addressee, and similarly for subject control. All nominal arguments (agents, patients and addressee) of the verbs may however be a dependent of superordinate verbs reachable via an edge labeled `th`, and they may be a dependent of a modifier (e.g. an adjective or a relative clause).

We realize this constraint using the *LockingDaughters principle*, which is defined on the dimensions d_1, d_2 and d_3, and has the following declarative semantics: for all nodes v, the dependents v' reachable on d_1 via an edge label l in the lexically specified set *lockDaughters* are "locked", i.e., on d_2, they cannot be a dependent of any node except:

1. v
2. those nodes above v on d_1 reachable via edge labeled l', where l' is in *exceptAbove*
3. those mothers of v' on d_2 which enter v via an edge labeled l', where l' is in *key*

Principle 18 (LockingDaughters)**.**

$$\begin{aligned}
&lockingDaughters_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\
&v \xrightarrow{l}_{d_1} v' \wedge l \in (d_3\ v).lex.lockDaughters \Rightarrow \forall v'' : v'' \rightarrow_{d_2} v' \Rightarrow \\
&v'' \doteq v \quad \vee \\
&(\exists l' \in (d_3\ v).lex.exceptAbove \wedge v'' \rightarrow^*_{d_1} \xrightarrow{l'}_{d_1} v) \quad \vee \\
&(\exists l' \in (d_3\ v).lex.key \wedge v'' \xrightarrow{l'}_{d_2} v')
\end{aligned} \tag{10.23}$$

We apply the principle as follows:

```
useprinciple "principle.lockingDaughters" {
  dims {D1: pa
        D2: pa
        D3: pa}
  args {LockDaughters: _.D3.entry.lockDaughters
        ExceptAbove: {th}
        Key: {agm patm}}}
```
(10.24)

where we use the `ExceptAbove` argument to allow the nominal arguments to be simultaneously arguments of superordinate verbs reachable via an edge labeled `th`. With the `Key` argument, we allow the nominal arguments to also be arguments of modifiers.

We instantiate the `lockDaughters` attribute in the lexical classes for subject and object control verbs below:

```
defclass "pa_subjcr" {
  dim pa {lockDaughters: {pat addr}}}

defclass "pa_objcr" {
  dim pa {lockDaughters: {ag addr}}}
```
(10.25)

`"pa_subjcr"` (for "subject control/raising") locks all nominal complements except the agent, i.e., patient and addressee, and `"pa_objcr"` ("object control/raising") all nominal complements except the patient, i.e., agent and addressee. Now we can exclude the wrong analysis shown in Figure 10.8: the object control verb *persuade* only allows its patient *Mary* to become the dependent of a subordinate verb, and locks its agent *Peter*. As a result, only *Mary* can become the agent of the subordinate verb *sleep*, but not *Peter*.

The LockingDaughters principle is not only useful for control verbs, but also for "normal" verbs, e.g. intransitive or transitive verbs. If the nominal arguments are not locked, they can e.g. be "taken over" by verbs inside a relative clause, as in the wrong analysis of the sentence below in Figure 10.9:

Mary sees a woman who tries to sleep. (10.26)

where *Mary*, the agent of the transitive verb *loves*, is incorrectly simultaneously the agent of the verb *sleep*. We rule out such analyses with the lexical class `"pa_nocr"` ("no control/raising"), which locks all nominal arguments:

```
defclass "pa_nocr" {
  dim pa {lockDaughters: {ag pat addr}}}
```
(10.27)

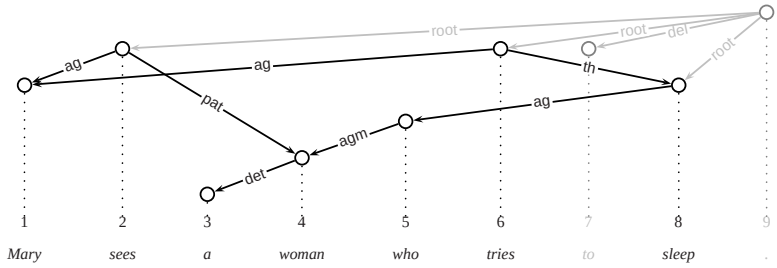


Figure 10.9.: Wrong PA DAG of *Mary sees a woman who tries to sleep.*

10.2. Scope Dimension

Turning to the dimension modeling scope, we begin with the example sentence below, which is ambiguous between the reading where every man loves another woman, and the reading where the same woman is loved by every man:

Every man loves a woman. (10.28)

The two readings are shown in predicate logic in (10.29) and (10.30). In the former, it is the universal quantifier which takes wide scope (weak reading), and in the latter, the existential quantifier (strong reading):

$$\forall x : man(x) \Rightarrow \exists y : woman(y) \wedge love(x,y) \tag{10.29}$$

$$\exists y : woman(y) \wedge \forall x : man(x) \Rightarrow love(x,y) \tag{10.30}$$

On the PA dimension, we have modeled the predicate-argument relations of the semantic representation, which are unambiguous and can be represented as the following multiset:

$$man(x), woman(y), love(x,y) \tag{10.31}$$

Complementary to the PA dimension, the *SCope structure* (SC) dimension is not concerned with predicate-argument structure, but solely with scopal relations. An SC analysis is an unordered tree called SC *tree* whose edges are labeled by scopal relationships. Figure 10.10 shows an SC tree of the weak reading (10.29), where *man* has the quantifier *every* (edge label q) and *woman* in its scope (edge label s), and *woman* in turn has the quantifier *a* and *loves* in its scope. Figure 10.11 shows an SC tree of the strong reading, where the existentially quantified *woman* takes wide scope.

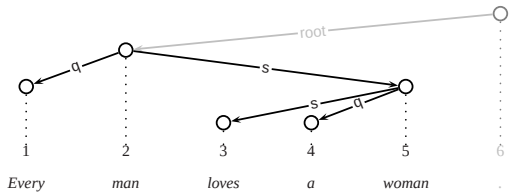


Figure 10.10.: SC tree of *Every man loves a woman.* (weak reading)

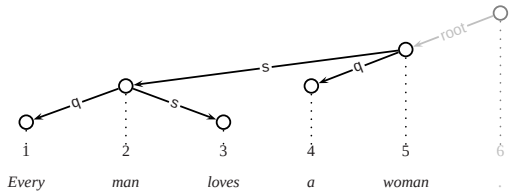


Figure 10.11.: SC tree of *Every man loves a woman.* (strong reading)

For illustration, we represent the two SC analyses schematically as follows, omitting the predicate-argument relations of (10.29) and (10.30):

$$\forall : man \Rightarrow \exists : woman \wedge love \tag{10.32}$$

$$\exists : woman \wedge \forall : man \Rightarrow love \tag{10.33}$$

As another example, adjectives on the SC dimension always end up in the restriction (edge label r) of the noun they modify, as in the analysis of the sentence below in Figure 10.12, where the adjectives *nice* and *little* end up in the restriction of the noun *product*:

Every nice little product hums. (10.34)

Schematically, Figure 10.12 can be represented as follows:

$$\forall : nice \wedge little \wedge product \Rightarrow hum \tag{10.35}$$

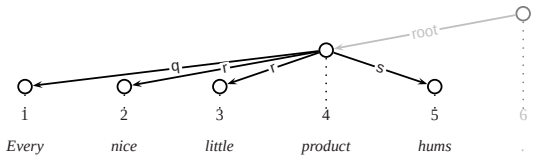


Figure 10.12.: SC tree of *Every nice little product hums.*

Adverbs and verbs with verbal complements take scope, i.e., they require an s dependent. For example, consider the SC trees in Figure 10.13 and Figure 10.14, which represent the two readings of the following sentence:

Every man seems to laugh. (10.36)

which we schematically represent below:

$$\forall : man \Rightarrow seem(laugh) \tag{10.37}$$

$$seem(\forall : man \Rightarrow laugh) \tag{10.38}$$

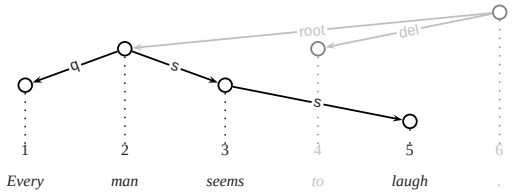


Figure 10.13.: SC tree of *Every man seems to laugh.* (reading where *every man* takes wide scope)

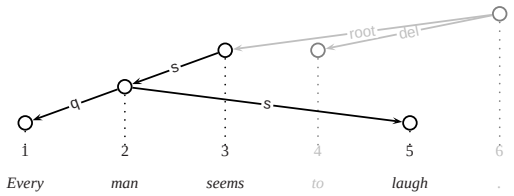


Figure 10.14.: SC tree of *Every man seems to laugh.* (reading where *seems* takes wide scope)

10.2.1. Types

Edge Labels. The type of edge labels on the SC dimension are defined as follows:

```
deftype "sc.label" {del q r root s}
deflabeltype "sc.label"
```
(10.39)

where `q` is the label for the quantifier of a common noun, `r` for its restriction, and `s` for the scope of a node. As on the PA dimension, `del` marks deleted nodes. `root` is the incoming edge label of the node taking the widest scope. We give an overview of the edge labels and their corresponding scopal relations in Figure 10.15.

edge label	scopal relation
`del`	deleted node
`q`	quantifier
`root`	root
`r`	restriction
`s`	scope

Figure 10.15.: SC edge labels and corresponding scopal relations

Attributes. The lexical attributes of the SC dimension comprise the valency attributes `in` and `out`:

```
defentrytype {in: valency("sc.label")
              out: valency("sc.label")}
```
(10.40)

10.2.2. Principles and Lexical Classes

Models. The models of the SC dimension must be trees:

```
useprinciple "principle.graph" { dims {D: sc} }
useprinciple "principle.tree" { dims {D: sc} }
```
(10.41)

Scopal Valency. Using the *Valency principle*, we constrain the incoming and outgoing edges of the nodes on the SC dimension, which we call their *scopal valency*:

```
useprinciple "principle.valency" {
  dims {D: sc}
  args {In: _.D.entry.in
        Out: _.D.entry.out}}
```
(10.42)

We define three lexical classes for constraining the incoming edges of the nodes:

1. Words with semantic content (main verbs, adverbs, adjectives, prepositional modifiers and nouns) can either end up in the restriction or scope of another node, or they take widest scope:

```
defclass "sc_cont" {
  dim sc {in: {r? s? root?}}}
```
(10.43)

2. Word without semantic content (auxiliary verbs, particles, complementizers, prepositional objects) are deleted:

```
defclass "sc_nocont" {
  dim sc {in: {del!}}}
```
(10.44)

3. Determiners are a special case: even though they are not deleted, they do not inherit from the class for words with semantic content `"sc_cont"`, since they cannot end up in the restriction/scope of another word, but only as a quantifier of a common noun with incoming edge label `q`:[3]

```
defclass "sc_det" {
  dim sc {in: {q?}}}
```
(10.45)

The next classes constrain the outgoing edges of the nodes. Root nodes require one outgoing edge labeled `root` for the node taking widest scope and can collect arbitrary many deleted nodes:

```
defclass "sc_root" {
  dim sc {in: {}
          out: {root! del*}}}
```
(10.46)

The class `"sc_sc"` is used words taking scope: adverbs, prepositional adverbs, verbs with verbal complements and nouns:

```
defclass "sc_sc" {
  dim sc {out: {s!}}}
```
(10.47)

Nouns not only have semantic content and take scope, but also license arbitrary many outgoing edges labeled `r` into their restriction:

```
defclass "sc_noun" {
  "sc_cont"
  "sc_sc"
  dim sc {out: {r*}}}
```
(10.48)

In addition, common nouns require an outgoing edge labeled `q` for their quantifier:

```
defclass "sc_cnoun" {
  "sc_noun"
  dim sc {out: {q!}}}
```
(10.49)

10.3. PA/SC Dimension

The interface between the PA and SC dimensions is specified by the PA/SC dimension, whose models are graphs without edges. Basically, the PA/SC dimension states two constraints:

1. the nominal arguments of verbs on PA take scope over the verbs on SC
2. the mothers of verbs on PA take scope over the verbs on SC

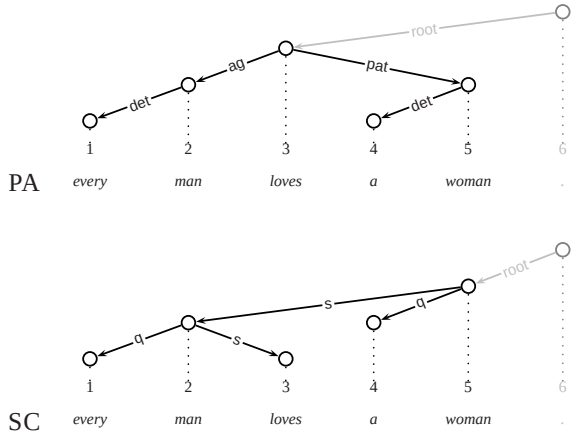


Figure 10.16.: PA/SC analysis of *Every man loves a woman.*

As an example for the former, Figure 10.16 shows an example PA/SC analysis of the sentence (10.28), where correctly, both nominal arguments *man* and *woman* of *loves* on PA take scope over it on SC, i.e., both dominate *loves* on SC.

Figure 10.17 shows an analysis of the example sentence below, where the mothers *seems* and *today* of *laugh* on PA both take scope over it on SC:

Every man seems to laugh today. (10.50)

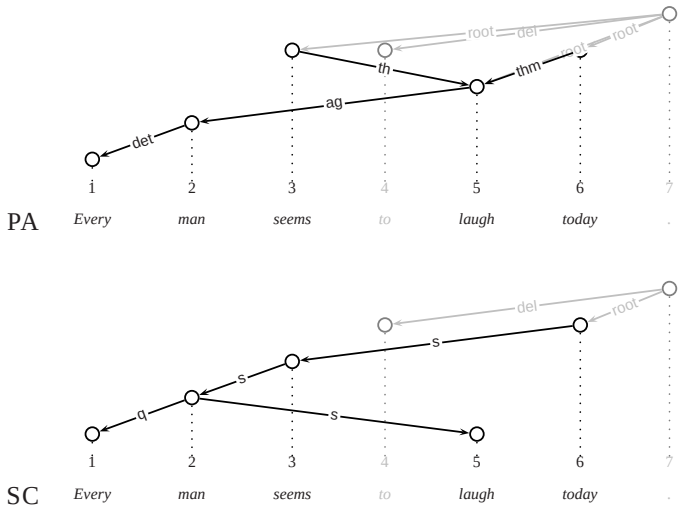


Figure 10.17.: PA/SC analysis of *Every man seems to laugh today.*

[3] As on the PA dimension, we could also choose to delete determiners on the SC dimension, but keep them to simplify the construction of a CLLS semantics, cf. appendix E.

10.3.1. Types

Attributes. The lexical attributes on the PA/SC dimension include the three vectors used to map PA edge labels to sets of SC edge labels:

```
defentrytype {linkAboveEnd: vec("pa.label" set("sc.label"))
              linkBelowStart: vec("pa.label" set("sc.label"))
              linkDaughterEnd: vec("pa.label" set("sc.label"))}
```
(10.51)

10.3.2. Principles and Lexical Classes

LinkingAboveEnd. We use the *LinkingAboveEnd principle* to state that the nominal arguments of nodes (on PA) take scope over them. The principle has the declarative semantics that if for an edge from v to v' labeled l on d_1, the value of *linkAboveEnd* for v and l on d_3 is non-empty, then for at least one edge label l' in this set, v' must be above v on d_2, and the path from v to v' must end with an edge labeled l'.

Principle 19 (LinkingAboveEnd)**.**

$$\begin{aligned} &\mathit{linkingAboveEnd}_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} v' \ \wedge \ (d_3\ v).\mathit{lex}.\mathit{linkAboveEnd}.l \neq \emptyset \Rightarrow \\ &\exists l' : l' \in (d_3\ v).\mathit{lex}.\mathit{linkAboveEnd}.l \ \wedge \ v' \xrightarrow{l'}_{d_2} \rightarrow^*_{d_2} v \end{aligned} \tag{10.52}$$

We apply the principle as follows:

```
useprinciple "principle.linkingAboveEnd" {
  dims {D1: pa
        D2: sc
        D3: pasc}
  args {End: ^.D3.entry.linkAboveEnd}}
```
(10.53)

where the `linkAboveEnd` attribute is used in the lexical class for main verbs defined below, where all possible nominal arguments (`ag`, `pat` and `addr`) on PA are constrained to `s` dominate their verbs on SC:[4]

```
defclass "pasc_main" {
  dim pasc {linkAboveEnd: {ag: {s}
                           pat: {s}
                           addr: {s}}
            linkBelowStart: {th: {s}}}}
```
(10.54)

The attribute `linkAboveEnd` is also used in the lexical class `"pasc_modn"` for "modifiers of nouns" (relative pronouns, adjectives and prepositional adjectives), where the modified noun is constrained to `r` dominate its modifiers:

```
defclass "pasc_modn" {
  dim pasc {linkAboveEnd: {agm: {r}}}}
```
(10.55)

As an example, consider the underspecified PA/SC analysis of the sentence below in Figure 10.18:

A nice woman often sleeps. (10.56)

where the noun modified by the adjective *nice* on PA, i.e., *woman*, `r` dominates *nice* on SC.

[4] The meaning of the attribute `linkBelowStart` is given shortly.

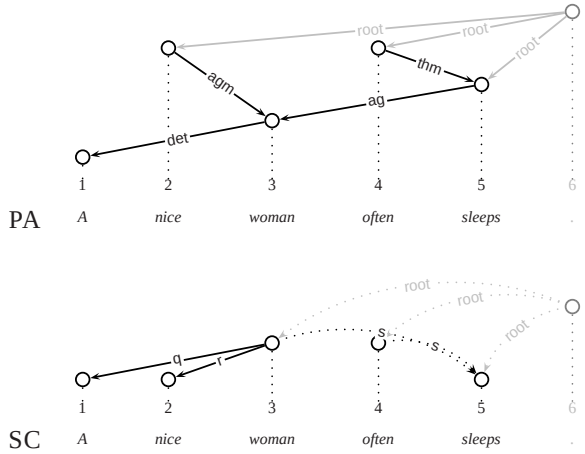


Figure 10.18.: Underspecified PA/SC analysis for *A nice woman often sleeps.*

Prepositional adjectives and prepositional adverbs also have a second argument in addition to the agent of a modifier (`agm`) of adjectives and the theme of a modifier (`thm`) of adverbs: the patient of a modifier `patm`, which is also a nominal argument. Using the LinkingAboveEnd principle, we constrain it to `s` dominate its preposition:

```
defclass "pasc_pmod" {
  dim pasc {linkAboveEnd: {patm: {s}}}}
```
(10.57)

LinkingBelowStart. To state that nodes always take scope over their verbal arguments (on PA), we make use of the *LinkingBelowStart principle*. The principle is symmetric to the LinkingAboveEnd principle: the only differences are that the daughter v' of v on d_1 must be below v on d_2, and that the path from v to v' on d_2 must start instead of end with an edge labeled l'.

Principle 20 (LinkingBelowStart)**.**

$$\begin{aligned} &\mathit{linkingBelowStart}_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} v' \wedge (d_3\ v).\mathit{lex}.\mathit{linkBelowStart}.l \neq \emptyset \Rightarrow \\ &\exists l' : l' \in (d_3\ v).\mathit{lex}.\mathit{linkBelowStart}.l \wedge v \xrightarrow{l'}_{d_2} \rightarrow^*_{d_2} v' \end{aligned} \tag{10.58}$$

We apply the principle using the lexical attribute `linkBelowStart`:

```
useprinciple "principle.linkingBelowStart" {
  dims {D1: pa
        D2: sc
        D3: pasc}
  args {Start: ^.D3.entry.linkBelowStart}}
```
(10.59)

We apply this attribute in the lexical class for modifiers of verbs below, which states that on SC, each node must `s` dominate its theme:

```
defclass "pasc_modv" {
  dim pasc {linkBelowStart: {thm: {s}}}}
```
(10.60)

The class is applied for adverbs and prepositional adverbs. As an example, reconsider Figure 10.18, where the verbal modifier *often* correctly `s` dominates its theme *sleeps*.

LinkingDaughterEnd. The third principle applied on the PA/SC dimension is the *LinkingDaughterEnd principle*, used with lexical attribute `linkDaughterEnd`:

```
useprinciple "principle.linkingDaughterEnd" {
  dims {D1: pa
        D2: sc
        D3: pasc}
  args {End: ^.D3.entry.linkDaughterEnd}}
```
(10.61)

We use the principle only to ensure that the quantifier of a common noun on SC corresponds to its determiner on PA:

```
defclass "pasc_cnoun" {
  dim pasc {linkDaughterEnd: {det: {q}}}}
```
(10.62)

10.4. Information Structure Dimension

Information structure is not concerned with the truth conditions of a sentence, but rather with its felicity in the discourse. This is of crucial importance for e.g. *Content-To-Speech systems* (*CTS*), where IS improves the quality of the speech output (Prevost & Steedman 1994), and *Machine Translation* (*MT*), where IS improves target word order, especially for free word order languages (Stys & Zemke 1995).

We adopt the approach of Steedman (2000*a*), where information structure divides each utterance into two parts: *theme*[5] and *rheme*. The theme relates the utterance to the prior discourse, and the rheme adds or modifies information about the theme. Steedman (2000*a*) further differentiates themes and rhemes into *focus* and *background*: the focus is the *accented* word of a theme or rheme, whereas the remaining words constitute the background.

As an example, consider the following sentence:

Peter_L+H loves_LH% Mary_H*_LL%* (10.63)

where we prosodically annotate[6] the words according to (Pierrehumbert 1980) and (Steedman 2000*a*) by:

1. their *pitch accents*
2. the *boundary tones* following them

In the example, *Peter_L+H** has the pitch accent *L+H**, *loves_LH%* is followed by the boundary tone *LH%*, and *Mary_H*_LL%* has the pitch accent *H** and the directly following boundary tone *LL%*. The pitch accent *L+H** indicates the focus of a theme, and *H** the focus of a rheme. The boundary tone *LH%* marks the end of a theme, and *LL%* the end of a rheme. As a result, the theme of the sentence is *Peter loves* and the rheme *Mary*, and within the theme, *Peter* is the focus and *loves* the background. This information structure is felicitous in a context where the question is *Who does Peter love?* where the theme *Peter loves* is already mentioned

[5] This "theme" is different from the thematic role called "theme" on the PA dimension.
[6] For our purposes, it suffices to know that here, *L* stands for "low" and *H* for "high" accent/tone.

in the context, and the rheme *Mary* is not. It is however not felicitous in the context *By whom is Mary loved?*, where *Mary* is already mentioned.

On the IS dimension, we model this structure using ordered projective trees whose edge labels reflect the theme/rheme and focus/background distinctions. Following (Jackendoff 2002), we position the IS dimension within the semantics in the overall architecture of our grammar (cf. Figure 10.1). We call an IS analysis IS *tree*. Figure 10.19 shows an example IS tree of sentence (10.63). Here, the additional node corresponding to the full stop has outgoing edges into the focus of the theme (edge label th) *Peter* and into the focus of the rheme (rh) *Mary*. *Peter* in turn has an outgoing edge into its background (bg) *loves*. Hence, *Peter loves* is the theme of the sentence, and *Mary* the rheme. We call the theme and rheme subtrees *information structural constituents* (IS *constituents*). For example, *Peter* and *Mary* constitute the IS constituent corresponding to the theme of the sentence, and *Mary* the IS constituent corresponding to the rheme.

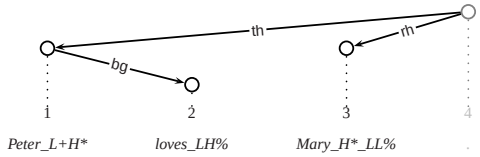


Figure 10.19.: IS tree of *Peter_L+H* loves_LH% Mary_H*_LL%*

Figure 10.20 shows another example IS tree. Here, *Mary* is again the rheme and *Peter loves* the theme. However, contrary to the previous example, the theme is an *unmarked theme*, not marked by a pitch accent, and thus not having focus. This is reflected in the IS tree by each word in the unmarked theme having an incoming edge labeled umth.

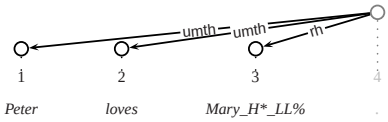


Figure 10.20.: IS tree of *Peter loves Mary_H*_LL%*.

10.4.1. Types

Labels. The type of edge labels on the IS dimension contains bg for background, rh for rheme, th for theme, and umth for unmarked theme:

```
deftype "is.label" {bg rh th umth}
```
(10.64)

We give an overview of the edge labels and their corresponding information structural categories in Figure 10.21.

Attributes. The lexical attributes of the IS dimension include the valencies in and out:

```
defentrytype {in: valency("is.label")
              out: valency("is.label")}
```
(10.65)

edge label	information structural category
bg	background
rh	rheme
th	theme
umth	unmarked theme

Figure 10.21.: IS edge labels and corresponding information structural categories

10.4.2. Principles and Lexical Classes

Models. The models of the IS dimension are ordered and projective trees, but with no particular order on the outgoing edges of the nodes:

```
useprinciple "principle.graph" { dims {D: is} }
useprinciple "principle.tree" { dims {D: is} }
useprinciple "principle.projectivity" { dims {D: is} }
useprinciple "principle.order" {
  dims {D: is}
  args {Order: <>}}
```
(10.66)

Information Structural Valency. We use the *Valency principle* to constrain the incoming and outgoing edges, which we call *information structural valency*:

```
useprinciple "principle.valency" {
  dims {D: is}
  args {In: _.D.entry.in
        Out: _.D.entry.out}}
```
(10.67)

For roots, we define the lexical class `"is_root"`:

```
defclass "is_root" {
  dim is {in: {}
          out: ({th* rh+}|{umth* rh+})}}
```
(10.68)

stating the following two constraints:

1. Each sentence must have at least one rheme.
2. The rheme can be accompanied by arbitrary many themes or unmarked themes, but not by both, i.e. an analysis cannot contain themes and unmarked themes at the same time.

The focus of a theme can only have an incoming edge labeled `th` and licenses arbitrary many dependents in its background:

```
defclass "is_tf" {
  dim is {in: {th?}
          out: {bg*}}}
```
(10.69)

The focus of a rheme can only have an incoming edge labeled `rh` and licenses arbitrary many `bg` dependents:

```
defclass "is_rf" {
  dim is {in: {rh?}
          out: {bg*}}}
```
(10.70)

Non-foci can either become background of the focus, or part of an unmarked theme:

```
defclass "is_nf" {
  dim is {in: {bg? umth?}}}
```
(10.71)

10.5. Emerging Phenomena

The separation of predicate-argument structure and scope structure allows us, in combination with the XDK constraint parser, to selectively postpone the enumeration of readings which differ only in their scope structure, which brings us *scope underspecification* for free, without any further stipulation.

10.5.1. Scope Underspecification

As explained in section 8.3.4 of chapter 8, the XDK constraint parser is able to selectively postpone the enumeration of readings on any of the the individual dimensions. If we decide to enumerate the readings only on the PA dimension, but not on the SC dimension, we get *scope underspecification* for free: a scopally underspecified semantic analysis is then simply a PA/SC analysis consisting of:

- a total PA analysis
- a partial SC analysis

where the partial SC analysis includes edges already determined by the constraint parser and additional information, e.g. stating which nodes are already known to dominate which other nodes.

As an example, Figure 10.22 shows an underspecified PA/SC analysis of (10.28). The partial SC analysis includes the edges labeled q from *man* to *every* and from *woman* to *a* which are already determined, and the information that *man* and *woman* both s dominate[7] *loves*, which is indicated by curved dotted edges. In appendix E, we show how to make use of partial SC analyses in an interface to the Constraint Language for Lambda Structures (CLLS).

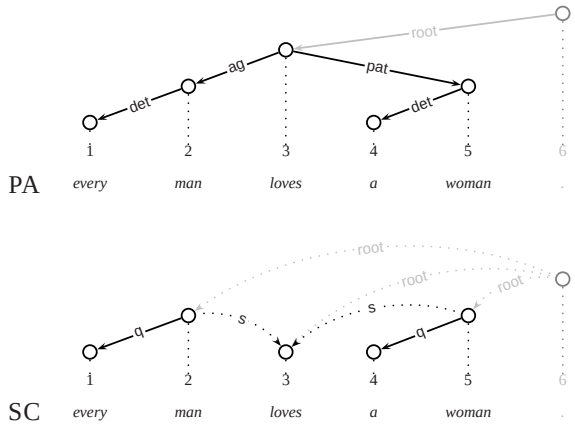


Figure 10.22.: Underspecified PA/SC analysis of *Every man loves a woman.*

[7] A node *v l* dominates another node *v′* if there is a path from *v* to *v′* starting with an edge labeled *l*.

10.6. Summary

In this chapter, we have modeled natural language semantics using the XDK. Inspired by the parallel grammar architecture of Sadock (1991) and Jackendoff (2002), we took a modular view on semantics, and distinguished the dimensions of predicate-argument structure (PA), scope structure (SC) and information structure (IS). The PA/SC dimension constrained the relation between the PA and SC dimensions. Our approach is one of the first to model "deep semantics" (including not only predicate-argument structure but also scope structure) in a dependency-based grammar formalism. In combination with the XDK constraint parser, our approach gave us scope underspecification for free, without further stipulation.

11. Phonology

In this chapter, we add phonology to our example grammar, in the form of the *Prosodic Structure* (PS) dimension. Dealing only with prosody, we cover only a very small subset of phonology, leave out many other aspects, e.g. rhythm, stress and syllabic structure. Our account of prosody follows (Pierrehumbert 1980) and (Steedman 2000*a*), and will lead, together with our model of information structure in section 10.4 of chapter 10, to a modular version of the prosodic account of information structure introduced in (Steedman 2000*a*). We display the position of the PS dimension in the overall architecture of the example grammar in Figure 11.1.

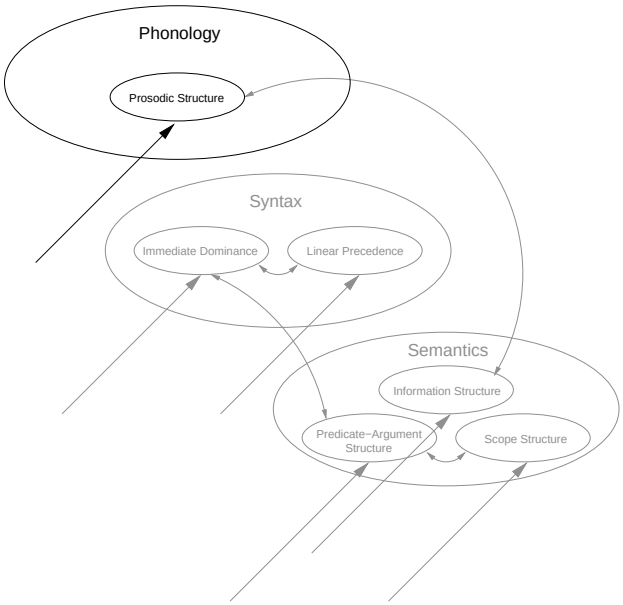


Figure 11.1.: Phonetics in the overall architecture of the example grammar

11.1. Prosodic Structure Dimension

We regard prosody as dividing sentences into substrings that we call *prosodic constituents* or PS *constituents* for short. PS constituents are delimited by boundary tones: as an example, consider the following prosodically annotated example sentence:

Peter_L+H loves_LH% Mary_H*_LL%.* (11.1)

where *Peter* carries the pitch accent *L+H**, *loves* is followed by the boundary tone *LH%*, and *Mary* both carries the pitch accent *H** and is followed by the boundary tone *LL%*. The

boundary tone following *loves* delimits the PS constituent *Peter loves* and the boundary tone following *Mary* the PS constituent *Mary*.

We model this structure on the *Prosodic Structure* (PS) dimension, whose models are ordered and projective trees called PS *trees*. In PS trees, all words followed by boundary tones are connected to the additional root node corresponding to the end-of-sentence marker, and the remaining words are connected to the next word followed by a boundary tone to the right. The words followed by boundary tones and their dependents constitute the PS constituents of the sentence.

Figure 11.2 shows an example PS tree of (11.1), where *loves*, followed by the boundary tone *LH%*, is connected to the additional root node by an edge labeled bt1 standing for "boundary tone 1", and *Mary*, carrying pitch accent *H** and followed by the boundary tone *LL%*, by an edge labeled pa2bt2 ("pitch accent 2 and boundary tone 2"). *Peter*, carrying the pitch accent *L+H**, is connected to the next word followed by a boundary tone (*loves*) by an edge labeled pa1 ("pitch accent 1"). The resulting PS constituents *Peter loves* and *Mary* correspond to the subtrees of *loves* and *Mary*.

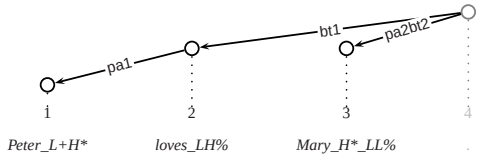


Figure 11.2.: PS tree of *Peter_L+H* loves_LH% Mary_H*_LL%*.

As another example, Figure 11.3 shows a PS tree for the sentence below, which contains only one prosodic constituent, i.e., *Peter loves Mary*. *Mary* has incoming edge label pa2bt2 standing for "pitch accent 2 and boundary tone 2". The other words are unaccented and connected to the next word followed by a boundary tone, *Mary*, by edges labeled ua.

*Peter loves Mary_H*_LL%.* (11.2)

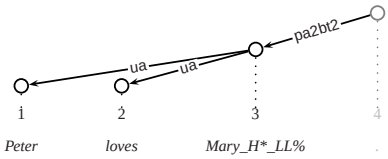


Figure 11.3.: PS tree of *Peter loves Mary_H*_LL%*.

11.1.1. Types

Edge Labels. The type of edge labels on the PS dimension is defined as:

```
deftype "ps.label" {bt1 bt2 pa1 pa1bt1 pa2 pa2bt2 ua}
deflabeltype "ps.label"
```
(11.3)

and includes:

1. `bt1`, `bt2` for the two boundary tones *LH%* and *LL%* covered by the grammar
2. `pa1`, `pa2` for the two *pitch accents* *L+H** and *H**
3. `pa1bt1`, for the combination of `pa1` and `bt1` and `pa2bt2` for the combination of `pa2` and `bt2`
4. `ua` for *unaccented*

We give an overview of the edge labels and the corresponding prosodic categories in Figure 11.4.

edge label	prosodic category
`bt1`	followed by boundary tone 1
`bt2`	followed by boundary tone 2
`pa1`	carrying pitch accent 1
`pa1bt1`	carrying pitch accent 1 and followed by boundary tone 1
`pa2`	carrying pitch accent 2
`pa2bt2`	carrying pitch accent 2 and followed by boundary tone 2
`ua`	unaccented

Figure 11.4.: `PS` edge labels and corresponding prosodic categories

Attributes. The lexical attributes include the valencies `in` and `out` and the set `order`, representing a strict partial order on the outgoing edges and the special anchor label `"^"`:

```
deftype "ps.label1" "ps.label" | {"^"}
defentrytype {in: valency("ps.label")
              out: valency("ps.label")
              order: set(tuple("ps.label1" "ps.label1"))}
```
(11.4)

11.1.2. Principles and Lexical Classes

Models. The models of the `PS` dimension are projective trees:

```
useprinciple "principle.graph" { dims {D: ps} }
useprinciple "principle.tree" { dims {D: ps} }
useprinciple "principle.projectivity" { dims {D: ps} }
```
(11.5)

Prosodic Valency and Order. We use the *Valency principle* to constrain the incoming and outgoing edges of the nodes, which we call *prosodic valency*, and we use the *Order principle* to order the dependents of boundary tones to their left:

```
useprinciple "principle.valency" {
  dims {D: ps}
  args {In: _.D.entry.in
        Out: _.D.entry.out}}
useprinciple "principle.order" {
  dims {D: ps}
  args {Order: _.D.entry.order}}
```
(11.6)

The Valency principle is applied using the lexical attributes `in` and `out`, and the Order principle using the lexical attribute `order`.

The additional root node (corresponding to the end-of-sentence marker) is characterized by the following lexical class:

```
defclass "ps_root" {
  dim ps {in: {}
          out: {bt1* bt2* pa1bt1* pa2bt2*}
          order: {[bt1 "^"] [bt2 "^"] [pa1bt1 "^"] [pa2bt2 "^"]}}}
```
(11.7)

That is, it does not license any incoming edge, and arbitrary many edges to nodes which correspond to words followed by boundary tones (either labeled `bt1`, `bt2`, `pa1bt1` or `pa2bt2`). By the `order` attribute, the root is constrained to follow its dependents. The order among its dependents is not constrained.

Words followed by any boundary tone (variable `BT`) license at most one incoming edge labeled by `BT`, arbitrary many outgoing edges to words carrying pitch accent `PA` and arbitrary many outgoing edges to unaccented words. It must precede its dependents:

```
defclass "ps_bt" BT PA {
  dim ps {in: {BT?}
          out: {PA* ua*}
          order: {[PA "^"] [ua "^"]}}}
```
(11.8)

Thus, words followed by a boundary tone only license outgoing edges to either unaccented words or to words carrying a specific pitch accent, i.e., prosodic constituents may only include words which carry appropriate pitch accents. For example, words followed by boundary tone 1 only license outgoing edges to unaccented words or words carrying pitch accent 1, and similarly for boundary tone 2 and for combinations of boundary tones and pitch accents:

```
defclass "ps_bt1" { "ps_bt" {BT: bt1 PA: pa1} }

defclass "ps_bt2" { "ps_bt" {BT: bt2 PA: pa2} }

defclass "ps_pa1bt1" { "ps_bt" {BT: pa1bt1 PA: pa1} }

defclass "ps_pa2bt2" { "ps_bt" {BT: pa2bt2 PA: pa2} }
```
(11.9)

These lexical classes exclude sentences such as the one below (analyzed in Figure 11.5), where the prosodic constituent delimited by *loves*, followed by boundary tone 1 (*L+H%*), includes *Peter* carrying the unappropriate pitch accent 2 (*H**):

Peter_H loves_L+H% Mary_H*_LL%.* (11.10)

Words carrying any pitch accent `PA` only license an incoming edge labeled `PA`, and no outgoing edges:

```
defclass "ps_pa" PA {
  dim ps {in: {PA?}}}
```
(11.11)

We instantiate this lexical class as follows for the two pitch accents covered by the grammar:

```
defclass "ps_pa1" { "ps_pa" {PA: pa1} }

defclass "ps_pa2" { "ps_pa" {PA: pa2} }
```
(11.12)

Unaccented words only license an incoming edge labeled `ua` and no outgoing edges:

```
defclass "ps_ua" {
  dim ps {in: {ua?}}}
```
(11.13)

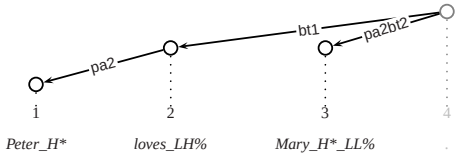


Figure 11.5.: PS tree of the ill-formed sentence *Peter_H* loves_L+H% Mary_H*_LL%*.

11.2. Summary

We have developed a simplified model of prosody following the account of (Pierrehumbert 1980) and (Steedman 2000*a*). Prosody will play an important role in the phonology-semantics interface developed in the next chapter, which realizes the prosodic account of information structure introduced in (Steedman 2000*a*).

12. Interfaces

This chapter introduces the *syntax-semantics interface* of the example grammar, realized by the ID/PA dimension, and the *phonology-semantics interface*, which is realized by the PS/IS dimension. The ID/PA dimension characterizes the relation between the ID dimension of *grammatical functions* and the PA dimension of *thematic roles* by constraining how semantic arguments must be realized syntactically. The PS/IS dimension completes our version of the prosodic account of information structure introduced in (Steedman 2000*a*) by constraining the relation between the PS and IS dimensions. We display the position of the interfaces in the overall architecture of the example grammar in Figure 12.1.

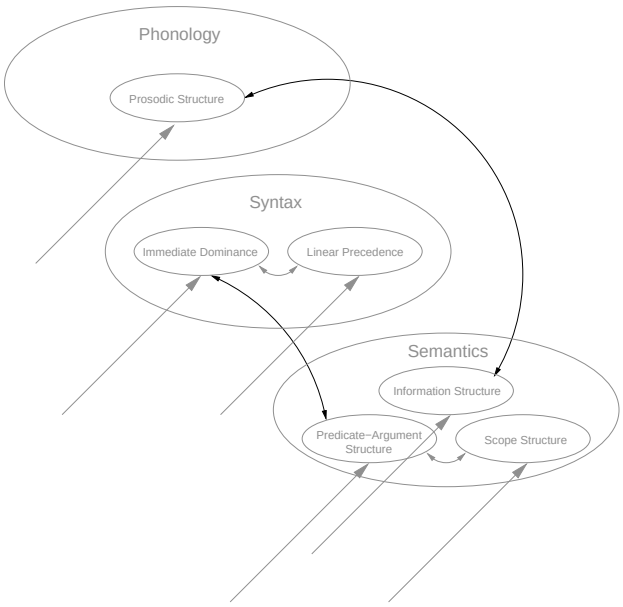


Figure 12.1.: The interfaces in the overall architecture of the example grammar

12.1. Syntax-Semantics Interface

The modularity of XDG allows us to specify the syntax-semantics interface solely in terms of the ID and PA dimensions on the ID/PA dimension. In particular, we do not need to take word order, scopal relationships, information structure or prosody into account. This is not to say that the syntax-semantics interface *must* be unrelated to these dimensions—only that it does not have to, which considerably reduces the complexity of the syntax-semantics interface and makes it less error-prone.

12.1.1. Types

Attributes. The lexical attributes of the ID/PA dimension consist of five vectors used to map PA edge labels to sets of ID edge labels for the linking principles, two sets of PA edge labels for the *LinkingMother principle* and the *PartialAgreement principle* (defined shortly), and a set of ID labels for the *LockingDaughters principle*:

```
defentrytype {linkDaughterEnd: vec("pa.label" set("id.label"))
              linkBelow1or2Start: vec("pa.label" set("id.label"))
              linkBelowStart: vec("pa.label" set("id.label"))
              linkAboveBelow1or2Start: vec("pa.label" set("id.label"))
              lockDaughters: set("id.label")
              linkMother: set("pa.label")
              linkAboveEnd: vec("pa.label" set("id.label"))
              agree: set("pa.label")}
```
(12.1)

12.1.2. Principles and Lexical Classes

The syntax-semantics interface is divided into four parts:

1. verbal arguments
2. modifiers
3. common nouns
4. relative clauses

Verbal Arguments. The largest part of the syntax-semantics interface consists of modeling the syntactic realization of verbal arguments. Given a verb node v on the PA dimension, its semantic argument v' can be realized on ID either:

1. as the dependent of v, or as the dependent of a dependent of v
2. as the dependent or as the dependent of a dependent of a superordinate verb of v
3. as a node below v

As an example for the first possibility, consider the ID/PA analysis in Figure 12.2 of the following sentence:

Peter assigns every task to a researcher. (12.2)

where the agent *Peter* and the patient *task* of *assigns* on the PA dimension are syntactically realized as dependents (subject and object) of *assigns*. The addressee *researcher* is realized as the dependent of the dependent *to*.

We implement the first possibility with the *LinkingBelow1or2Start principle*, which has the following declarative semantics: if for an edge from v to v' labeled l on d_1, the value of *linkBelow1or2Start* for v and l on d_3 is non-empty, then for at least one edge label l' in this set, there must either be an edge directly going from v to v' on d_2 labeled l', or an edge labeled l' going from v to another node v'', and one from v'' to v'.

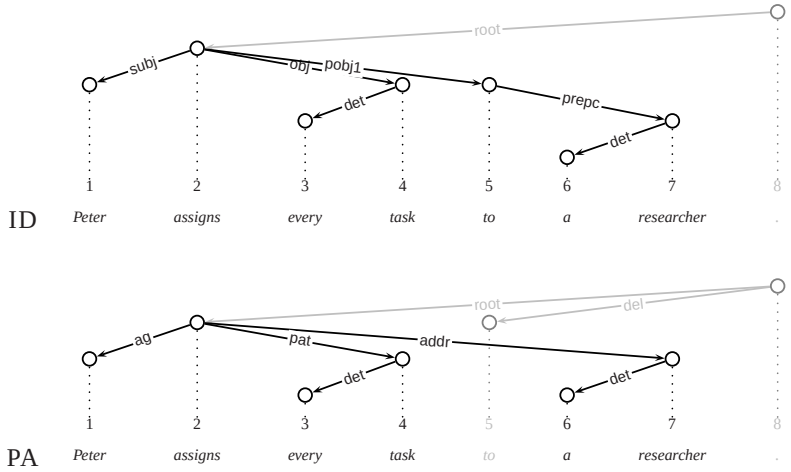


Figure 12.2.: ID/PA analysis of *Peter assigns every task to a researcher.*

Principle 21 (LinkingBelow1or2Start)**.**

$$\begin{aligned}&\mathit{linkingBelow1or2Start}_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} v' \wedge (d_3\ v).\mathit{lex.linkBelow1or2Start.l} \neq \emptyset \Rightarrow \\ &\exists l' : l' \in (d_3\ v).\mathit{lex.linkBelow1or2Start.l} \wedge \\ &v \xrightarrow{l'}_{d_2} v' \vee \exists v'' : v \xrightarrow{l'}_{d_2} v'' \wedge v'' \rightarrow_{d_2} v'\end{aligned} \tag{12.3}$$

We apply this principle as follows:

```
useprinciple "principle.linkingBelow1or2Start" {
  dims {D1: pa
        D2: id
        D3: idpa}
  args {Start: ^.D3.entry.linkBelow1or2Start}}
```
(12.4)

and specify the lexical attribute in the lexical class `"idpa_pat_obj"` which states that the patient is realized as an object, or `"idpa_addr_iobj"` which states that the addressee is realized by the indirect object:

```
defclass "idpa_pat_obj" {
  dim idpa {linkBelow1or2Start: {pat: {obj}}}}

defclass "idpa_addr_iobj" {
  dim idpa {linkBelow1or2Start: {addr: {iobj}}}}
```
(12.5)

The lexical class `"idpa_addr_pobj1"` states that the addressee is realized by prepositional object 1. For passives, we define the class `"idpa_ag_pobj2"` stating that the agent is realized by prepositional object 2:

```
defclass "idpa_addr_pobj1" {
   dim idpa {linkBelow1or2Start: {addr: {pobj1}}}}

defclass "idpa_ag_pobj2" {
  dim idpa {linkBelow1or2Start: {ag: {pobj2}}}}
```
(12.6)

The second possibility for the realization of verbal arguments is by a superordinate verb. An example is the sentence below, whose ID/PA analysis is displayed in Figure 12.3:

Peter seems to laugh. (12.7)

Here, the agent *Peter* of *laugh* is realized as the subject of the superordinate subject raising verb *seems* on the ID dimension.

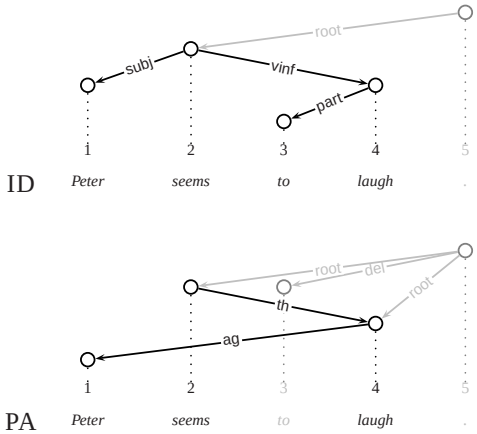


Figure 12.3.: ID/PA analysis of *Peter seems to laugh.*

The agents of subordinate verbs need not always be realized as subjects. In the example below, analyzed in Figure 12.4, the PP control verb *appeals* realizes the agent of *laugh* as its prepositional object:

Peter appeals to Mary to laugh. (12.8)

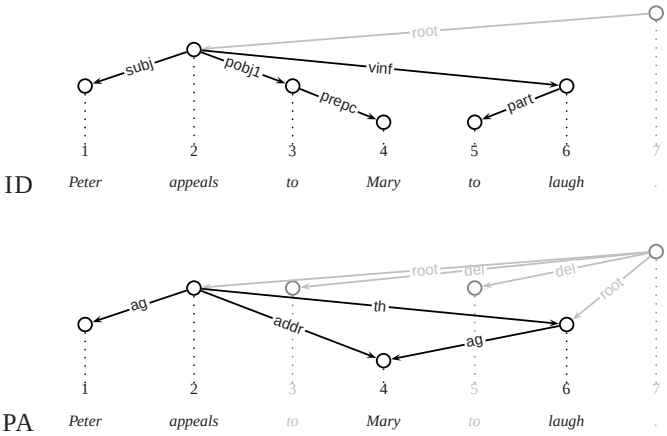


Figure 12.4.: ID/PA analysis of *Peter appeals to Mary to laugh.*

We implement this second possibility for the syntactic realization of verbal arguments with the *LinkingAboveBelow1or2Start principle*. Its declarative semantics are analogous to the LinkingBelow1or2Start principle, with the exception that for all nodes from v to v' on d_1, v' not necessarily has to be the dependent (or the dependent of a dependent) of v on d_2, but can also be the dependent of a superordinate node v'' of v on d_2.

Principle 22 (LinkingAboveBelow1or2Start)**.**

$$\begin{aligned}&\mathit{linkingAboveBelow1or2Start}_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} v' \wedge (d_3\ v).\mathit{lex.linkAboveBelow1or2Start.l} \neq \emptyset \Rightarrow \\ &\exists l' : l' \in (d_3\ v).\mathit{lex.linkAboveBelow1or2Start.l} \wedge \\ &\exists v'' : v'' \rightarrow^*_{d_2} v \wedge (v'' \xrightarrow{l'}_{d_2} v' \vee \exists v''' : v'' \xrightarrow{l'}_{d_2} v''' \wedge v''' \rightarrow_{d_2} v')\end{aligned} \tag{12.9}$$

We apply the principle as follows:

```
useprinciple "principle.linkingAboveBelow1or2Start" {
  dims {D1: pa
        D2: id
        D3: idpa}
  args {Start: ^.D3.entry.linkAboveBelow1or2Start}}
```
(12.10)

and use it in the lexical class `"idpa_ag_super"`, which states that the agent can be realized either as a subject, an object, an indirect object or a prepositional object of the verb itself or a superordinate verb:

```
defclass "idpa_ag_super" {
  dim idpa {linkAboveBelow1or2Start: {ag: {subj obj iobj pobj1 pobj2}}}}
```
(12.11)

This lexical class rules out e.g. the wrong analysis of the sentence below given in Figure 12.5, where *tries* incorrectly takes *Mary* and not *Peter* as its agent:

Peter tries to persuade Mary to sleep. (12.12)

The analysis is ruled out because *Mary* neither is a syntactic dependent or a syntactic dependency of a syntactic dependent of *tries* itself, nor of any superordinate verb.

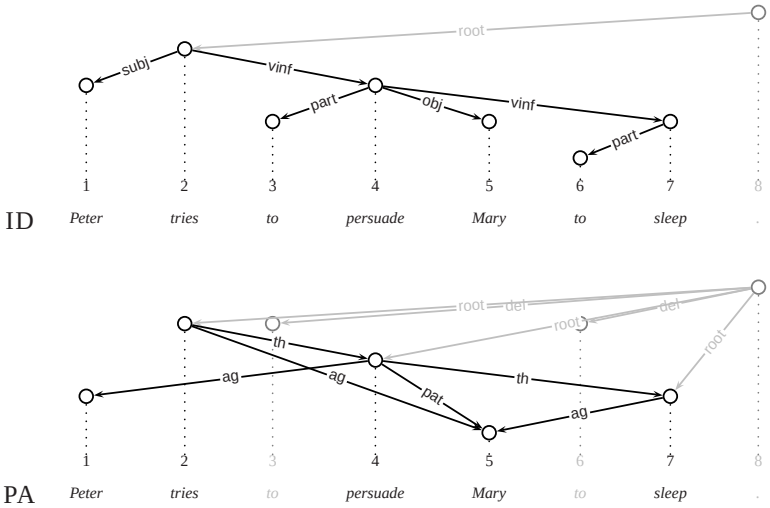


Figure 12.5.: Wrong ID/PA analysis of *Peter tries to persuade Mary to sleep.*

Verbs in passive form do not realize their agent but their patient as the subject of a superordinate verb, as in the example below, analyzed in Figure 12.6, where the patient of *Peter* of *loved* is realized as the subject of the superordinate passive auxiliary *is*:

Peter is loved by Mary. (12.13)

We capture this in the lexical class below:

```
defclass "idpa_pat_super" {
  dim idpa {linkAboveBelow1or2Start: {pat: {subj obj iobj pobj1 pobj2}}}}
```
(12.14)

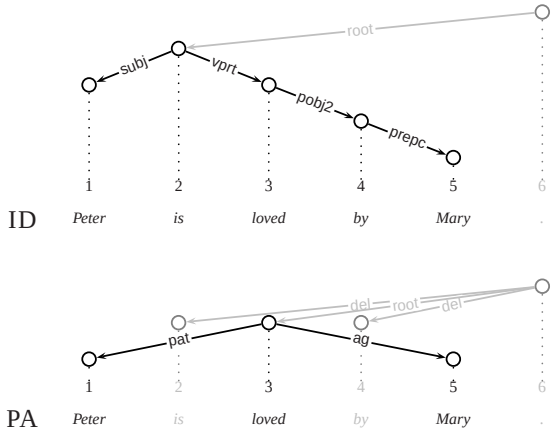


Figure 12.6.: ID/PA analysis of *Peter is loved by Mary.*

The third possibility for the syntactic realization of verbal arguments concerns themes, which can be realized either by infinitives or by subordinate clauses. In the examples above, e.g. in Figure 12.4, it seems that the theme argument is always realized by the corresponding full infinitive dependent. The analysis in Figure 12.7 of the sentence below however shows that the theme of a verb can also be realized further below:

Peter seems to have been persuaded to sleep. (12.15)

Here, the theme *persuaded* of *seems* is not realized as a syntactic dependent of *seems* but further below.

We capture this realization possibility with the *LinkingBelowStart principle*, which we apply as follows:

```
useprinciple "principle.linkingBelowStart" {
  dims {D1: pa
        D2: id
        D3: idpa}
  args {Start: ^.D3.entry.linkBelowStart}}
```
(12.16)

and use the lexical class `"idpa_th_vinf"` to state that the theme must be realized below the full infinitive dependent on the ID dimension:

```
defclass "idpa_th_vinf" {
  dim idpa {linkBelowStart: {th: {vinf}}}}
```
(12.17)

It seems as if the linking principles presented so far suffice to constrain the realization of the semantic arguments of verbs. But this is not quite true. Consider the correct analysis in Figure 12.8 of the sentence below, where the object raising verb *believes* does not have a patient on the PA dimension, but only an agent:

Peter believes Mary to laugh. (12.18)

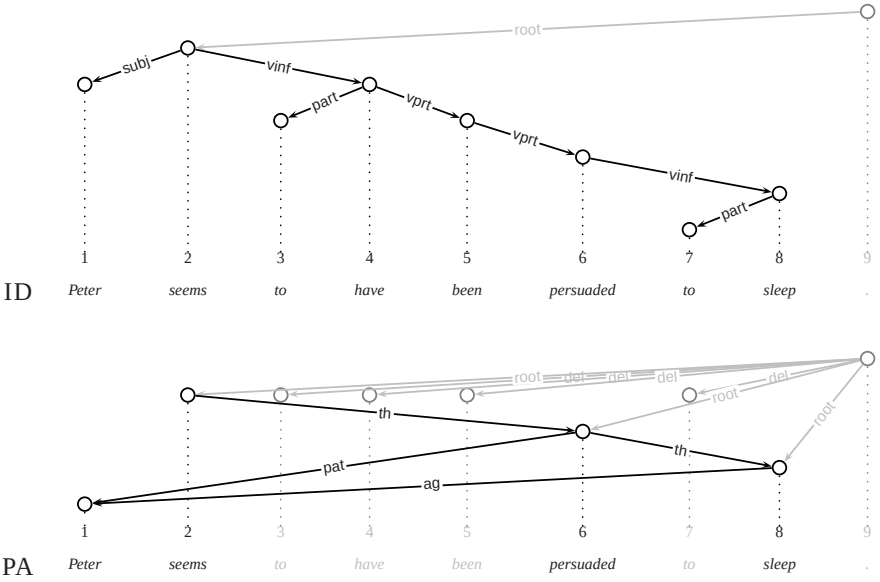


Figure 12.7.: ID/PA analysis of *Peter seems to have been persuaded to sleep.*

According to the lexical class `"idpa_ag_super"` (12.11) above, the agent can be realized by any nominal grammatical function on the ID dimension. But this means that the agent could also be realized by the object of *believes*, leading to the wrong analysis shown in Figure 12.9, where in addition, the agent of *laugh* is *Peter* and not *Mary*.

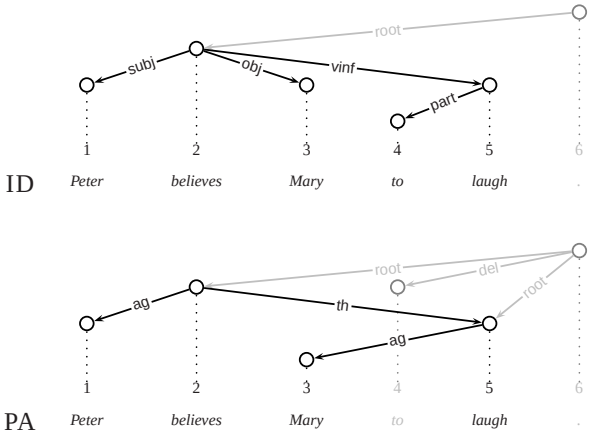


Figure 12.8.: ID/PA analysis of *Peter believes Mary to laugh.*

How can we rule out this analysis? The idea is to reuse the *LockingDaughters principle* (cf. principle 18 in chapter 10). Why can we not state this constraint on the PA dimension, where we also applied the LockingDaughters principle? On the PA dimension alone, we could only say that the agent of *believes* may not simultaneously be the agent of a subordinate verb. But this constraint is satisfied in Figure 12.9: the agent *Mary* of *believes* is in fact not simultaneously the agent of the subordinate verb *laugh*. What we need to state instead is a constraint spanning over both the PA dimension and the ID dimension that the subject can

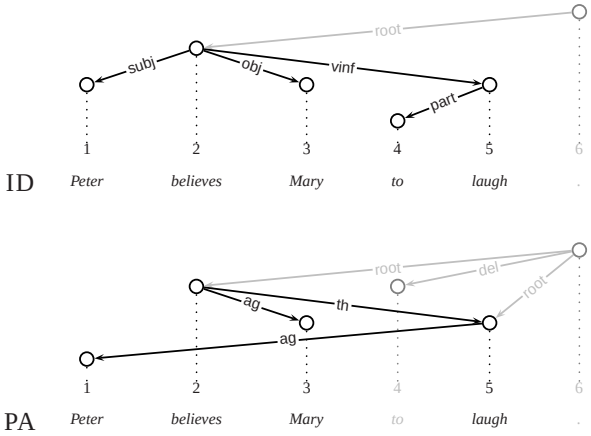


Figure 12.9.: Wrong ID/PA analysis of *Peter believes Mary to laugh.*

realize a semantic argument of the verb itself, but not of a subordinate verb. As a result, the subject *Peter* of *believes* can only the agent of itself, and not of the subordinate verb *laugh*.

We realize this idea by applying the LockingDaughters principle, and using the lexical attribute `lockDaughters`. As the dependents are locked on the ID dimension, which is a tree, we can safely set `ExceptAbove` to the empty set: there can be no nodes above on the ID dimension which are also mothers of the locked dependents. By the argument `Key`, we stipulate that the locked dependents may still be modified:

```
useprinciple "principle.lockingDaughters" {
  dims {D1: id
        D2: pa
        D3: idpa}
  args {LockDaughters: _.D3.entry.lockDaughters
        ExceptAbove: {}
        Key: {agm patm}}}
```
(12.19)

The lexical class `"idpa_objcr"` for object raising verbs such as *believes* in the example above locks the subject and the indirect object:

```
defclass "idpa_objcr" {
  dim idpa {lockDaughters: {subj iobj}}}
```
(12.20)

Modifiers. The arguments of modifiers on the PA dimension are realized by their syntactic heads on the ID dimension. As an example, consider the sentence below, analyzed in Figure 12.10:

With Peter, a pretty woman smiles today. (12.21)

where the agent of the adjective *pretty* is realized by its syntactic head, the noun *woman* on the ID dimension. Similarly, the theme of the adverb *today* and the prepositional adverb *with* are both realized by their syntactic head, the verb *smiles*.

We implement this idea using the *LinkingMother principle*:

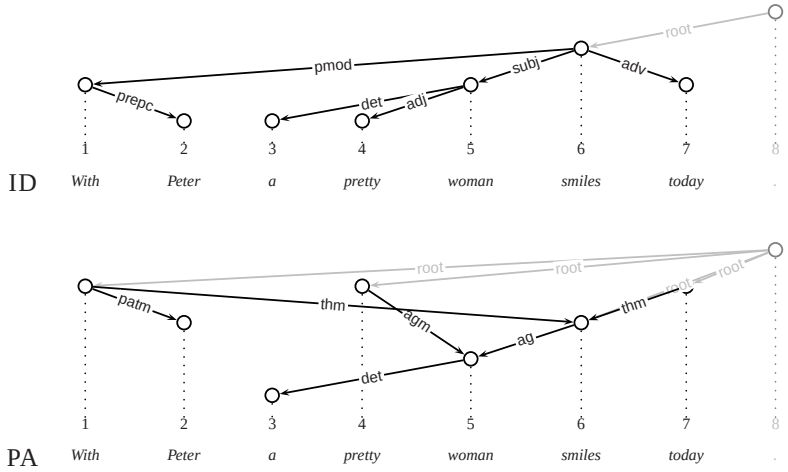


Figure 12.10.: ID/PA analysis of *With Peter, a pretty woman smiles today.*

```
useprinciple "principle.linkingMother" {
  dims {D1: pa
        D2: id
        D3: idpa}
  args {Which: ^.D3.entry.linkMother}}
```
(12.22)

and define the following lexical classes. For adjectives, the class `"idpa_adj"` stipulates that the agent of the adjective is realized by its syntactic head:

```
defclass "idpa_adj" {
  dim idpa {linkMother: {agm}}}
```
(12.23)

The class `"idpa_adv"` states the analogue for adverbs:

```
defclass "idpa_adv" {
  dim idpa {linkMother: {thm}}}
```
(12.24)

The patient of prepositional modifiers is realized as their `prepc` dependent on the ID dimension. For example, in Figure 12.10, the patient of the prepositional modifier *with* on PA is realized by its `prepc` dependent *Peter* on ID. This is expressed in the following two lexical classes for prepositional adjectives (`"idpa_padj"`) and prepositional adverbs (`"idpa_padv"`):

```
defclass "idpa_padj" {
  "idpa_adj"
  dim idpa {linkDaughterEnd: {patm: {prepc}}}}

defclass "idpa_padv" {
  "idpa_adv"
  dim idpa {linkDaughterEnd: {patm: {prepc}}}}
```
(12.25)

Common Nouns. The determiner of a common noun on PA is realized syntactically also as the determiner of the noun, as can be seen e.g. in Figure 12.12 above. We state this simple

constraint using the *LinkingDaughterEnd principle*:

```
useprinciple "principle.linkingDaughterEnd" {
  dims {D1: pa
        D2: id
        D3: idpa}
  args {End: ^.D3.entry.linkDaughterEnd}}
```
(12.26)

and using the lexical class below:

```
defclass "idpa_cnoun" {
  dim idpa {linkDaughterEnd: {det: {det}}}}
```
(12.27)

Relative Clauses. In our example grammar, we analyze relative clauses such as the following as shown in Figure 12.11:

Mary sees a woman who smiles. (12.28)

That is, on the ID dimension, the finite verb (here: *smiles*) heading the relative clause is a `rel` dependent of the modified noun (*woman*). On the PA dimension, the modified noun is an `agm` dependent of the relative pronoun (here: *who*).

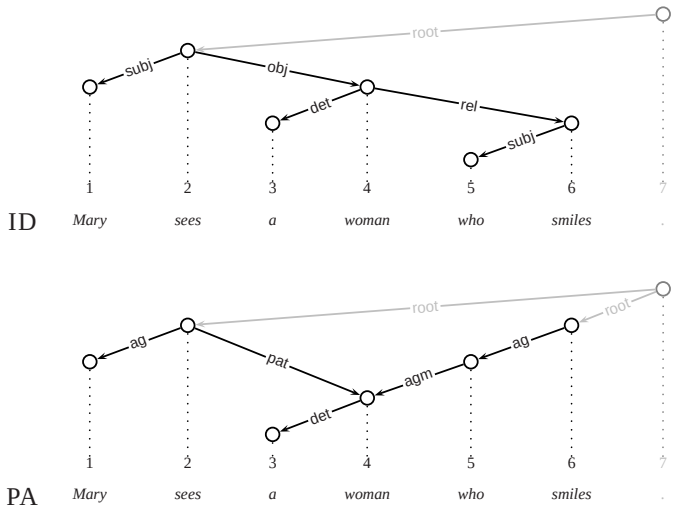


Figure 12.11.: ID/PA analysis of *Mary sees a woman who smiles.*

With respect to relative clauses, the syntax-semantics interface stipulates:

1. partial agreement of the relative pronoun with its `agm` dependent, i.e., the modified noun
2. the syntactic realization of the agent of the relative pronoun as a node above the relative pronoun, where the path to the node ends with an edge labeled `rel`

Partial agreement of the relative pronoun with the modified noun is motivated by the following contrast, which is caused by the relative pronoun and the modified noun having a gender mismatch:

Mary sees a woman who smiles.
Mary sees a woman that smiles.
**Mary sees a woman which smiles.* (12.29)

In the light of the notion of agreement in our grammar, where *agreement tuples* include also case, the agreement of relative pronouns and modified nouns is only partial. For example, the cases of the personal pronoun and the modified noun do not have to match: in *Mary sees a woman who smiles*, *woman* is accusative and *who* nominative. To express partial agreement, we introduce the *PartialAgreement principle*, which is defined analogously to the *Agreement principle* (cf. principle 9 in chapter 4), but stipulates that only a subset of the projections (lexical attribute *projs*) of the agreement tuple must agree.

Principle 23 (Partial Agreement)**.**

$$\begin{aligned} &\mathit{partialAgreement}_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} v' \wedge l \in (d_3\ v).\mathit{lex.agree} \Rightarrow \\ &\forall i \in (d_3\ v).\mathit{lex.projs} : (d_2\ v).\mathit{agr}.i \doteq (d_2\ v').\mathit{agr}.i \end{aligned} \tag{12.30}$$

We apply the principle as follows, where we set `Projs`, the set of projections of the agreement tuple which must agree, to the set containing only 3 (gender):

```
useprinciple "principle.partialAgreement" {
  dims {D1: pa
        D2: id
        D3: idpa}
  args {Agr1: ^.D2.attrs.agr
        Agr2: _.D2.attrs.agr
        Agree: ^.D3.entry.agree
        Projs: {3}}}
```
(12.31)

By the lexical class `"idpa_relpro_agree"`, we then state that the `agm` dependent of the relative pronoun on the PA dimension must agree with it in gender:

```
defclass "idpa_relpro_agree" {
  dim idpa {agree: {agm}}}
```
(12.32)

The syntax-semantics interface is secondly concerned with the syntactic realization of the agent of the relative pronoun. As can be seen from the analysis in Figure 12.11 above, the agent of the relative pronoun, i.e., the modified noun, can be found above the relative pronoun on the ID dimension, and the last edge on the path from the relative pronoun to the modified noun is labeled `rel`. We express this in our grammar applying the LinkingAboveEnd principle:

```
useprinciple "principle.linkingAboveEnd" {
  dims {D1: pa
        D2: id
        D3: idpa}
  args {End: ^.D3.entry.linkAboveEnd}}
```
(12.33)

and the accompanying lexical class `"idpa_relpro_link"`:

```
defclass "idpa_relpro_link" {
  dim idpa {linkAboveEnd: {agm: {rel}}}}
```
(12.34)

This linking specification also covers more complex cases such as *pied piping* constructions. Consider the pied piping example below, where the relative pronoun is a dependent of the prepositional adverb *with*:

Mary sees a woman with whom Peter smiles (12.35)

We show an analysis of the sentence in Figure 12.12. Here, the modified noun is also above the relative pronoun on the ID dimension, and the last edge on the path to it is also labeled with `rel`.

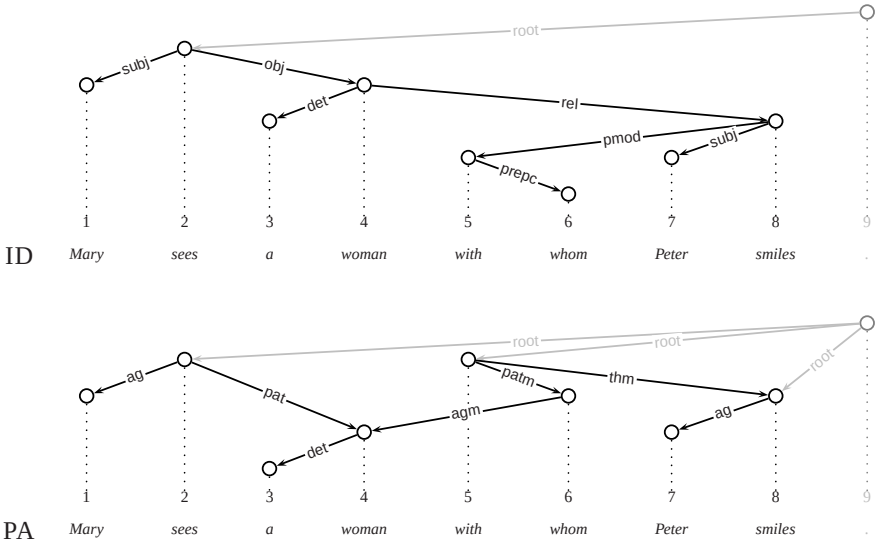


Figure 12.12.: ID/PA analysis of *Mary sees a woman with whom Peter smiles.*

12.2. Phonology-Semantics Interface

We realize the Phonology-Semantics interface by the PS/IS dimension, which constrains the relation of prosodic structure (PS) and information structure (IS). Its position in the overall architecture of the grammar is displayed in Figure 12.1 above, and it purpose is twofold:

1. As pitch accents and boundary tones are characteristic for either theme or rheme, to ensure that words carrying theme pitch accents and words followed by theme boundary tones only occur in themes, and analogously for rhemes.

2. Ensure that IS *constituents* are always contained in PS *constituents*.

The PS, IS and PS/IS dimensions constitute a modular adaptation of the prosodic account of information structure of Steedman (2000*a*). It is not connected with the account of information structure for TDG developed in (Kruijff & Duchier 2003), which also integrates other sources of information in addition to prosody.

12.2.1. Principles and Lexical Classes

Pitch Accents and Boundary Tones. Our first task is to ensure that words carrying theme pitch accents or words followed by theme boundary tones may only occur in themes. To this

end, we define the following lexical classes:

```
defclass "psis_th_pa" {
  "ps_pa1"
  "is_tf"}

defclass "psis_th_pabt" {
  "ps_pa1bt1"
  "is_tf"}

defclass "psis_th_bt" {
  "ps_bt1"
  "is_nf"}
```
(12.36)

where "psis_th_pa" states that words carrying pitch accent 1 can only be the focus of a theme, "psis_th_pabt" that words simultaneously carrying pitch accent 1 and followed by boundary tone 1 can also only be the focus of a theme, and "psis_th_bt" that words followed by boundary tone 1 must be non-foci.

The following classes state the analogues for rhemes:

```
defclass "psis_rh_pa" {
  "ps_pa2"
  "is_rf"}

defclass "psis_rh_pabt" {
  "ps_pa2bt2"
  "is_rf"}

defclass "psis_rh_bt" {
  "ps_bt2"
  "is_nf"}
```
(12.37)

Unaccented words are covered by the lexical class "psis_ua", which stipulates that they must be non-foci:

```
defclass "psis_th_ua" {
  "ps_ua"
  "is_nf"}
```
(12.38)

As an example, consider the analysis in Figure 12.13 of the sentence below, where *Marcel* carries a theme pitch accent, *proves* is followed by a theme boundary tone, and *completeness* simultaneously carries a rheme pitch accent and is followed by a rheme boundary tone. Both *Marcel* and *proves* are correctly in the theme of the IS analysis, and *completeness* in the rheme:

Marcel_L+H proves_LH% completeness_H*_LL%.* (12.39)

IS and PS Constituents. On the IS dimension, words carrying pitch accents are the heads of IS constituents, and on the PS dimension, words followed by boundary tones are the heads of PS constituents. The relation between IS constituents and PS constituents is constrained as follows: each IS constituent must either correspond to a PS constituent or be contained in one. For example, consider the sentence below:

*Marcel_LH% proves completeness_H*_LL%.* (12.40)

The sentence licenses two analyses, which we show in Figure 12.14 and Figure 12.15, where:

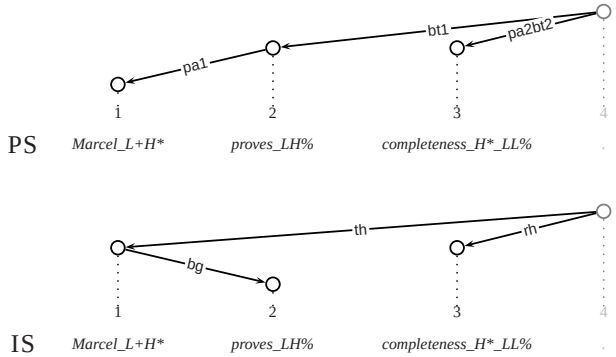


Figure 12.13.: PS/IS analysis of *Marcel_L+H* proves_LH% completeness_H*_LL%*.

1. In Figure 12.14, the IS and PS constituents *Marcel* and *proves completeness* converge.
2. In Figure 12.15, the IS constituent *Marcel* converges with the PS constituent *Marcel*, and the IS constituents *proves* and *completeness* are contained in the PS constituent *proves completeness*.

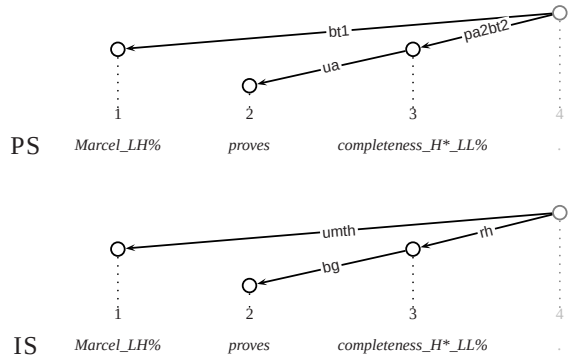


Figure 12.14.: PS/IS analysis of *Marcel_LH% proves completeness_H*_LL%*.

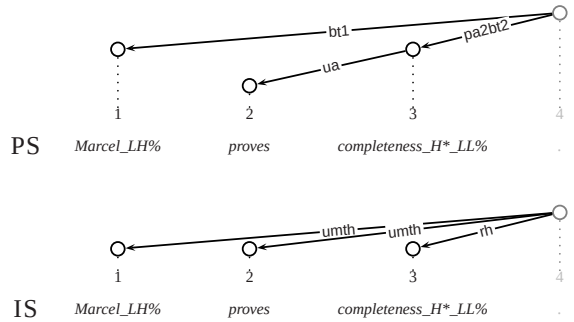


Figure 12.15.: PS/IS analysis of *Marcel_LH% proves completeness_H*_LL%*.

We express this relation between PS and IS constituents using the *Subgraphs principle*, which has the following declarative semantics: given three dimensions d_1, d_2 and d_3, for all

nodes v and v' and for all edge labels l on d_1, if v' is below an edge labeled l emanating from v on d_1 and the lexically specified set *subgraphsStart* is non-empty for l, then it must contain at least one edge label l' and v' must also be below an edge labeled l' on d_2, also emanating from v.

Principle 24 (Subgraphs)**.**

$$\begin{aligned} &\mathit{subgraphs}_{d_1,d_2,d_3} = \forall v, v' : \forall l : \\ &v \xrightarrow{l}_{d_1} \rightarrow^*_{d_1} v' \wedge (d_3\ v).\mathit{lex}.\mathit{subgraphsStart}.l \neq \emptyset \Rightarrow \\ &\exists l' : l' \in (d_3\ v).\mathit{lex}.\mathit{subgraphsStart}.l \wedge v \xrightarrow{l'}_{d_2} \rightarrow^*_{d_2} v' \end{aligned} \tag{12.41}$$

We apply the principle as follows, stating that all elements in the theme must be contained in the corresponding PS constituent headed by a word followed by boundary tone 1, and analogously for the rheme:

```
useprinciple "principle.subgraphs" {
  dims {D1: is
        D2: ps
        D3: psis}
  args {Start: {th: {bt1 pa1bt1}
                rh: {bt2 pa2bt2}}}}
```
(12.42)

We do not need to constrain unmarked themes since the corresponding IS constituents always consist of precisely one word, which is always contained in one of the available PS constituents.

As an example, Figure 12.16 shows an ill-formed PS/IS analysis excluded by the Subgraphs principle. The PS analysis is the same as in Figure 12.13 above, defining the PS constituents *Marcel proves* and *completeness*. The IS analysis has theme *Marcel* and rheme *completeness*, and the latter has *proves* in its background. The resulting IS constituents are *Marcel* (theme) and *proves completeness* (rheme). This is wrong, since the IS rheme constituent *proves completeness* is not contained in the corresponding PS constituent *completeness*.

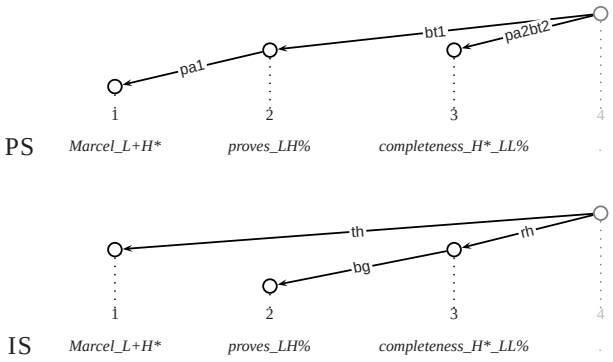


Figure 12.16.: Wrong PS/IS analysis of *Marcel_L+H* proves_LH% completeness_H*_LL%*

12.3. Emerging Phenomena

The syntax-semantics interface of our grammar covers arbitrarily complicated control, raising and auxiliary constructions, and in combination with the XDK constraint solver, supports e.g. attachment underspecification out of the box.

12.3.1. Control, Raising and Auxiliary Constructions

An example complicated case of control, raising, and auxiliary constructions is shown in the analysis of the sentence below in Figure 12.17. The sentence includes the perfect auxiliary (*has*), the subject raising verb (*seemed*), the passive auxiliary (*to be*), the object control verb (*persuaded*) and the subject control verb (*to try*):

Peter has seemed to be persuaded to try to sleep by Mary. (12.43)

The analysis of this sentence in our grammar is shown in Figure 12.17. In fact, our grammar correctly licenses precisely this analysis and no other. Notice the simplicity in particular of the PA analysis of this very complicated construction: it is easy to see that *Peter* is the patient of *persuaded*, and the agent of *try* and *sleep*. *Mary* is the agent of *persuaded*, and *sleep* is the theme of *try*, which is the theme of *persuaded*, which is the theme of *seemed*.

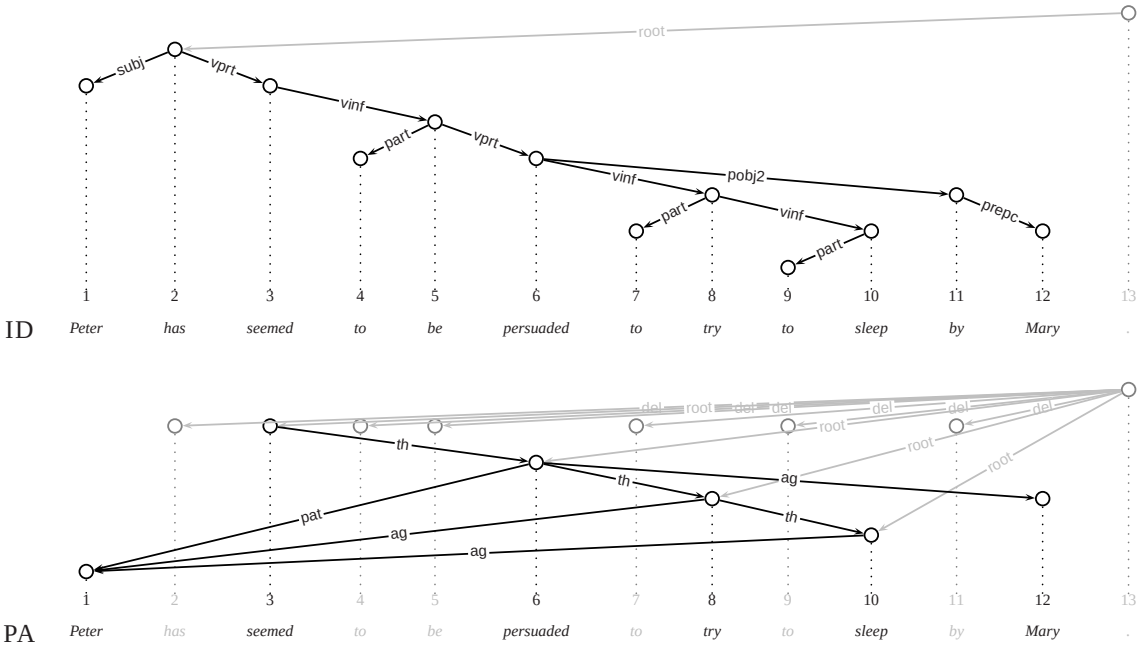


Figure 12.17.: Underspecified ID/PA analysis of *Peter has seemed to be persuaded to try to sleep by Mary.*

12.3.2. PP-Attachment Underspecification

In combination with the XDK constraint parser, our grammar not only supports the underspecification of scope as in (10.5.1), but also of any other linguistic aspect. For instance, it is possible to postpone the enumeration of models on the ID and PA dimensions, which gives us underspecification of PP-attachment for free. As an example, consider the sentence below, which is ambiguous between the reading where the PP (prepositional phrase) *with a telescope* modifies the verb *sees* or the noun *man*:

Mary sees the man with a telescope. (12.44)

If we postpone the enumeration of models on the ID and PA dimensions, we get the underspecified ID/PA analysis shown in Figure 12.18, where the constraint parser already knows that the PP must eventually be below *sees* (as indicated by the dotted edge from *sees* to *with*). Underspecified analyses like this could be a starting point for disambiguation, e.g. using statistically driven *oracles*.

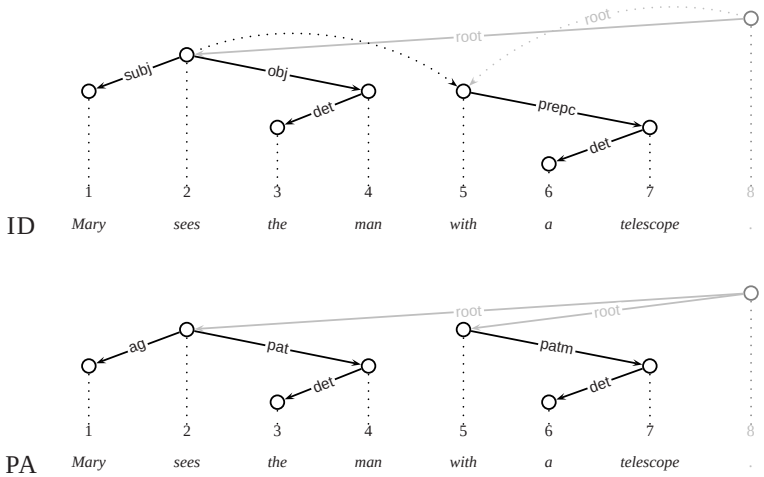


Figure 12.18.: Underspecified ID/PA analysis of *Mary sees the man with a telescope.*

12.4. Summary

We have introduced the syntax-semantics interface and the phonology-semantics interface of the example grammar. The syntax-semantics interface is simple and elegant, while covering very complicated control, raising, and auxiliary constructions, and leading to the emergence of PP-attachment underspecification. The simplicity of the interface is the result of the modularity of XDG, which allows us to concentrate entirely on the syntactic realization of semantic arguments, and to factor out all other issues such as word order, scope, information structure and prosody. It is of course possible to bring these factors back in and e.g. add constraints to reduce the number of scopal readings for certain word orders, but the basic interface would remain the same.

13. Conclusion

This chapter sums up the thesis and points out ideas for future work.

13.1. Summary

We have developed the grammar formalism of Extensible Dependency Grammar (XDG), combining dependency grammar, model-theoretic syntax and Jackendoff's (2002) parallel grammar architecture. This combination yields a novel, radically modular design allowing to describe arbitrary many linguistic aspects within the same formalism, but at the same time largely independently from each other. This significantly simplifies the modeling of linguistic phenomena, since individual aspects such as grammatical functions, word order or predicate-argument structure can also be modeled individually. For example, although word order variation is irrelevant for the interface from syntax to predicate-argument structure, previous approaches still have to take it into account, which unnecessarily complicates their syntax-semantics interface. In XDG, both aspects can be completely dissociated. This approach makes many otherwise problematic linguistic phenomena such as extraction, scope ambiguities and control and raising simply fall out as by-products, without any further stipulation.

This thesis contained three contributions in order to show that XDG is not only an abstract idea, but that it can also be concretely realized: the first formalization of XDG as a multigraph description language in higher order logic, the first implementation of XDG within an extensive grammar development system, and the first application of this system to natural language.

The first formalization of XDG was developed in part I, where we also showed how the core concepts of dependency grammar, including lexicalization, valency and order, can be realized in XDG. This prepared the ground for first investigations of the expressivity and the computational complexity of XDG. XDG is at least as expressive as context-free grammar, and that also non-context-free languages such as $a^n b^n c^n$ and linguistic benchmarks such as cross-serial dependencies and scrambling can be elegantly modeled. The price for this degree of expressivity is that the XDG recognition problem is NP-hard.

Despite this high complexity, the XDG constraint parser developed in part II of the thesis is reasonably fast on smaller, handwritten grammars. Around the parser, we built an extensive grammar development environment, the XDG Development Kit (XDK), which allows to comfortably create grammars by hand or automatically and then to test them. The XDK is important not only for the development of the XDG grammar theory, but it has also been successfully used for teaching.

In part III, we developed a grammar for a fragment of English, which modeled syntax, semantics and also phonology. We demonstrated how complicated phenomena such as extraction (including pied piping) in syntax, scope ambiguities in the semantics, and control

and raising in the syntax-semantics interface simply fall out as by-products of the modular grammar description, and do not have to be explicitly stipulated.

13.2. Future Work

In this thesis, we have shown that XDG is perfectly able to model and process smaller fragments of e.g. English, and due to its modularity, very elegantly so. Whether it is possible to model and process realistic grammars in XDG remains an open question. Finding an answer to this question must be the next step. There are two reasons to be optimistic: firstly, the complexity of established grammar formalisms like GPSG and LFG is at least as high (Barton, Berwick & Ristad 1987), while they *can* be efficiently processed in practice, and secondly, the basic design principle of XDG, modularity, clearly speaks in favor of scalability.

We plan to answer the question from two directions. In the first, the algorithmic direction, we want to deepen our understanding of the expressivity and the computational complexity of XDG, and its relation to other multi-dimensional grammar formalisms such as LFG, STAG and *Generalized Multitext Grammars* (GMTG) (Melamed, Satta & Wellington 2004). Our goal is to find restrictions of XDG which on the one hand leave as much of the expressivity intact, but on the other hand significantly reduce the complexity of XDG parsing. For instance, it would be interesting to see how much of XDG could be carried over to GMTG, which is also multi-dimensional, but contrary to XDG parsable in polynomial time.

In the second direction, that of constraint programming, we plan to profile the constraint parser of the XDK to find out what has gone wrong previously when it was used for large-scale parsing (Möhl 2004, Bojar 2004), to rewrite the parser using the new and more efficient *Gecode* constraint library (Schulte & Stuckey 2004), and to find global constraints for XDG parsing—so far, the constraint parser does not use a single one. Global constraints are usually indispensable for efficient constraint programming (Beldiceanu & Contjean 1994, Henz, Müller & Thiel 2004), hence this line of future work could prove very fruitful.

Further future work includes the continuation of work on the *distribution strategy* of the constraint parser to optimize the shape of the search tree, as has been shown by a prototype of the *NEGRA* project (Smolka & Uszkoreit 1996–2001) by Denys Duchier and Thorsten Brants (p.c.). This could be complemented by continuing the line of work on *guided search* (Dienes et al. 2003, Narendranath 2004), where the authors use A^* search to find the optimal solution first. We also consider optimizing the parser using the technique of *supertagging* (Joshi & Bangalore 1994, Clark & Curran 2004) to reduce lexical ambiguity, and by using the technique of *segmentation* proposed in (Kubon 2001).

XDG grammar theory is also far from complete—interesting future work includes finding an account of *coordination and ellipsis*. Also, it is not at all clear how to best do XDG *grammar induction* from treebanks (Korthals 2003, Bojar 2004). Finally, the reversibility of XDG has already been exploited for *generation* in combination with TAG in (Koller & Striegnitz 2002), but *pure* XDG generation, first discussed in (Debusmann 2004*b*) and (Pelizzoni & das Gracas Volpe Nunes 2005), would have the advantage that the same grammar could be used for parsing and for generation.

Appendix

A. Lattice Functors

In this appendix, we describe the *lattice functors* of the XDK, which provide functionality for the metagrammar compiler, the constraint parser and the visualizer of the XDK, as displayed in Figure A.1.

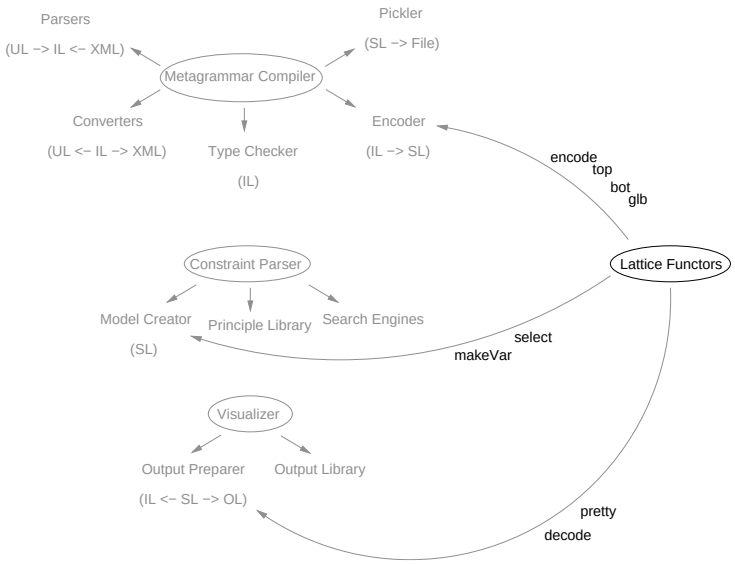


Figure A.1.: The lattice functors in the XDK architecture

Each type of the XDK description language corresponds to such a lattice functor, which is an ADT implementing the following methods:

- `encode`: encode IL terms into sets of SL core terms
- `top`, `bot`, `glb`: top, bottom and greatest lower bound of SL core terms
- `makeVar`: create an SL constraint variable
- `select`: efficiently select one SL core term from a list of SL core terms
- `decode`: convert SL core terms into IL core terms
- `pretty`: convert SL core terms into OL terms (for pretty printing)

In the following, we will write *lf* `T` for the lattice functor of type `T`. In our explanations of the methods, we use Mozart/Oz pseudo code instead of the actual Mozart/Oz code for better readability.

A.1. Encode

The lattice functors support the encoding of terms in IL syntax into sets[1] of core terms in SL syntax, proceeding in two steps[2]:

1. interpretation: terms are interpreted as sets of core terms
2. compilation: the core terms are compiled into SL syntax for further processing in the constraint solver

Given a lattice functor `Lat`, the encode method of the lattice functors is thus defined as:

$$\texttt{Lat.encode t} \;=\; \{\texttt{Lat.compile t}' \mid \texttt{t}' \in \texttt{Lat.interpret t}\} \tag{A.1}$$

A.1.1. Interpretation

The interpretation function `Lat.interpret` is defined as follows.

- atoms and integers, given $\texttt{Lat} = \mathit{lf}\ \{\texttt{a}_1 \ldots \texttt{a}_n\}$ or $\texttt{Lat} = \mathit{lf}\ \texttt{string}$ or $\texttt{Lat} = \mathit{lf}\ \texttt{int}$:

  $$\texttt{Lat.interpret t} \;=\; \{\texttt{t}\} \tag{A.2}$$

  That is, the interpretation of terms `t` of these types is the singleton set containing `t`.

- sets, given $\texttt{Lat} = \mathit{lf}\ \texttt{set(T)}$ or $\texttt{Lat} = \mathit{lf}\ \texttt{iset(T)}$, $\texttt{Lat}' = \mathit{lf}\ \texttt{T}$:

  $$\begin{array}{l}\texttt{Lat.interpret}\ \{\texttt{t}_1 \ldots \texttt{t}_n\} = \\ \{\{\texttt{t}'_1 \ldots \texttt{t}'_n\} \mid \texttt{t}'_1 \in \texttt{Lat}'\texttt{.interpret}\ \texttt{t}_1, \ldots, \texttt{t}'_n \in \texttt{Lat}'\texttt{.interpret}\ \texttt{t}_n\}\end{array} \tag{A.3}$$

  The interpretation of a set is the set of all sets described by it. For example:

  $$\texttt{Lat.interpret}\ \{(\texttt{subj} \mid \texttt{obj})\ \texttt{adv}\} \;=\; \{\{\texttt{subj adv}\}, \{\texttt{obj adv}\}\} \tag{A.4}$$

- infinite sets of integers, given $\texttt{Lat} = \mathit{lf}\ \texttt{set(int)}$, $\texttt{Lat} = \mathit{lf}\ \texttt{iset(int)}$ or $\texttt{Lat} = \mathit{lf}\ \texttt{card}$:

  $$\texttt{Lat.interpret}\ \{\texttt{i}_1 \ldots \texttt{i}_n \ldots\} \;=\; \{\{\texttt{i}_1 \ldots \texttt{i}_n \ldots\}\} \tag{A.5}$$

  The interpretation of an infinite sets of integers `t` is the singleton set containing `t`.

- lists, given $\texttt{Lat} = \mathit{lf}\ \texttt{list(T)}$ and $\texttt{Lat}' = \mathit{lf}\ \texttt{T}$:

  $$\begin{array}{l}\texttt{Lat.interpret}\ [\texttt{t}_1 \ldots \texttt{t}_n] = \\ \{[\texttt{t}'_1 \ldots \texttt{t}'_n] \mid \texttt{t}'_1 \in \texttt{Lat}'\texttt{.interpret}\ \texttt{t}_1, \ldots, \texttt{t}'_n \in \texttt{Lat}'\texttt{.interpret}\ \texttt{t}_n\}\end{array} \tag{A.6}$$

  A list is interpreted as the set of lists which is describes.

[1] In the actual implementation, the sets are implemented as lists.

[2] Contrary to the actual implementation, which interleaves the two steps for efficiency, we present them separately here for clarity.

- tuples, given $\mathtt{Lat} = \mathit{lf}\ \mathtt{tuple}(\mathtt{T_1} \ldots \mathtt{T_n})$, $\mathtt{Lat_1} = \mathit{lf}\ \mathtt{T_1}, \ldots, \mathtt{Lat_n} = \mathit{lf}\ \mathtt{T_n}$:

$$\begin{array}{l}\mathtt{Lat.interpret}\ [\mathtt{t_1 g} \ldots \mathtt{t_n}] = \\ \{[\mathtt{t'_1} \ldots \mathtt{t'_n}] \mid \mathtt{t'_1} \in \mathtt{Lat_1.interpret}\ \mathtt{t_1}, \ldots, \mathtt{t'_n} \in \mathtt{Lat_n.interpret}\ \mathtt{t_n}\}\end{array} \tag{A.7}$$

  A tuple is interpreted as the set of tuples which it describes.

- record specifications and empty records, given $\mathtt{Lat} = \mathit{lf}\ \{\mathtt{a_1} : \mathtt{T_1} \ldots \mathtt{a_n} : \mathtt{T_n}\}$, $\mathtt{Lat_1} = \mathit{lf}\ \mathtt{T_1}, \ldots, \mathtt{Lat_n} = \mathit{lf}\ \mathtt{T_n}$, $\mathtt{t} = \{\mathtt{a'_1} : \mathtt{t_1} \ldots \mathtt{a'_k} : \mathtt{t_k}\}$, and writing $\mathtt{t.a}$ for the value of attribute $\mathtt{a}$ of $\mathtt{t}$:

$$\begin{array}{l}\mathtt{Lat.interpret}\ \mathtt{t} = \\ \{\{\mathtt{a_1} : \mathtt{t_1} \ldots \mathtt{a_n} : \mathtt{t_n}\} \mid \text{for } 1 \leq \mathtt{i} \leq \mathtt{n},\ \mathtt{t_i} = \mathtt{Lat_1.top} \text{ if } \mathtt{a_i} \notin \{\mathtt{a'_1}, \ldots, \mathtt{a'_k}\}, \\ \qquad\qquad \text{otherwise } \mathtt{t_i} \in \mathtt{Lat_i.interpret}\ \mathtt{t.a_i}\}\end{array} \tag{A.8}$$

  That is, any omitted attribute is set to the top value of its lattice. Otherwise, the values of the attributes are set to those described in the record specification.

- cardinalities, given $\mathtt{Lat} = \mathit{lf}\ \mathtt{card}$:

$$\begin{array}{rcl}\mathtt{Lat.interpret}\ \mathtt{!} & = & \{\{\mathtt{1}\}\} \\ \mathtt{Lat.interpret}\ \mathtt{?} & = & \{\{\mathtt{0\ 1}\}\} \\ \mathtt{Lat.interpret}\ * & = & \{\{\mathtt{0\ 1\ 2} \ldots\}\} \\ \mathtt{Lat.interpret}\ + & = & \{\{\mathtt{1\ 2} \ldots\}\} \\ \mathtt{Lat.interpret}\ \#\{\mathtt{i_1} \ldots \mathtt{i_n}\} & = & \{\{\mathtt{i_1} \ldots \mathtt{i_n}\}\} \\ \mathtt{Lat.interpret}\ \#[\mathtt{i_1\ i_2}] & = & \{\{\mathtt{i'_1} \ldots \mathtt{i'_2}\}\}\end{array} \tag{A.9}$$

  The interpretation of a cardinality $\mathtt{c}$ is the singleton set containing $\mathtt{c}$.

- valencies, given $\mathtt{Lat} = \mathit{lf}\ \mathtt{valency}(\mathtt{T})$ and $\mathtt{Lat'} = \mathit{lf}\ \mathtt{card}$:

$$\mathtt{Lat.interpret}\ \{\mathtt{a_1\ c_1} \ldots \mathtt{a_n\ c_n}\} = \mathtt{Lat.interpret}\ \{\mathtt{a_1} : \mathtt{c_1} \ldots \mathtt{a_n} : \mathtt{c_n}\} \tag{A.10}$$

  Valencies are interpreted as records.

- tops, bottoms and greatest lower bounds, given $\mathtt{Lat} = \mathit{lf}\ \mathtt{T}$:

$$\mathtt{Lat.interpret\ top} = \{\mathtt{Lat.top}\} \tag{A.11}$$

$$\mathtt{Lat.interpret\ bot} = \{\mathtt{Lat.bot}\} \tag{A.12}$$

$$\begin{array}{l}\mathtt{Lat.interpret}\ \mathtt{t_1} \,\&\, \mathtt{t_2} = \\ \{\mathtt{Lat.glb}\ \mathtt{t'_1}\ \mathtt{t'_2} \mid \mathtt{t'_1} \in \mathtt{Lat.interpret}\ \mathtt{t_1}, \mathtt{t'_2} \in \mathtt{Lat.interpret}\ \mathtt{t_2}\}\end{array} \tag{A.13}$$

  That is, the greatest lower bound of two terms $\mathtt{t_1}$ and $\mathtt{t_2}$ is interpreted as the set of core terms described by it.

- alternations, given $\mathtt{Lat} = \mathit{lf}\ \mathtt{T}$:

$$\mathtt{Lat.interpret}\ \mathtt{t_1} \,|\, \mathtt{t_2} = (\mathtt{Lat.interpret}\ \mathtt{t_1}) \cup (\mathtt{Lat.interpret}\ \mathtt{t_2}) \tag{A.14}$$

  An alternation between $\mathtt{t_1}$ and $\mathtt{t_2}$ is interpreted as the set union of the interpretations of $\mathtt{t_1}$ and $\mathtt{t_2}$.

- set generators, given $\texttt{Lat} = \mathit{lf}\ \texttt{set(tuple(T}_1 \ldots \texttt{T}_n\texttt{))}$ or $\texttt{Lat} = \mathit{lf}\ \texttt{iset(tuple(T}_1 \ldots \texttt{T}_n\texttt{))}$, $\texttt{Lat}_1 = \mathit{lf}\ \texttt{T}_1$, …, $\texttt{Lat}_n = \mathit{lf}\ \texttt{T}_n$:

$$\texttt{Lat.interpret \$ g} = \{\texttt{Lat.gInterpret g}\} \tag{A.15}$$

where Lat.gInterpret is defined as:

$$\texttt{Lat.gInterpret a} = \{[\texttt{a}_1 \ldots \texttt{a}_n] \mid \begin{array}{l}\text{for } 1 \leq \texttt{i} \leq \texttt{n}, \texttt{a}_\texttt{i} = \texttt{a} \text{ if } \texttt{a} \in \texttt{T}_\texttt{i}, \\ \text{otherwise } \texttt{a}_\texttt{i} \in \texttt{T}_\texttt{i}\}\end{array} \tag{A.16}$$

That is, the interpretation of the atom a is the set of tuples with a at projection i if a is in the domain $\texttt{T}_\texttt{i}$ of that projection, and with any of the elements of $\texttt{T}_\texttt{i}$ at the other projections.

$$\texttt{Lat.gInterpret g}_1\texttt{\&g}_2 = (\texttt{Lat.gInterpret g}_1) \cap (\texttt{Lat.gInterpret g}_2) \tag{A.17}$$

$$\texttt{Lat.gInterpret g}_1\texttt{|g}_2 = (\texttt{Lat.gInterpret g}_1) \cup (\texttt{Lat.gInterpret g}_2) \tag{A.18}$$

It is important that the interpretation of conjunctions (&) and disjunctions (|) within a set generator is different from that outside a set generator. Within, they are interpreted as a single term: the set of tuples licensed by the set generator. Outside, they are interpreted as a set of terms: the set of core terms described by the term to be interpreted. As a consequence, using disjunctions outside a set generator multiplies the number of generated lexical entries, and should therefore be used with caution, whereas set generator disjunctions do not.

- orders, given $\texttt{Lat} = \mathit{lf}\ \texttt{set(tuple(T T))}$ or $\texttt{Lat} = \mathit{lf}\ \texttt{iset(tuple(T T))}$, $\texttt{Lat}' = \mathit{lf}\ \texttt{T}$:

$$\begin{array}{l}\langle \texttt{t}_1 \ldots \texttt{t}_n \rangle = \\ \{\{[\texttt{t}'_\texttt{i}\ \ \texttt{t}'_\texttt{j}] \mid 1 \leq \texttt{i}' < \texttt{j}' \leq \texttt{n}\} \mid \texttt{t}'_1 \in \texttt{Lat.interpret t}_1, \ldots, \texttt{t}'_n \in \texttt{Lat.interpret t}_n\}\end{array} \tag{A.19}$$

That is, the interpretation of an order $\langle \texttt{t}_1 \ldots \texttt{t}_n \rangle$ is the set of all sets of pairs whose first projection precedes the right projection in $\langle \texttt{t}'_1 \ldots \texttt{t}'_n \rangle$, where $\texttt{t}'_\texttt{i} \in \texttt{Lat.interpret t}_\texttt{i}$ for all $1 \leq \texttt{i} \leq \texttt{n}$.

- concatenations, given $\texttt{Lat} = \mathit{lf}\ \texttt{string}$:

$$\texttt{Lat.interpret t}_1\texttt{@t}_2 = \{\texttt{t}'_1\texttt{t}'_2 \mid \begin{array}{l}\texttt{t}'_1 \in \texttt{Lat.interpret t}_1, \\ \texttt{t}'_2 \in \texttt{Lat.interpret t}_2\}\end{array} \tag{A.20}$$

- feature paths:

$$\texttt{Lat.interpret p} = \{\texttt{p}\} \tag{A.21}$$

- type annotations:

$$\texttt{Lat.interpret t :: T} = \{\texttt{t}\} \tag{A.22}$$

i.e., we simply discard the type annotations in the interpretation step.

A.1.2. Compilation

In the second step, we compile the core terms obtained in the interpretation into Mozart/Oz SL syntax for further processing the constraint solver. Here, *feature paths* bring in a slight complication, as they can only be resolved dynamically during parsing. We solve this complication by lifting the type of a compiled core term to a function expecting two *node records*.

Here is the definition of compilation, where we write $\backslash x_1, \ldots, x_n.\ e$ for an Oz function abstracting over $x_1, \ldots, x_n$ in e, and $e\ \ e_1 \ldots e_n$ for the application of function e to the arguments $e_1 \ldots e_n$:

- atoms from a finite domain, given $\mathtt{Lat} = \mathit{lf}\ \{a_1 \ldots a_n\}$, where the atoms $a_1 \ldots a_n$ are in lexical order (defined by the function `Value.'<'` of Mozart/Oz):

$$\mathtt{Lat.compile}\ a_i\ =\ \backslash v, v'.\ i \tag{A.23}$$

  That is, we encode the ith element of the sorted finite domain as the integer i.

- atoms of type `string`, given $\mathtt{Lat} = \mathit{lf}$ `string`:

$$\mathtt{Lat.compile}\ a\ =\ \backslash v, v'.\ a \tag{A.24}$$

  We encode atoms of type `string` simply as themselves.

- integers, given $\mathtt{Lat} = \mathit{lf}$ `int`:

$$\mathtt{Lat.compile}\ i\ =\ \backslash v, v'.\ i \tag{A.25}$$

  Likewise, integers are also encoded simply as themselves.

- sets, given $\mathtt{Lat} = \mathit{lf}$ `set(`$\mathtt{T}$`)` or $\mathtt{Lat} = \mathit{lf}$ `iset(`$\mathtt{T}$`)`, $\mathtt{Lat}' = \mathit{lf}\ \mathtt{T}$:

$$\begin{array}{l}\mathtt{Lat.compile}\ \{t_1 \ldots t_n\} = \\ \backslash v, v'.\ \mathtt{FS.value.make}\ [(\mathtt{Lat}'\mathtt{.compile}\ t_1)\ v\ v' \ldots (\mathtt{Lat}'\mathtt{.compile}\ t_n)\ v\ v']\end{array} \tag{A.26}$$

  where `FS.value.make` is a Mozart/Oz function creating a finite set of integers constant from a set description, in this case, a list of integers. Sets over domains which cannot be compiled into integers are not supported.

- infinite sets of integers, given $\mathtt{Lat} = \mathit{lf}$ `set(int)`, $\mathtt{Lat} = \mathit{lf}$ `iset(int)` or $\mathtt{Lat} = \mathit{lf}$ `card`:

$$\begin{array}{l}\mathtt{Lat.compile}\ \{i_1 \ldots i_n\ \ldots\} = \\ \backslash v, v'.\ \mathtt{FS.value.make}\ [i_1 \ldots i_n \# \mathtt{FS.sup}]\end{array} \tag{A.27}$$

  where `FS.sup` denotes the greatest possible element of a set in the actual Mozart/Oz implementation, with which we approximate infinity.

- lists, given $\mathtt{Lat} = \mathit{lf}$ `list(`$\mathtt{T}$`)`, $\mathtt{Lat}' = \mathit{lf}\ \mathtt{T}$:

$$\begin{array}{l}\mathtt{Lat.compile}\ [t_1 \ldots t_n] = \\ \backslash v, v'.\ [(\mathtt{Lat}'\mathtt{.compile}\ t_1)\ v\ v' \ldots (\mathtt{Lat}'\mathtt{.compile}\ t_n)\ v\ v']\end{array} \tag{A.28}$$

- tuples (projections are exclusively finite domains), given $\texttt{Lat} = \mathit{lf}\ \texttt{tuple}(\texttt{T}_1 \ldots \texttt{T}_n)$, $\texttt{Lat}_1 = \mathit{lf}\ \texttt{T}_1$, ..., $\texttt{Lat}_n = \mathit{lf}\ \texttt{T}_n$:

$$\begin{aligned}&\texttt{Lat.compile}\ [\texttt{a}_1 \ldots \texttt{a}_n] = \\ &\backslash \texttt{v}, \texttt{v}'.\ 1 + \sum_{\texttt{i}=1}^{\texttt{n}} (((\texttt{Lat.compile}\ \texttt{a}_\texttt{i})\ \texttt{v}\ \texttt{v}') - 1) * \prod_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} (|\texttt{T}_\texttt{j}| * 1)\end{aligned} \tag{A.29}$$

  Hence, we encode tuples whose projections are exclusively finite domains into integers. This is an optimization for the constraint parser, since the Mozart/Oz constraint system can only yield propagation on integers and finite sets of integers, but not e.g. on lists. As an example, here is the encoding of the tuples in the type $\texttt{tuple}(\{\texttt{1 2 3}\}\ \{\texttt{sg pl}\})$:

$$\begin{aligned}[\texttt{1 sg}] &\mapsto 1 \\ [\texttt{1 pl}] &\mapsto 2 \\ [\texttt{2 sg}] &\mapsto 3 \\ [\texttt{2 pl}] &\mapsto 4 \\ [\texttt{3 sg}] &\mapsto 5 \\ [\texttt{3 pl}] &\mapsto 6\end{aligned} \tag{A.30}$$

- other tuples, given $\texttt{Lat} = \mathit{lf}\ \texttt{tuple}(\texttt{T}_1 \ldots \texttt{T}_n)$, $\texttt{Lat}_1 = \mathit{lf}\ \texttt{T}_1$, ... ,$\texttt{Lat}_n = \mathit{lf}\ \texttt{T}_n$:

$$\begin{aligned}&\texttt{Lat.compile}\ [\texttt{t}_1 \ldots \texttt{t}_n] = \\ &\backslash \texttt{v}, \texttt{v}'.\ [(\texttt{Lat}_1\texttt{.compile}\ \texttt{t}_1)\ \texttt{v}\ \texttt{v}' \ldots (\texttt{Lat}_n\texttt{.compile}\ \texttt{t}_n)\ \texttt{v}\ \texttt{v}']\end{aligned} \tag{A.31}$$

  Tuples whose projections are not exclusively finite domain types are encoded as lists.

- records, given $\texttt{Lat} = \mathit{lf}\ \{\texttt{a}_1 : \texttt{T}_1 \ldots \texttt{a}_n : \texttt{T}_n\}$, $\texttt{Lat}_1 = \mathit{lf}\ \texttt{T}_1$, ..., $\texttt{Lat}_n = \mathit{lf}\ \texttt{T}_n$:

$$\begin{aligned}&\texttt{Lat.compile}\ \{\texttt{a}_1 : \texttt{t}_1 \ldots \texttt{a}_n : \texttt{t}_n\} = \\ &\backslash \texttt{v}, \texttt{v}'.\ \texttt{o}(\texttt{a}_1 : (\texttt{Lat}_1\texttt{.compile}\ \texttt{t}_1)\ \texttt{v}\ \texttt{v}' \ldots \texttt{a}_n : (\texttt{Lat}_n\texttt{.compile}\ \texttt{t}_n)\ \texttt{v}\ \texttt{v}')\end{aligned} \tag{A.32}$$

  Records are encoded as Oz records with the dummy label `o`.

- feature paths, where the function `Dot` takes a record `e` and a list of attributes $\texttt{a}_1 \ldots \texttt{a}_n$, and returns the value $\texttt{e.a}_1\texttt{. ... .a}_n$:

$$\texttt{Lat.compile}\ \texttt{\_.D.entry.a}_1\texttt{. ... .a}_n = \backslash \texttt{v}, \texttt{v}'.\ \texttt{Dot}\ \texttt{v}'\texttt{.D.entry}\ [\texttt{a}_1 \ldots \texttt{a}_n] \tag{A.33}$$

$$\texttt{Lat.compile}\ \texttt{\^{}.D.entry.a}_1\texttt{. ... .a}_n = \backslash \texttt{v}, \texttt{v}'.\ \texttt{Dot}\ \texttt{v.D.entry}\ [\texttt{a}_1 \ldots \texttt{a}_n] \tag{A.34}$$

$$\texttt{Lat.compile}\ \texttt{\_.D.attrs.a}_1\texttt{. ... .a}_n = \backslash \texttt{v}, \texttt{v}'.\ \texttt{Dot}\ \texttt{v}'\texttt{.D.attrs}\ [\texttt{a}_1 \ldots \texttt{a}_n] \tag{A.35}$$

$$\texttt{Lat.compile}\ \texttt{\^{}.D.attrs.a}_1\texttt{. ... .a}_n = \backslash \texttt{v}, \texttt{v}'.\ \texttt{Dot}\ \texttt{v.D.attrs}\ [\texttt{a}_1 \ldots \texttt{a}_n] \tag{A.36}$$

  That is, we postpone the encoding of feature paths by returning a function expecting two node records `v` and $\texttt{v}'$ as arguments. When applied during parsing, the function returns the actual value of lexical or non-lexical attribute of `v` or $\texttt{v}'$.

A.2. Top, Bot and Glb

The lattice functors implement lattice top, bottom and greatest lower bound for each type of the XDK.[3] Given the lattice functor `Lat` $=$ *lf* `T`, we write:

- `Lat.top` for the top value (as a core term)
- `Lat.bot` for the bottom value (as a core term)
- `Lat.glb` $\texttt{t}_1$ $\texttt{t}_2$ for the greatest lower bound of two core terms $\texttt{t}_1$ and $\texttt{t}_2$

The purpose of lattice top is to act as the default value for the attributes omitted in a record specification, lattice bottom represents inconsistency, and the greatest lower bound of two terms represents a term which is at least as restrictive, where "at least as restrictive" is defined depending on the principle which acts on the term.

The inhabitants of finite domain, string, integer and list types are arranged in flat lattices. Sets can either be arranged in accumulative lattices, intersective lattices or cardinality lattices. Lattices for tuples and records are are defined inductively.

A.2.1. Flat Lattices

As already mentioned in Definition 29, the interpretation of each finite domain, string, int or list type includes the additional atoms $\top$ and $\bot$. We use $\top$ and $\bot$ as the top and bottom values of the lattice corresponding to the type. For example, the lattice corresponding to finite domain $\{\texttt{a}_1 \ldots \texttt{a}_\texttt{n}\}$ is displayed in Figure A.2.

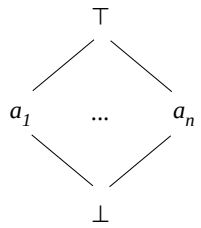


Figure A.2.: Flat lattice for finite domain $\{a_1, \ldots, a_n\}$

The top, bottom and greatest lower bound methods of flat lattices are defined as follows:

- top:

$$\texttt{Lat.top} \;=\; \top \tag{A.37}$$

- bottom:

$$\texttt{Lat.bot} \;=\; \bot \tag{A.38}$$

[3] We do not implement least upper bound since it is simply not used anywhere in the XDK.

- greatest lower bound:

$$\texttt{Lat.glb}\ \texttt{t}_1\ \texttt{t}_2 \;=\; \begin{cases} \texttt{t}_1 & \text{if } \texttt{t}_2 = \top \\ \texttt{t}_2 & \text{if } \texttt{t}_1 = \top \\ \texttt{t}_1 & \text{if } \texttt{t}_1 = \texttt{t}_2 \\ \bot & \text{otherwise} \end{cases} \tag{A.39}$$

As a practical example, the greatest lower bound of the atom `"eat"` and lattice top yields `"eat"`:

$$\texttt{Lat.glb "eat"}\ \top \;=\; \texttt{"eat"} \tag{A.40}$$

and the greatest lower bound of the atoms `"eat"` and `"want"` yields $\bot$, i.e., inconsistency:

$$\texttt{Lat.glb "eat" "want"} \;=\; \bot \tag{A.41}$$

A.2.2. Accumulative Lattices

Accumulative lattices "accumulate" their elements from top to bottom: the top value of an accumulative set lattice for type `set(T)` is the empty set, and the bottom value the full set, i.e., the interpretation of `T`. Greatest lower bound corresponds to set union. We illustrate this in Figure A.3.

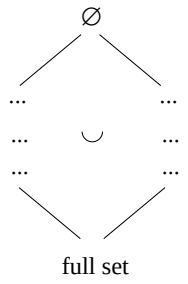


Figure A.3.: Accumulative lattice

For a lattice functor `Lat` = *lf* `set(T)`, the methods of flat lattices are defined as:

- top:

$$\texttt{Lat.top} \;=\; \{\} \tag{A.42}$$

- bottom:

$$\texttt{Lat.bot} \;=\; \texttt{t} \tag{A.43}$$

  where `t` is the interpretation of the `T`, the domain of the accumulative set.

- greatest lower bound:

$$\texttt{Lat.glb}\ \texttt{t}_1\ \texttt{t}_2 \;=\; \texttt{t}_1 \cup \texttt{t}_2 \tag{A.44}$$

Accumulative lattices are convenient e.g. for the attribute `agree` of the Agreement principle, which represents the set of edge labels describing with which daughters the node must agree. The more elements this set has, the more restrictive it becomes.

A.2.3. Intersective Lattices

Intersective lattices are exactly the mirror image of accumulative lattice: their top value is the full set, their bottom value the empty set, and greatest lower bound corresponds to intersection. Figure A.4 illustrates this.

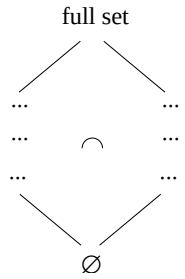


Figure A.4.: Intersective lattice

For a lattice functor $\texttt{Lat} = \mathit{lf}\ \texttt{iset}(\texttt{T})$, the methods of flat lattices are defined as:

- top:
$$\texttt{Lat.top} \;=\; \texttt{t} \tag{A.45}$$
where $\texttt{t}$ is the interpretation of the $\texttt{T}$, the domain of the accumulative set.
- bottom:
$$\texttt{Lat.bot} \;=\; \{\} \tag{A.46}$$
- greatest lower bound:
$$\texttt{Lat.glb}\ \texttt{t}_1\ \texttt{t}_2 \;=\; \texttt{t}_1 \cap \texttt{t}_2 \tag{A.47}$$

Intersective lattices are useful e.g. for the `agrs` attribute of the Agreement principle, which represents the sets of agreement tuples of a node. Contrary to the sets of the `agree` attribute, which became more restrictive the more elements they contained, sets of agreements become more restrictive the less elements they contain.

A.2.4. Cardinality Lattices

The lattice operations of *cardinality lattices* are illustrated in Figure A.5: top is defined as the set $\{0\}$, bottom as the empty set, and greatest lower bound as set intersection (except when one of the arguments is top).

Given $\texttt{Lat} = \mathit{lf}\ \texttt{card}$, the lattice functor methods are defined as:

- top:
$$\texttt{Lat.top} \;=\; \{0\} \tag{A.48}$$
- bottom:
$$\texttt{Lat.bot} \;=\; \{\} \tag{A.49}$$

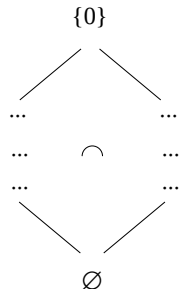


Figure A.5.: Cardinality lattice

- greatest lower bound:

$$\texttt{Lat.glb}\ \texttt{t}_1\ \texttt{t}_2 \;=\; \begin{cases} \texttt{t}_1 & \text{if } \texttt{t}_2 = \{\texttt{0}\} \\ \texttt{t}_2 & \text{if } \texttt{t}_1 = \{\texttt{0}\} \\ \texttt{t}_1 & \text{if } \texttt{t}_1 = \texttt{t}_2 \\ \texttt{t}_1 \cap \texttt{t}_2 & \text{otherwise} \end{cases} \tag{A.50}$$

Cardinality lattices are specifically designed for the *Valency principle* using valency types, i.e., vectors used to map edge labels to cardinalities. Generally, cardinalities become more restrictive the less elements they contain, e.g. the set $\{\texttt{1}\}$ licensing precisely one edge is more restrictive than the set $\{\texttt{0 1}\}$ licensing zero or one edges. That is, generally, the greatest lower bound of two cardinalities is their intersection. But then, why can we not simply model them using intersective lattices? The motivation for introducing a new lattice is that we want lattice top of cardinalities not to be the set of all integers but the set $\{\texttt{0}\}$, because this gives us the intuitive interpretation of valencies that if a cardinality is missing for an edge label, no edge with that label is allowed: as valencies are interpreted as record specifications, all omitted edge labels are automatically set to lattice top of the cardinality lattice, i.e., $\{\texttt{0}\}$.

A.2.5. Tuple Lattices

We define the lattices for tuples inductively. Given a tuple lattice functor `Lat` defined as:

$$\texttt{Lat} \;=\; \mathit{lf}\ \texttt{tuple}(\texttt{T}_1 \ldots \texttt{T}_n) \tag{A.51}$$

with lattice functors $\texttt{Lat}_1 \ldots \texttt{Lat}_n$ for its projections defined as:

$$\begin{aligned} \texttt{Lat}_1 &= \mathit{lf}\ \texttt{T}_1 \\ &\ldots \\ \texttt{Lat}_n &= \mathit{lf}\ \texttt{T}_n \end{aligned} \tag{A.52}$$

we define:

$$\texttt{Lat.top} \;=\; [\texttt{Lat}_1\texttt{.top} \ldots \texttt{Lat}_n\texttt{.top}] \tag{A.53}$$

$$\texttt{Lat.bot} \;=\; [\texttt{Lat}_1\texttt{.bot} \ldots \texttt{Lat}_n\texttt{.bot}] \tag{A.54}$$

$$\texttt{Lat.glb}\ [\texttt{t}_1 \ldots \texttt{t}_n]\ [\texttt{t}'_1 \ldots \texttt{t}'_n] \;=\; [\texttt{Lat}_1\texttt{.glb}\ \texttt{t}_1\ \texttt{t}'_1 \ldots \texttt{Lat}_n\texttt{.glb}\ \texttt{t}_n\ \texttt{t}'_n] \tag{A.55}$$

A.2.6. Record Lattices

Record lattices are defined analogously to tuple lattices. Given a record lattice functor Lat defined as:

$$\texttt{Lat} \;=\; \mathit{lf}\ \{\texttt{a}_1 : \texttt{T}_1 \ldots \texttt{a}_n : \texttt{T}_n\} \tag{A.56}$$

with lattice functors $\texttt{Lat}_1 \ldots \texttt{Lat}_n$ for attributes, we define:

$$\texttt{Lat.top} \;=\; \{\texttt{a}_1 : \texttt{Lat}_1\texttt{.top} \ldots \texttt{a}_n : \texttt{Lat}_n\texttt{.top}\} \tag{A.57}$$

$$\texttt{Lat.bot} \;=\; \{\texttt{a}_1 : \texttt{Lat}_1\texttt{.bot} \ldots \texttt{a}_n : \texttt{Lat}_n\texttt{.bot}\} \tag{A.58}$$

$$\texttt{Lat.glb}\ \{\texttt{a}_1 : \texttt{t}_1 \ldots \texttt{a}_n : \texttt{t}_n\}\ \{\texttt{a}_1 : \texttt{t}'_1 \ldots \texttt{a}_n : \texttt{t}'_n\} = \{\texttt{a}_1 : \texttt{Lat}_1\texttt{.glb}\ \texttt{t}_1\ \texttt{t}'_1 \ldots \texttt{a}_n : \texttt{Lat}_n\texttt{.glb}\ \texttt{t}_n\ \texttt{t}'_n\} \tag{A.59}$$

A.3. Constraint Variable Creation, Lexical Selection

For the constraint parser, the lattice functors implement the two methods makeVar for the creation of constraint variables, and select for the selection of values from a set of alternatives.

A.3.1. MakeVar

- atoms from a finite domain, given $\texttt{Lat} = \mathit{lf}\ \{\texttt{a}_1 \ldots \texttt{a}_n\}$:

  $$\texttt{Lat.makeVar} \;=\; \texttt{FD.int 1\#n} \tag{A.60}$$

  where FD.int is a Mozart/Oz function creating a *finite domain constraint variable* from a specification of a finite domain. Here, the finite domain ranges from 1 to n.

- atoms of type string, given $\texttt{Lat} = \mathit{lf}\ \texttt{string}$:

  $$\texttt{Lat.makeVar} \;=\; \_ \tag{A.61}$$

  where _ creates a *logic variable* in Mozart/Oz.

- integers, given $\texttt{Lat} = \mathit{lf}\ \texttt{int}$:

  $$\texttt{Lat.makeVar} \;=\; \texttt{FD.int 1\#FD.sup} \tag{A.62}$$

  where FD.sup is the greatest natural number for integers in Mozart/Oz.

- sets, given $\texttt{Lat} = \mathit{lf}\ \texttt{set}(\texttt{T})$ or $\texttt{Lat} = \mathit{lf}\ \texttt{iset}(\texttt{T})$:

  $$\texttt{Lat.makeVar} \;=\; \texttt{FS.var.upperBound 1\#n} \tag{A.63}$$

  where n is the cardinality of the interpretation of T, and where FS.var.upperBound creates a *finite set constraint variable* from a specification of its upper bound, i.e., the set including its potential elements (here: $\{1, \ldots, \texttt{n}\}$).

- infinite sets of integers, given $\texttt{Lat} = \mathit{lf}\ \texttt{set(int)}$, $\texttt{Lat} = \mathit{lf}\ \texttt{iset(int)}$ or $\texttt{Lat} = \mathit{lf}\ \texttt{card}$:

$$\texttt{Lat.makeVar} \quad = \quad \texttt{FS.var.upperBound 1\#FS.sup} \tag{A.64}$$

- list: $\texttt{Lat} = \mathit{lf}\ \texttt{list(T)}$, $\texttt{Lat}' = \mathit{lf}\ \texttt{T}$:

$$\texttt{Lat.makeVar} \quad = \quad [\texttt{Lat}'\texttt{.makeVar} \ldots \texttt{Lat}'\texttt{.makeVar}] \tag{A.65}$$

- tuples (projections are exclusively finite domains), given $\texttt{Lat} = \mathit{lf}\ \texttt{tuple(}\texttt{T}_1 \ldots \texttt{T}_n\texttt{)}$, $\texttt{Lat}_1 = \mathit{lf}\ \texttt{T}_1$, ..., $\texttt{Lat}_n = \mathit{lf}\ \texttt{T}_n$:

$$\texttt{Lat.makeVar} \quad = \quad \texttt{FD.int 1\#}(\prod_{i=1}^{n} |\texttt{T}_i|) \tag{A.66}$$

  where $\prod_{i=1}^{n} |\texttt{T}_i|$ is the cardinality of $\texttt{T}$.

- other tuples, given $\texttt{Lat} = \mathit{lf}\ \texttt{tuple(}\texttt{T}_1 \ldots \texttt{T}_n\texttt{)}$, $\texttt{Lat}_1 = \mathit{lf}\ \texttt{T}_1$, ..., $\texttt{Lat}_n = \mathit{lf}\ \texttt{T}_n$:

$$\texttt{Lat.makeVar} \quad = \quad [\texttt{Lat}_1\texttt{.makeVar} \ldots \texttt{Lat}_n\texttt{.makeVar}] \tag{A.67}$$

- records, given $\texttt{Lat} = \mathit{lf}\ \{\texttt{a}_1 : \texttt{T}_1 \ldots \texttt{a}_n : \texttt{T}_n\}$, $\texttt{Lat}_1 = \mathit{lf}\ \texttt{T}_1$, ..., $\texttt{Lat}_n = \mathit{lf}\ \texttt{T}_n$:

$$\texttt{Lat.makeVar} \quad = \quad \texttt{o}(\texttt{a}_1 : \texttt{Lat}_1\texttt{.makeVar} \ldots \texttt{a}_n : \texttt{Lat}_n\texttt{.makeVar}) \tag{A.68}$$

A.3.2. Select

- atoms from a finite domain, given $\texttt{Lat} = \mathit{lf}\ \{\texttt{a}_1 \ldots \texttt{a}_n\}$:

$$\texttt{Lat.select}\ [\texttt{i}_1 \ldots \texttt{i}_n]\ \texttt{i} \quad = \quad \texttt{Select.fd}\ [\texttt{i}_1 \ldots \texttt{i}_n]\ \texttt{i} \tag{A.69}$$

  where $\texttt{i}_1, \ldots, \texttt{i}_n$ are integers encoding the finite domain elements $\texttt{a}_1 \ldots \texttt{a}_n$, and where `Select.fd` is the *selection constraint* (Duchier 1999, Duchier 2003) for finite domain constraint variables. Its declarative semantics is to select the `i`th element of a list $[\texttt{i}_1 \ldots \texttt{i}_n]$ of finite domain constraint variables. During constraint solving, the selector `i` is often underspecified. In this case, the selection constraint significantly improves constraint propagation, as all commonalities of the remaining alternatives are immediately propagated to the selected value.

- atoms of type `string`, lists, given $\texttt{Lat} = \mathit{lf}\ \texttt{string}$ or $\texttt{Lat} = \mathit{lf}\ \texttt{list(T)}$:

$$\texttt{Lat.select}\ [\texttt{s}_1 \ldots \texttt{s}_n]\ \texttt{i} \quad = \quad s_{\texttt{Select.fd}\ [\texttt{i}_1 \ldots \texttt{i}_n]\ \texttt{i}} \tag{A.70}$$

  where $\texttt{s}_1, \ldots, \texttt{s}_n$ are SL strings or lists.

- integers:

$$\texttt{Lat.select}\ [\texttt{i}_1 \ldots \texttt{i}_n]\ \texttt{i} \quad = \quad \texttt{Select.fd}\ [\texttt{i}_1 \ldots \texttt{i}_n]\ \texttt{i} \tag{A.71}$$

- sets, infinite sets of integers:

$$\texttt{Lat.select}\ [\texttt{s}_1 \ldots \texttt{s}_n]\ \texttt{i} \ = \ \texttt{Select.fs}\ [\texttt{s}_1 \ldots \texttt{s}_n]\ \texttt{i} \tag{A.72}$$

  where `Select.fs` is the selection constraint for finite set constraint variables.

- tuples (projections are exclusively finite domains): $\texttt{Lat} = \textit{lf}\ \texttt{tuple}(\texttt{T}_1 \ldots \texttt{T}_n)$, $\texttt{Lat}_1 = \textit{lf}\ \texttt{T}_1$, ..., $\texttt{Lat}_n = \textit{lf}\ \texttt{T}_n$, and $\texttt{T}_1 = \{\ldots\}$, ..., $\texttt{T}_n = \{\ldots\}$:

$$\texttt{Lat.select}\ [\texttt{i}_1 \ldots \texttt{i}_n]\ \texttt{i} \ = \ \texttt{Select.fd}\ [\texttt{i}_1 \ldots \texttt{i}_n]\ \texttt{i} \tag{A.73}$$

  Hence, for tuples whose projections are exclusively finite domains, we can use the selection constraint for finite domain constraint variables. This yields much better propagation than if we had not encoded such tuples as integers in the compilation step above.

- other tuples, given $\texttt{Lat} = \textit{lf}\ \texttt{tuple}(\texttt{T}_1 \ldots \texttt{T}_n)$, $\texttt{Lat}_1 = \textit{lf}\ \texttt{T}_1$, ..., $\texttt{Lat}_n = \textit{lf}\ \texttt{T}_n$:

$$\begin{array}{l}\texttt{Lat.select}\ [[\texttt{s}_1^1, \ldots, \texttt{s}_n^1] \ldots [\texttt{s}_1^k, \ldots, \texttt{s}_n^k]]\ \texttt{i} = \\ [\texttt{Lat}_1\texttt{.select}\ [\texttt{s}_1^1 \ldots \texttt{s}_1^k]\ \texttt{i} \ldots \texttt{Lat}_1\texttt{.select}\ [\texttt{s}_n^1 \ldots \texttt{s}_n^k]\ \texttt{i}]\end{array} \tag{A.74}$$

- records, given $\texttt{Lat} = \textit{lf}\ \{\texttt{a}_1 : \texttt{T}_1 \ldots \texttt{a}_n : \texttt{T}_n\}$, $\texttt{Lat}_1 = \textit{lf}\ \texttt{T}_1$, ..., $\texttt{Lat}_n = \textit{lf}\ \texttt{T}_n$:

$$\begin{array}{l}\texttt{Lat.select}\ [\texttt{o}(\texttt{a}_1 : \texttt{s}_1^1, \ldots, \texttt{a}_n : \texttt{s}_n^1) \ldots \texttt{o}(\texttt{a}_1 : \texttt{s}_1^k, \ldots, \texttt{a}_n : \texttt{s}_n^k)]\ \texttt{i} = \\ \texttt{o}(\texttt{a}_1 : \texttt{Lat}_1\texttt{.select}\ [\texttt{s}_1^1 \ldots \texttt{s}_1^k]\ \texttt{i} \ldots \texttt{a}_n : \texttt{Lat}_1\texttt{.select}\ [\texttt{s}_n^1 \ldots \texttt{s}_n^k]\ \texttt{i})\end{array} \tag{A.75}$$

A.4. Decode and Pretty

The `decode` method decodes SL terms back into IL syntax, whereas the `pretty` decodes SL into OL syntax for pretty printing. For sets of tuples whose projections are exclusively finite domains into OL syntax, the conversion of SL into OL involves a function to convert sets using *set generators*.

A.5. Summary

We introduced the lattice functors of the XDK, which provide functionality for all modules of the XDK, i.e, the metagrammar compiler, the constraint parser and the visualizer. For the metagrammar compiler, it provides methods for encoding metagrammars into Mozart/Oz syntax. For the constraint parser, it provides methods for constraint variable creation and selection, and for the visualizer, methods for decoding and pretty printing of analyses.

B. Metagrammar Compiler

This appendix deals with the *metagrammar compiler* of the XDK. Metagrammars are descriptions of grammars. The task of the metagrammar compiler of the XDK is to compile these descriptions into actual grammars usable in the constraint solver. Figure B.1 shows the position of the metagrammar compiler within the overall architecture of the XDK.

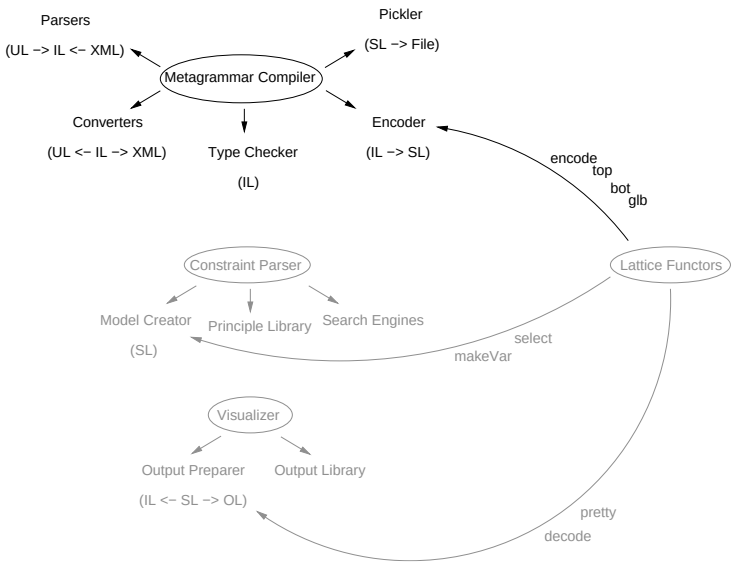


Figure B.1.: The metagrammar compiler in the XDK architecture

Metagrammar compilation starts from a metagrammar in one of three concrete syntaxes: UL, XML or IL, and proceeds in four steps:

1. parsing the metagrammar if it is in UL or XML syntax
2. converting the parsed metagrammar into IL
3. type checking the IL metagrammar
4. compiling out the IL metagrammar and encoding it into the SL

Encoded grammars can then either be pickled, i.e., written into files, or used for constraint parsing. The metagrammar compiler is assisted by the `encode` method and the methods implementing the lattice operations (`top`, `bot` and `glb`) of the *lattice functors*.

B.1. Parsers and Converters

The task of the parsers and converters is to bring grammars from UL or XML into IL syntax. Because of the modular interface of the parsers and converters to the XDK, adding new concrete syntaxes is easy. The converters are bi-directional: i.e., also able to convert grammars from IL back to UL or XML. As a result, via the IL, the XDK also supports conversion of UL into XML and vice versa. For parsing UL metagrammars, we use an efficient LALR parser written in Mozart/Oz by Denys Duchier. For parsing XML metagrammars, we apply the efficient XML parser from the Mozart/Oz Standard Library, also written by Denys Duchier.

B.2. Type Checker

In this section, we define the type checker for the full set of terms of the XDK description language. It is defined in terms of inference rules for type judgments (e.g. Pierce 2002). We write

$$\Gamma \vdash \mathtt{t} : \mathtt{T} \tag{B.1}$$

for the type judgment stating that term $\mathtt{t}$ has type $\mathtt{T}$ under the environment Γ. Γ is a set containing the following four functions:

- $\Gamma_v : \mathtt{V} \rightarrow \mathtt{Te}$ maps variables to terms. The mapping is defined upon reference of lexical classes in the lexicon description.
- $\Gamma_d : \mathtt{DV} \rightarrow \mathtt{D}$ maps *dimension variables* to dimensions in $\mathtt{D}$. The mapping is defined upon principle instantiation.
- $\Gamma_p : \mathtt{D} \rightarrow \{\mathtt{entry}, \mathtt{attrs}\} \rightarrow \mathtt{A}^* \rightarrow \mathtt{Ty}$ returns the type of the feature path $\mathtt{a}_1, \ldots, \mathtt{a}_n \in \mathtt{A}^*$ starting from the lexical (`entry`) or non-lexical (`attrs`) attributes of dimension $\mathtt{d} \in \mathtt{D}$. The types of the lexical and non-lexical attributes of a dimension are provided by the type definitions of the metagrammar (`defentrytype` and `defattrstype`.
- $\Gamma_t : \mathtt{TV} \rightarrow \mathtt{Ty}$ maps type variables to types.

The type checker is used for both lexicon description and principle instantiations.

The inference rules of the type checker are defined as follows:

- atoms from a finite domain:

$$\frac{\mathtt{a} \in \{\mathtt{a}_1 \ldots \mathtt{a}_n\}}{\Gamma \vdash \mathtt{a} : \{\mathtt{a}_1 \ldots \mathtt{a}_n\}} \tag{B.2}$$

- atoms of type `string`, given a set $\mathtt{A}$ of atoms:

$$\frac{\mathtt{a} \in \mathtt{A}}{\Gamma \vdash \mathtt{a} : \mathtt{string}} \tag{B.3}$$

- integers:

$$\frac{\mathtt{i} > 0}{\Gamma \vdash \mathtt{i} : \mathtt{int}} \tag{B.4}$$

The XDK supports only natural numbers. This allows us to encode all numbers as finite domain integers in Mozart/Oz, which must be greater than zero.

- variables:

$$\frac{\Gamma_v\, \mathtt{v} = \mathtt{t} \quad \Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \mathtt{v} : \mathtt{T}} \tag{B.5}$$

Variables can only be used inside lexical classes, not in principle instantiations. They are instantiated upon reference of the lexical classes.

- sets:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \quad \ldots \quad \Gamma \vdash \mathtt{t}_n : \mathtt{T}}{\Gamma \vdash \{\mathtt{t}_1 \ldots \mathtt{t}_n\} : \mathtt{set(T)}} \tag{B.6}$$

The rule for `isets` is defined analogously.

- infinite sets of integers:

$$\frac{\Gamma \vdash \mathtt{i}_1 : \mathtt{int} \quad \ldots \quad \Gamma \vdash \mathtt{i}_n : \mathtt{int}}{\Gamma \vdash \{\mathtt{i}_1 \ldots \mathtt{i}_n \ \ldots\} : \mathtt{set(int)}} \tag{B.7}$$

The rules for `iset(int)` and `card` are defined analogously.

- lists:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \quad \ldots \quad \Gamma \vdash \mathtt{t}_n : \mathtt{T}}{\Gamma \vdash [\mathtt{t}_1 \ldots \mathtt{t}_n] : \mathtt{list(T)}} \tag{B.8}$$

- tuples:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \quad \ldots \quad \Gamma \vdash \mathtt{t}_n : \mathtt{T}_n}{\Gamma \vdash [\mathtt{t}_1 \ldots \mathtt{t}_n] : \mathtt{tuple(T_1 \ldots T_n)}} \tag{B.9}$$

- record specifications and empty records:

$$\frac{\begin{array}{c}\{\mathtt{a}'_1, \ldots, \mathtt{a}'_k\} \subseteq \{\mathtt{a}_1, \ldots, \mathtt{a}_n\} \\ \Gamma \vdash \mathtt{t}_j : \mathtt{T}_i \text{ if } \mathtt{a}'_j = \mathtt{a}_i \\ 1 \leq \mathtt{j} \leq \mathtt{k}, \quad 1 \leq \mathtt{i} \leq \mathtt{n}\end{array}}{\Gamma \vdash \{\mathtt{a}'_1 : \mathtt{t}_1 \ldots \mathtt{a}'_k : \mathtt{t}_k\} : \{\mathtt{a}_1 : \mathtt{T}_1 \ldots \mathtt{a}_n : \mathtt{T}_n\}} \tag{B.10}$$

In record specifications, any number of attributes can be omitted. In order to be well-typed, the attributes of a record specification must be a subset of the full set of attributes of its record type, and the value of each given attribute must have the appropriate type.

- cardinalities:

$$\frac{\mathtt{c} \in \{!, ?, *, +\}}{\Gamma \vdash \mathtt{c} : \mathtt{card}} \tag{B.11}$$

$$\frac{\Gamma \vdash \mathtt{i}_1 : \mathtt{int} \quad \ldots \quad \Gamma \vdash \mathtt{i}_n : \mathtt{int}}{\Gamma \vdash \#\{\mathtt{i}_1 \ldots \mathtt{i}_n\} : \mathtt{card}} \tag{B.12}$$

$$\frac{\Gamma \vdash \mathtt{i}_1 : \mathtt{int} \quad \Gamma \vdash \mathtt{i}_2 : \mathtt{int}}{\Gamma \vdash \#[\mathtt{i}_1\ \mathtt{i}_2] : \mathtt{card}} \tag{B.13}$$

- valencies:

$$\frac{\Gamma \vdash c_1 : \texttt{card} \quad \ldots \quad \Gamma \vdash c_n : \texttt{card}}{\Gamma \vdash \{a_1\ c_1 \ldots a_n\ c_n\} : \texttt{valency}(\{a_1, \ldots a_n\})} \tag{B.14}$$

- lattice tops:

$$\frac{}{\Gamma \vdash \texttt{top} : \mathtt{T}} \tag{B.15}$$

- lattice bottoms:

$$\frac{}{\Gamma \vdash \texttt{bot} : \mathtt{T}} \tag{B.16}$$

- lattice greatest lower bounds:

$$\frac{\Gamma \vdash t_1 : \mathtt{T} \quad \Gamma \vdash t_2 : \mathtt{T}}{\Gamma \vdash t_1 \,\&\, t_2 : \mathtt{T}} \tag{B.17}$$

- alternations:

$$\frac{\Gamma \vdash t_1 : \mathtt{T} \quad \Gamma \vdash t_2 : \mathtt{T}}{\Gamma \vdash t_1 \mid t_2 : \mathtt{T}} \tag{B.18}$$

- set generators:

$$\frac{\mathtt{T}_1 = \{a_1, \ldots, a_n\} \qquad \mathtt{T}_n = \{a'_1, \ldots, a'_m\} \quad \mathit{atoms}\ g \subseteq \mathtt{T}_1 \uplus \ldots \uplus \mathtt{T}_n}{\Gamma \vdash \$\, g : \texttt{set}(\texttt{tuple}(\mathtt{T}_1 \ldots \mathtt{T}_n))} \tag{B.19}$$

  where *atoms* g returns the set of atoms in set generator g. That is, the types $\mathtt{T}_1, \ldots, \mathtt{T}_n$ must all be finite domains, and the atoms occurring in g must be a subset of their atoms. In addition, $\mathtt{T}_1, \ldots, \mathtt{T}_n$ must be disjoint to avoid ambiguous set generators.

  The rule for `isets` is defined analogously.

- orders:

$$\frac{a_1 : \mathtt{T} \quad \ldots \quad a_n : \mathtt{T}}{\Gamma \vdash \langle a_1 \ldots a_n \rangle : \texttt{set}(\texttt{tuple}(\mathtt{T}\ \mathtt{T}))} \tag{B.20}$$

  The type of an order must be a set whose domain is a pair of type T.

  The rule for `isets` is defined analogously.

- concatenations:

$$\frac{\Gamma \vdash t_1 : \texttt{string} \quad \Gamma \vdash t_2 : \texttt{string}}{\Gamma \vdash t_1 @ t_2 : \texttt{string}} \tag{B.21}$$

- feature paths:

$$\frac{\Gamma_p\ (\Gamma_d\ \mathtt{D})\ \texttt{entry}\ a_1, \ldots, a_n = \mathtt{T}}{\Gamma \vdash \_.\mathtt{D}.\texttt{entry}.a_1. \ \ldots \ .a_n : \mathtt{T}} \tag{B.22}$$

  The type of a feature path can be inferred from the metagrammar type definitions.

  The rules for the other feature paths are defined analogously.

- type annotations:

$$\frac{\Gamma \vdash t : \mathtt{T}}{\Gamma \vdash (t :: \mathtt{T}) : \mathtt{T}} \tag{B.23}$$

- type variables:

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma, \Gamma_t \cup \{\mathtt{X} \mapsto \mathtt{T}\} \vdash \mathtt{t} : \mathtt{tv}(\mathtt{X})} \tag{B.24}$$

  i.e., when we can prove that term `t` has type `T`, we can instantiate the type variable (`X`) with type `T`.

B.3. Encoder

The encoder compiles the metagrammar into SL syntax for the constraint solver. To this end, it uses the encode method of the lattice functors (cf. section A.1). After encoding, each resulting lexical entry is checked for integrity, i.e.:

- it must define the `lex` dimension
- it must define the `word` attribute on the `lex` dimension
- no finite domain, string, int or list may remain undefined ($\top$) or may have become inconsistent ($\bot$)

B.4. Pickler

The task of the pickler is to write compiled out SL grammars into files called *pickles*. Before pickling, all stateful values, i.e., lattice functors, dynamically linked principles from the principle library and outputs from the output library, must be transformed into stateless values. The largest part of a typical SL grammar, the lexicon, can then be written in two ways:

- as a Mozart/Oz record
- into a database, using the Mozart/Oz GNU GDBM interface

The former is more compatible across platforms than the latter: e.g., the GNU GDBM library is only standardly installed on Unix-ish platforms but not on Microsoft Windows. Grammars written as a Mozart/Oz record are also more compact than those using the GNU GDBM interface. The big advantage of the latter is however the significantly more efficient treatment of large lexicons.

B.5. Runtime

We compiled the handcrafted metagrammars `diss.ul` (part III of this thesis), `Diplom.ul` (Debusmann 2001), `softproj.ul` (Bader et al. 2004), and a large metagrammar automatically generated by the system described in (Bojar 2004) on an AMD Athlon with 1.2 GHz and 512 MBytes of RAM. The runtimes include all components of the metagrammar compiler,

i.e., the parsers, converters, the type checker, encoder and the pickler. As can be seen, the metagrammar compiler is fast:

Name	Length (KB)	Entries	Time (s)
`Diplom.ul`	30414	190	2.29
`diss.ul`	51587	122	5.4
`softproj.ul`	90066	423	6.85
`test.1.chunk1.xdk.xml`	8561336	492	27.9

(B.25)

B.6. Summary

We described the metagrammar compiler of the XDK, comprising parsers and converters for metagrammars, a static type checker, an encoder and a pickler. We defined the type checker using inference rules. For the encoder, we could simply make use of the encode method of the lattice functors defined in appendix A. The result of encoding is a grammar in SL syntax suitable for the constraint parser.

C. Visualizer

This appendix explains the *visualizer* of the XDK, whose purpose is to visualize the solutions of the constraint parser. Figure C.1 displays the position of the visualizer in the overall architecture of the XDK.

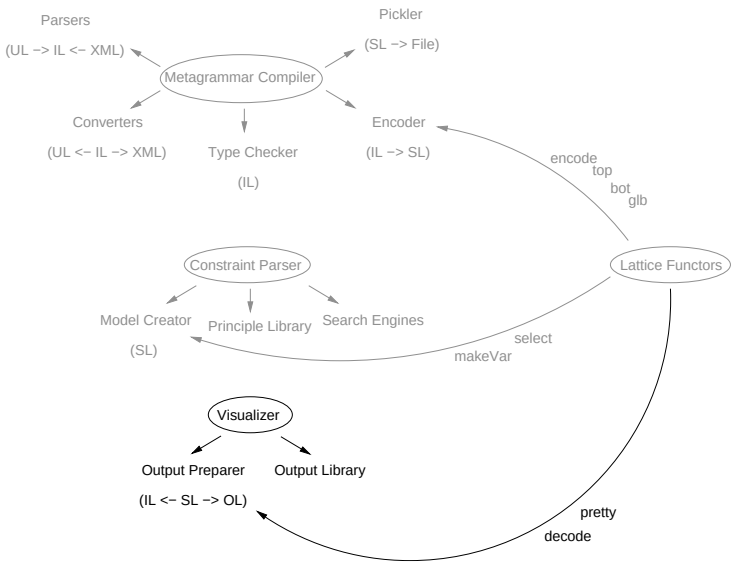


Figure C.1.: The metagrammar compiler in the XDK architecture

C.1. Output Preparer

The purpose of the *output preparer* is to prepare (possibly partial) solutions for visualization by the *output functors* of the XDK *output library* by:

1. decoding the solution from SL into IL and *Output Language* (*OL*) syntax
2. creating an *edge record* representing the determined edges and dominance relationships of the (possibly partial) solution

As a result, the output preparer hides the internal representation of the nodes in the constraint parser (see section 8.1.1) from the developers of the output functors, making their task considerably easier.

C.1.1. Decoding

Decoding is done using the `decode` and `pretty` methods of the lattice functors described in appendix A, where the OL syntax is a blend of UL and IL syntax: contrary to the UL, it uses Oz syntax for better integration with the Mozart/Oz output tools, but it is at the same time more readable than the IL in order to ease debugging. We display an example analysis in IL syntax in Figure C.2, and in OL syntax in Figure C.3.

```
1#
[o(args:[elem(data:entryIndex tag:constant)#elem(data:1 tag:integer)
         elem(data:index tag:constant)#elem(data:1 tag:integer)
         elem(data:word tag:constant)#elem(data:every tag:constant)
         elem(data:sc tag:constant)#
         elem(
            args:[elem(data:attrs tag:constant)#elem(args:nil tag:record)
                  elem(data:entry tag:constant)#
                  elem(
                     args:[elem(data:'in' tag:constant)#
                           elem(
                              args:[elem(data:del tag:constant)#elem(args:nil tag:'card.set')
                                    elem(data:q tag:constant)#elem(arg:'?' tag:'card.wild')
                                    elem(data:r tag:constant)#elem(args:nil tag:'card.set')
                                    elem(data:root tag:constant)#elem(args:nil tag:'card.set')
                                    elem(data:s tag:constant)#elem(args:nil tag:'card.set')]
                              tag:record)
                           elem(data:out tag:constant)#
                           elem(
                              args:[elem(data:del tag:constant)#elem(args:nil tag:'card.set')
                                    elem(data:q tag:constant)#elem(args:nil tag:'card.set')
                                    elem(data:r tag:constant)#elem(args:nil tag:'card.set')
                                    elem(data:root tag:constant)#elem(args:nil tag:'card.set')
                                    elem(data:s tag:constant)#elem(args:nil tag:'card.set')]
                              tag:record)]
                     tag:record)
                  elem(data:model tag:constant)#
                  elem(
                     args:[elem(data:daughters tag:constant)#elem(args:nil tag:set)
                           elem(data:daughtersL tag:constant)#
                           elem(
                              args:[elem(data:del tag:constant)#elem(args:nil tag:set)
                                    elem(data:q tag:constant)#elem(args:nil tag:set)
                                    elem(data:r tag:constant)#elem(args:nil tag:set)
                                    elem(data:root tag:constant)#elem(args:nil tag:set)
                                    elem(data:s tag:constant)#elem(args:nil tag:set)]
                              tag:record)
                           elem(data:down tag:constant)#elem(args:nil tag:set)
                           elem(data:downL tag:constant)#
                           elem(
                              args:[elem(data:del tag:constant)#elem(args:nil tag:set)
                                    elem(data:q tag:constant)#elem(args:nil tag:set)
                                    elem(data:r tag:constant)#elem(args:nil tag:set)
                                    elem(data:root tag:constant)#elem(args:nil tag:set)
                                    elem(data:s tag:constant)#elem(args:nil tag:set)]
                              tag:record)
                           elem(data:eq tag:constant)#
                           elem(args:[elem(data:1 tag:integer)] tag:set)
                           elem(data:eqdown tag:constant)#
                           elem(args:[elem(data:1 tag:integer)] tag:set)
                           elem(data:equp tag:constant)#
                           elem(args:[[1#2 6] [1#2 5#6] 3#4] tag:'_')
```

Figure C.2.: Analysis in IL syntax (first node, partial)

C.1.2. Edge Record Creation

In this step, the output preparer creates a record called *edge record* containing for all dimensions of the multigraph, the edges and *dominance edges* determined by the constraint parser for the (possibly partial) solution.

```
1#
[o(entryIndex:1
   index:1
   sc:o(attrs:top
        entry:o('in':o(q:'?'))
        model:o(eq:[1]
                eqdown:[1]
                equp:'_'([1#2 6] [1#2 5#6] 3#4)
                index:1
                labels:[q]
                mothers:[2]
                mothersL:o(q:[2])
                up:'_'([2 6] [2 5#6] 2#3)
                upL:o(q:'_'([2 6] [2 5#6] 2#3))))
   word:every)
 o(entryIndex:1
   index:2
   sc:o(attrs:top
        entry:o('in':o(r:'?' root:'?' s:'?') out:o(q:'!' r:'*' s:'!'))
        model:o(daughters:'_'([1] [1 3 5] 2)
                daughtersL:o(q:[1] s:'_'(nil [3 5] 1))
                down:'_'([1 3] [1 3#5] 2#4)
                downL:o(q:[1] s:'_'([3] [3#5] 1#3))
                eq:[2]
                eqdown:'_'([1#3] [1#5] 3#5)
                equp:'_'([2 6] [2 5#6] 2#3)
                index:2
                labels:'_'(nil [root#s] 1)
                mothers:'_'(nil [5#6] 1)
                mothersL:o(root:'_'(nil [6] 0#1) s:'_'(nil [5] 0#1))
                up:'_'([6] [5#6] 1#2)
                upL:o(root:'_'(nil [6] 0#1) s:'_'(nil [5#6] 0#2))))
   word:man)
 o(entryIndex:9
   index:3
   sc:o(attrs:top
        entry:o('in':o(r:'?' root:'?' s:'?'))
        model:o(eq:[3]
                eqdown:[3]
                equp:[2 3 5 6]
                index:3
                labels:[s]
                mothers:'_'(nil [2 5] 1)
                mothersL:o(s:'_'(nil [2 5] 1))
                up:[2 5 6]
                upL:o(s:[2 5 6])))
   word:loves)
```

Figure C.3.: Analysis in OL syntax (first three nodes)

Edges. For each dimension, the edge record contains three kinds of edges, which are represented in the following lists:

1. `Edges`: the list of records `edge(I1 I2)` representing the determined edges from node index `I1` to node index `I2`. We obtain this information from the lower bounds of the `daughters` sets of the nodes. For example, consider the underspecified analysis displayed in Figure C.3, where the `daughters` set of the node with index `2` is defined as:

   ```
   daughters:'_'([1] [1 3 5] 2)
   ```
   (C.1)

   which indicates (by the underscore) that this set is not yet fully determined. What the constraint solver already knows about this set is that:

   a) the list of elements which the set is already known to include (its lower bound) is $[1]$, i.e., it includes at least the node `1`

   b) the list of elements which the set may still include (its upper bound) includes `1`, `3` and `5`

   c) the cardinality of the set is already determined to be `2`

That is, from the lower bound, we can infer the following edge and add it to the list of already determined edges:

```
edge(2 1)
```
(C.2)

The upper bound and the cardinality are not considered.

2. `LEdges`: the list of records `edge(I1 I2 LA)` of the determined labeled edges from `I1` to `I2` labeled `LA`, obtained from the lower bounds of the `daughtersL` sets of the nodes. For example, in Figure C.3, the `daughtersL` sets for node `2` indicate that node `1` is already known to be the `q` daughter and that the set of daughters with edge label `s` has cardinality `1`, i.e., it contains precisely one node, which is either `3` or `5`:

```
daughtersL:o(q:[1] s:'_'(nil [3 5] 1))
```
(C.3)

Since the upper bounds and the cardinalities of the sets are not considered, this only allows us to add the following labeled edge to the list of already determined labeled edges:

```
edge(2 1 q)
```
(C.4)

3. `LUSEdges`: the list of records `edge(I1 I2)` representing the determined edges from `I1` to `I2` whose edge label is not yet determined.

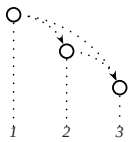
Dominance Edges. Also for each dimension, the edge record contains three kinds of dominance edges:

1. `DEdges`: the list of records `dom(I1 I2)` representing the determined dominance edges from `I1` to `I2`. We obtain this information from the lower bounds of the `down` sets of the nodes. For any node with index `I1` whose `daughters` set is not yet determined, we add dominance edges to all nodes `I2` in the lower bound of the `down` set of `I1` which:

   a) have an underspecified `mothers` set

   b) are not in any of the `down` sets of the nodes in the `down` set of `I1`

   where the latter condition excludes redundant dominance edges which are already entailed by transitivity. Why? As an example, consider an underspecified graph with three nodes with indices `1`, `2` and `3`, where the `down` set of node `1` contains both `2` and `3`, and `2` contains `3`. That is, if we would not exclude redundant dominance edges, we would add the three dominance edges `dom(1 2)`, `dom(1 3)` and `dom(2 3)`. We represent the "dominance graph" containing these dominance edges below, where we draw the dominance edges in a curved and dotted form:



(C.5)

Clearly, the dominance edge from node `1` to `3` is redundant, and is thus excluded.

As an example for finding the dominance edges, consider the down set of node 2 in Figure C.3:

```
down:'_'([1 3] [1 3#5] 2#4)
```
(C.6)

The lower bound of the set contains the nodes 1 and 3, where 1 is already a daughter of node 2 as we know from the daughters set in (C.1). Thus, 3 remains the only possible endpoint for a dominance edge from 2. In fact, it is an endpoint for the following dominance edge:

```
dom(2 3)
```
(C.7)

because node 3 has an underspecified mothers set (see Figure C.3) and is not entailed by transitivity: there is no other node in the down set of node 2 which has 3 in its down set.

2. LDEdges: the list of records dom(I1 I2 LA) of the determined labeled dominance edges from I1 to I2 labeled LA. For any I1, I2 and LA, dom(I1 I2 LA) is in LDEdges if:

   a) I2 is in the lower bound of the downL set of I1 for edge label LA

   b) dom(I1 I2) is in the list LDEdges

   For example, the downL sets of node 2 in Figure C.3 is the following:

   ```
   downL:o(q:[1] s:'_'([3] [3#5] 1#3))
   ```
   (C.8)

   The downL set for edge label q contains 1, which is not added as a labeled dominance edge since dom(2 1) is not in DEdges. The lower bound of the set for edge label s contains only node 3, which is added as the following labeled dominance edge, since dom(2 3) is in fact contained in DEdges:

   ```
   dom(2 3 s)
   ```
   (C.9)

3. LUSDEdges: the list of records dom(I1 I2) of dominance edges from I1 to I2 whose edge label is underspecified.

The dominance edges will prove beneficial for our example grammar in part III of the thesis, and in particular for its interface to CLLS, where we are interested in transforming partial, underspecified analyses obtained by the XDK constraint parser to CLLS constraints, i.e., underspecified semantic representations. The exclusion of redundant dominance edges, e.g. entailed by transitivity, will help us to avoid stipulating redundant CLLS constraints.

C.2. Output Library

The extensible output library contains functors for various kinds of visualizations:

- *Decode*: decoded solution (IL syntax), as in Figure C.2 above
- *Pretty*: pretty printed solution (OL syntax), as in Figure C.3 above

- *Dag*: graphical display of multigraphs using Tcl/Tk, see Figure 2.8 above
- *Latex*: graphical display of multigraphs as LaTeX code, used for all multigraph illustrations in the thesis.
- *CLLS*: visualizing (underspecified) solutions graphically as CLLS constraints, using *uDraw(Graph)* (Bernd Krieg-Brueckner's Group 2005). This output functor is explained in more detail in appendix E.

All textual output of the output functors can be redirected to standard I/O, into a file, into the *Oz Browser* or the *Oz Inspector* (Brunklaus 2000).

C.3. Summary

This appendix introduced the visualizer of the XDK. Visualization of the solutions of the constraint parser proceeds in two steps: output preparation followed by invoking a subset of the output functors from the extensible output library. That is, even the visualizer of the XDK is very modular and thus easily extensible.

D. Programs

This appendix deals with the programs which expose the functionality of the XDK: the metagrammar converter `xdkconv`, the metagrammar compiler `xdkc`, the constraint solver `xdks`, and the GUI `xdk`. We also describe the additional features of the XDK: the example grammars, a set of useful shell scripts, and its extensive documentation.

D.1. Metagrammar Converter

The metagrammar converter `xdkconv` converts metagrammars between the three metagrammar input syntaxes UL, XML and IL. For example, to convert the grammar `nut1.ul` from UL into XML syntax, it is called as follows:

```
$ xdkconv.exe -g Grammars/nut1.ul -o Grammars/nut1.xml
Converting grammar file "Grammars/nut1.ul" to "Grammars/nut1.xml"...  done.  (30ms)
```
(D.1)

D.2. Metagrammar Compiler

The metagrammar compiler `xdkc` compiles metagrammars, and is also able to merge a set of grammars into a single one, given that their type definitions are the same. Compiled grammars can then either be saved into Mozart/Oz records or into a GNU GDBM database. For example, to compile the grammar `nut1.ul` and save it into a GNU GDBM database, `xdkc` is called as follows:

```
$ xdkc.exe -g Grammars/nut1.xml -w db
Compiling grammar "Grammars/nut1.xml" ...  done.  (110ms)
Saved compiled grammar as "Grammars/nut1.slp_db".
```
(D.2)

D.3. Constraint Solver

The constraint solver `xdks` is a shell-based constraint parser. Input grammars can either be newly compiled using the metagrammar compiler or read in from precompiled pickles. `xdks` parses all sentences from a list of example sentences and prints out comprehensive parsing statistics using XML to standard I/O. To parse all sentences in `nut1.txt` using the precompiled grammar `nut1.slp_db`, and save the parsing statistics in the file `nut1.stat.xml`, the program is called as follows:

```
$ xdks.exe -g Grammars/nut1.slp_db -e Grammars/nut1.txt >nut1.stat.xml
```
(D.3)

We show parts of the file `nut1_statistics.xml` in Figure D.1 (grammar) and Figure D.2 (individual parses and aggregate counts). The statistics include:

- information about the grammar, including its dimensions (XML tag `dimensions`) and principles (`principles`)
- grammar profiling information (`gprofile`): the number of constraint variables created for each node, the number of entries in the lexicon etc.
- information about the individual parses (`string`), including the number of choices, the depth of the search tree, the number of failed and succeeded nodes, the parsing time
- individual parses profiling information (`sprofile`): the number of constraint variables and propagators used for parsing, and the number of entries per node
- aggregate counts and averages of the parses (`counts`)
- aggregate counts and averages of the profiling information (`profilecounts`)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE statistics SYSTEM "Extras/statistics.dtd">
<statistics>
  <grammar data="Grammars/nut1.slp_db"/>
  <examples data="Grammars/nut1.txt" count="13"/>
  <date data="Feb 24, 2006 14:51:12"/>
  <solutions data="9999"/>
  <reco data="1"/>
  <dimensions>
    <dimension data="lex"/>
    <dimension data="sem"/>
    <dimension data="syn"/>
    <dimension data="synsem"/>
  </dimensions>
  <principles>
    <principle data="syn principle.graph (syn)"/>
    <principle data="syn principle.tree (syn)"/>
    <principle data="syn principle.valency (syn)"/>
    <principle data="syn principle.agr (syn)"/>
    <principle data="syn principle.agreement (syn)"/>
    <principle data="syn principle.order3 (syn)"/>
    <principle data="syn principle.parse (syn)"/>
    <principle data="sem principle.graph (sem)"/>
    <principle data="sem principle.dag (sem)"/>
    <principle data="sem principle.valency (sem)"/>
    <principle data="synsem principle.linkingEnd (sem,syn)"/>
    <principle data="synsem principle.linkingMother (sem,syn)"/>
  </principles>
  <gprofile fd="5" fs="101" fdfs="106" entries="16">
    <gpnode fd="2" fs="1" fdfs="3"/>
    <gpattrs fd="1" fs="0" fdfs="1">
      <gpdimension data="lex" fd="0" fs="0" fdfs="0"/>
      <gpdimension data="sem" fd="0" fs="0" fdfs="0"/>
      <gpdimension data="syn" fd="1" fs="0" fdfs="1"/>
      <gpdimension data="synsem" fd="0" fs="0" fdfs="0"/>
    </gpattrs>
    <gpentry fd="0" fs="31" fdfs="31">
      <gpdimension data="lex" fd="0" fs="0" fdfs="0"/>
      <gpdimension data="sem" fd="0" fs="10" fdfs="10"/>
      <gpdimension data="syn" fd="0" fs="15" fdfs="15"/>
      <gpdimension data="synsem" fd="0" fs="6" fdfs="6"/>
    </gpentry>
    <gpmodel fd="2" fs="69" fdfs="71">
      <gpdimension data="lex" fd="0" fs="0" fdfs="0"/>
      <gpdimension data="sem" fd="1" fs="28" fdfs="29"/>
      <gpdimension data="syn" fd="1" fs="41" fdfs="42"/>
      <gpdimension data="synsem" fd="0" fs="0" fdfs="0"/>
    </gpmodel>
    <gplabel>
      <gpdimension data="lex" label="0"/>
      <gpdimension data="sem" label="5"/>
      <gpdimension data="syn" label="6"/>
      <gpdimension data="synsem" label="0"/>
    </gplabel>
  </gprofile>
  ...
```

Figure D.1.: XML parsing statistics and profiling (grammar)

```
...
<string id="string6">
  <words>
  Peter wants Mary to eat spaghetti today .
  </words>
  <outputs>
  </outputs>
  <choices data="3"/>
  <depth data="3"/>
  <failed data="0"/>
  <succeeded data="4"/>
  <time data="110"/>
  <sprofile fd="536" fs="2171" fdfs="2707" pr="16056" entries="1.5" words="8">
    <spnode index="1" word="Peter" entries="1" fd="5" fs="101" fdfs="106" pr="31"/>
    <spnode index="2" word="wants" entries="2" fd="5" fs="101" fdfs="106" pr="31"/>
    <spnode index="3" word="Mary" entries="1" fd="5" fs="101" fdfs="106" pr="31"/>
    <spnode index="4" word="to" entries="1" fd="5" fs="101" fdfs="106" pr="31"/>
    <spnode index="5" word="eat" entries="4" fd="5" fs="101" fdfs="106" pr="31"/>
    <spnode index="6" word="spaghetti" entries="1" fd="5" fs="101" fdfs="106" pr="31"/>
    <spnode index="7" word="today" entries="1" fd="5" fs="101" fdfs="106" pr="31"/>
    <spnode index="8" word="." entries="1" fd="5" fs="101" fdfs="106" pr="31"/>
  </sprofile>
</string>
...
<counts>
  <cchoices min="0" max="3" average="0.461538"/>
  <cdepth min="1" max="3" average="1.38462"/>
  <cfailed min="0" max="1" average="0.384615"/>
  <csucceeded min="0" max="4" average="1.07692"/>
  <ctime min="0" max="150" average="60.0"/>
</counts>
<profilecounts>
  <cfd min="126" max="536" average="278.615"/>
  <cfs min="816" max="2171" average="1378.85"/>
  <cfdfs min="942" max="2707" average="1657.46"/>
  <cpr min="4646" max="16056" average="9060.0"/>
  <cwords min="3" max="8" average="5.07692"/>
  <centries min="1" max="4" average="1.60606"/>
</profilecounts>
</statistics>
```

Figure D.2.: XML parsing statistics and profiling (individual parses and aggregate counts)

D.4. Graphical User Interface

The GUI `xdk` offers a convenient front-end for all the main functionality of the XDK: metagrammar conversion, metagrammar compilation, merging and pickling, constraint solving and the generation of parsing statistics. In addition, the GUI offers a variety of additional functions for grammar debugging. For instance, dimensions and principles can be individually switched off and on again, the *generate all orderings* function helps to spot overgeneration, and the graphical search engines Oz Explorer and IOzSeF give an overview of the search space of the constraint parser.

D.5. Example Grammars, Scripts and Documentation

The XDK comes with a large number of handcrafted example grammars. This includes:

- all grammars described in this thesis
- the German grammars described in (Duchier & Debusmann 2001), (Debusmann 2001) and (Bader et al. 2004):
- the German grammars described in the ESSLLI 2004 course
- the Dutch grammar described in (Debusmann & Duchier 2002)
- the English grammar used for the CHORUS project demonstration in April 2004, which is partly described in (Debusmann, Duchier, Koller, Kuhlmann, Smolka & Thater 2004)

- the English grammar described in (Debusmann 2004*b*)
- the English grammar described in (Debusmann et al. 2005)
- the Arabic grammar described in (Odeh 2004)

Moreover, the XDK provides a number of shell scripts e.g. for the convenient creation of pictures displaying multigraphs or metagrammars for inclusion in papers and presentations:

- `xdag2eps`, `xdag2jpg`, `xdag2pdf`: generate EPS, JPG or PDF files from the LaTeX code obtained using the visualizer for solutions of the constraint parser, using the LaTeX style file `xdag.sty` also provided by the XDK
- `code2pic`, generate EPS, JPG or PDF files from the LaTeX code obtained from the scripts `ozcolor` (for Mozart/Oz code), `ulcolor` (UL), `xmlcolor` (XML)
- `ulterse`: minimize UL metagrammars
- `diffnotime`: compare parsing statistics
- `addprinciple`, `mvprinciple`, `rmprinciple`: add, rename or remove principles to, in, or from the principle library

Many of these tools, and the additional GNU Emacs mode `ul.el`, have already been used to prepare this thesis.

The XDK is comprehensively documented by a manual which is over 200 pages long written using texinfo. It is available as an online version (in either HTML or info) and as an offline version for printing (in either PDF and Postscript). Even more in-depth documentation is available on the XDG website in form of slides of the ESSLLI 2004 course.

D.6. Summary

This appendix presented the programs of the XDK, the provided example grammars, the set of useful shell scripts, and its documentation. All grammars described in this thesis are implemented in XDK and can be tested “live”.

E. Interface to CLLS

In this chapter, we build an interface from the semantics module of our grammar to the *Constraint Language for Lambda Structures* (*CLLS*) (Egg et al. 2001). The interface covers the entire example grammar developed in this part of the thesis, i.e., for each analysis, it yields a corresponding CLLS constraint. We realize this interface by introducing the CLLS dimension to gather the necessary data to visualize underspecified PA/SC analyses as CLLS constraints, and the *CLLS* output functor to implement the visualization, using the graph visualizer *uDraw(Graph)* (Bernd Krieg-Brueckner's Group 2005).

E.1. CLLS

This section gives a very brief and informal introduction to CLLS. A more detailed description of CLLS can be found e.g. in (Egg et al. 2001). CLLS is a description language for lambda terms based on *dominance constraints* (Marcus, Hindle & Fleck 1983). Compared to other formalisms for semantic underspecification such as *Quasi Logical Form* (*QLF*) (Alshawi 1991), *Hole Semantics* (Bos 1996) and *Minimal Recursion Semantics* (*MRS*) (Copestake et al. 2004), CLLS has a number of advantages:

- descriptions from other formalisms, e.g. Hole Semantics or MRS, can be converted into CLLS (Koller, Niehren & Thater 2003, Fuchss, Koller, Niehren & Thater 2004)
- CLLS has by far the best algorithmic properties of the available formalisms
- CLLS has an open-source implementation: *Utool* (Koller, Kuhlmann & Thater 2005), not only offering services to translate descriptions from other formalisms into CLLS, but also to solve them very efficiently

E.1.1. Constraints

For simplicity, we restrict ourselves to a fragment of CLLS excluding parallelism constraints. Here is its syntax:

$$\varphi \quad ::= \quad X : f(X_1, \ldots, X_n) \mid X \lhd^* Y \mid \lambda(X) = Y \mid \varphi \wedge \varphi' \tag{E.1}$$

$X : f(X_1, \ldots, X_n)$ is a *labeling constraint*. It constrains the node variable X to have label f, and daughters $X_1, \ldots, X_n$ (in this order). $X \lhd^* Y$ is a *dominance constraint* requiring that X dominates Y. $\lambda(X) = Y$ is a *binding constraint* and requiring that X is bound by Y, where X must have label *var* for "variable", and Y must have label *lam* for "lambda binder".

E.1.2. Example

As an example, we create a CLLS description of the two readings of

Every man loves a woman (E.2)

in a step-by-step fashion. In the first step, we assign to each word a CLLS constraint called *fragment* which describes its semantic contribution. To the determiners *every* and *a*, we assign the following trivial labeling constraints:

$$\begin{array}{l} X_1 : every \\ X_1 : a \end{array} \tag{E.3}$$

To the noun *man*, we assign the fragment below, where @ stands for "application":

$$\begin{array}{l} X_1 : @(X_2, X_8) \wedge X_2 : @(X_3, X_4) \wedge X_4 \lhd^* X_5 \wedge X_5 : @(X_6, X_7) \wedge \\ X_6 : man \wedge X_7 : var \wedge X_8 : lam(X_9) \wedge \lambda(X_7) = X_8 \end{array} \tag{E.4}$$

The noun *woman* is assigned the same fragment except for the labeling of node X_8.

CLLS constraints can be represented more perspicuously as graphs. A representation of (E.4) is shown in Figure E.1, where unlabeled nodes have label _, edges are drawn as solid (black) lines going downward, dominance constraints as dotted (blue) lines also going downward, and binding constraints by dotted (green) lines going upward. Anchor nodes of the fragments, labeled by the words to which they correspond, are highlighted (in yellow).

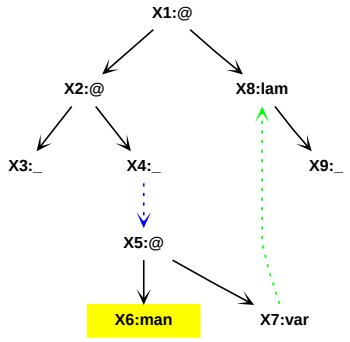


Figure E.1.: Graph representation of CLLS constraint (E.4) for the noun *man*

The CLLS fragment for the transitive verb *loves*, graphically represented in Figure E.2, represents the binary predicate $love(x,y)$, where x corresponds to the variable X_3 in the fragment (the agent), and y to X_5 (the patient):

$$X_1 : @(X_3, X_2) \wedge X_2 : @(X_5, X_4) \wedge X_4 : love \wedge X_5 : var \wedge X_3 : var \tag{E.5}$$

In the second step, we combine the constraints of the determiners and nouns by:

1. conjoining them
2. adding a dominance constraint from quantifier node X_3 of the noun constraint to the root node X_1 of the determiner constraint

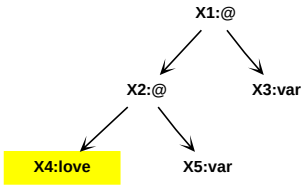


Figure E.2.: Graph representation of CLLS constraint (E.5) for the transitive verb *loves*

To make the nodes of the individual fragments distinct, we amalgamate the names of the node variables with the corresponding words. For instance, X_3 of the fragment of *man* becomes X_3^{man}. The resulting constraint for *every man*, represented graphically in Figure E.3, is the following:

$$\begin{array}{l} X_1^{man} : @(X_2^{man}, X_8^{man}) \wedge X_2^{man} : @(X_3^{man}, X_4^{man}) \wedge \\ X_4^{man} \lhd^* X_5^{man} \wedge X_5^{man} : @(X_6^{man}, X_7^{man}) \wedge \\ X_6^{man} : man \wedge X_7^{man} : var \wedge X_8^{man} : lam(X_9^{man}) \wedge \\ \lambda(X_7^{man}) = X_8^{man} \wedge \\ X_1^{every} : every \wedge X_3^{man} \lhd^* X_1^{every} \end{array} \tag{E.6}$$

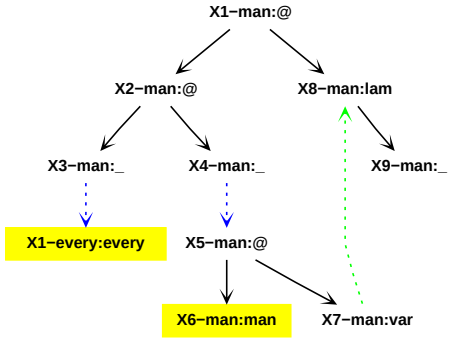


Figure E.3.: Graph representation of CLLS constraint (E.6) for the noun phrase *every man*

Intuitively, the constraint represents a *generalized quantifier* where the restriction is already instantiated: the constraint for *every man* for instance represents the following generalized quantifier:

$$\lambda Q. \forall x. man(x) \Rightarrow Q \tag{E.7}$$

where the restriction is already instantiated with $man(x)$, but the scope Q is not. The combination of *a* and *woman* proceeds analogously.

In the third step, we combine the constraints for the verb *loves* with those for *every man* and *a woman*. This amounts to:

1. again conjoining them
2. adding dominance constraints from the scope node X_9 of the noun fragments to the root node X_1 of the verb fragment
3. adding binding constraints from the variable nodes X_3 and X_5 of the verb fragment to the lambda binder node X_8 of the noun fragments.

The combined CLLS constraint, displayed graphically in Figure E.4, is the following:

$$
\begin{array}{l}
X_1^{man} : @(X_2^{man}, X_8^{man}) \wedge X_2^{man} : @(X_3^{man}, X_4^{man}) \wedge \\
X_4^{man} \lhd^* X_5^{man} \wedge X_5^{man} : @(X_6^{man}, X_7^{man}) \wedge \\
X_6^{man} : man \wedge X_7^{man} : var \wedge X_8^{man} : lam(X_9^{man}) \wedge \\
\lambda(X_7^{man}) = X_8^{man} \wedge \\
X_1^{every} : every \wedge X_3^{man} \lhd^* X_1^{every} \wedge \\
X_1^{woman} : @(X_2^{woman}, X_8^{woman}) \wedge X_2^{woman} : @(X_3^{woman}, X_4^{woman}) \wedge \\
X_4^{woman} \lhd^* X_5^{woman} \wedge X_5^{woman} : @(X_6^{woman}, X_7^{woman}) \wedge \\
X_6^{woman} : woman \wedge X_7^{woman} : var \wedge X_8^{woman} : lam(X_9^{woman}) \wedge \\
\lambda(X_7^{woman}) = X_8^{woman} \wedge \\
X_1^{a} : a \wedge X_3^{woman} \lhd^* X_1^{a} \wedge \\
X_1^{loves} : @(X_3^{loves}, X_2^{loves}) \wedge X_2^{loves} : @(X_5^{loves}, X_4^{loves}) \wedge \\
X_4^{loves} : love \wedge X_5^{loves} : var \wedge X_3^{loves} : var \wedge \\
X_9^{man} \lhd^* X_1^{loves} \wedge X_9^{woman} \lhd^* X_1^{loves} \wedge \\
\lambda(X_3^{loves}) = X_8^{man} \wedge \lambda(X_5^{loves}) = X_8^{woman}
\end{array}
\tag{E.8}
$$

The constraint reflects the intuition that the nominal arguments of the verb both take scope over it, that the agent-variable of the verb is bound by the subject fragment corresponding to *every man*, and the patient by the object fragment corresponding to *a woman*.

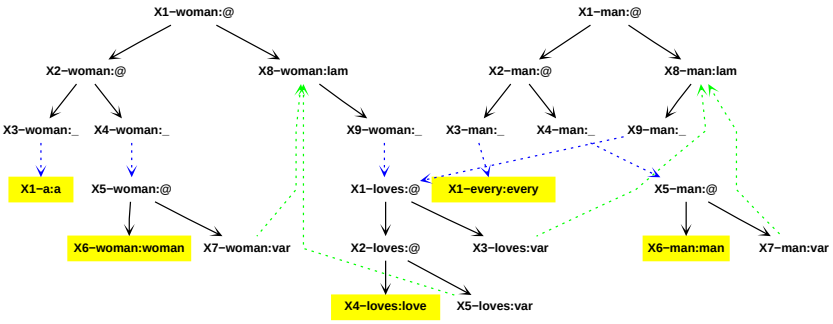


Figure E.4.: Graph representation of CLLS constraint (E.8) for the sentence *Every man loves a woman.*

CLLS constraints such as the one developed above describe sets of trees, which can be enumerated by solving the CLLS constraint. We display the two *solved forms* of constraint (E.8) in Figure E.5 (weak reading) and Figure E.6 (strong reading). Solved forms directly correspond to lambda terms.

E.2. CLLS Dimension

To integrate CLLS into our example grammar, we introduce the CLLS dimension, whose purpose is to provide the information required to construct a CLLS constraint from an underspecified PA/SC analysis. As a result, we will always be able to visualize the semantic part of an analysis as a CLLS constraint, using the specialized CLLS output functor of the XDK output library. The models of the CLLS dimension are graphs without edges.

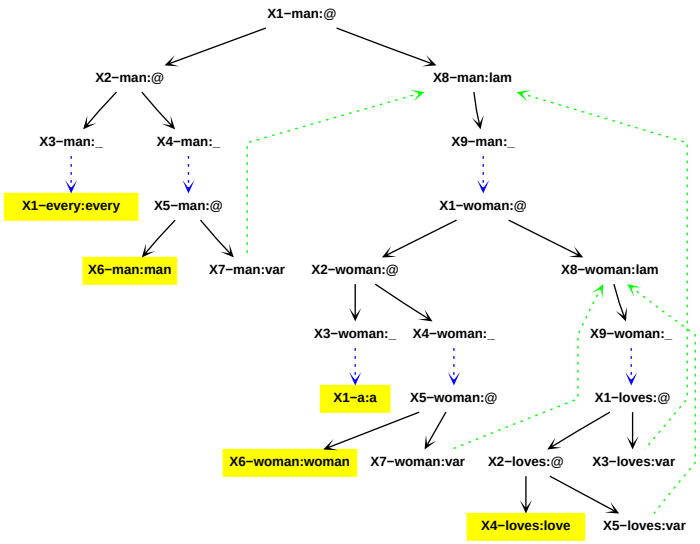


Figure E.5.: Weak reading of *Every man loves a woman.*

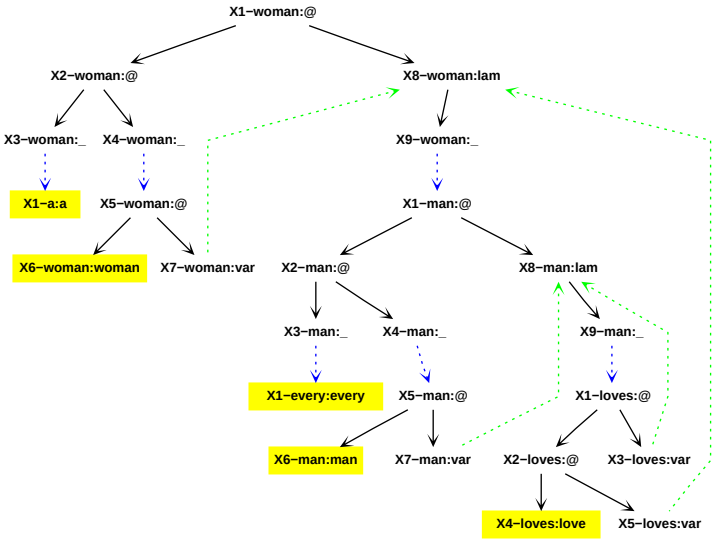


Figure E.6.: Strong reading of *Every man loves a woman.*

E.2.1. Types

The lexical attributes on the CLLS dimension are defined as follows, making use of the type `"clls.var"` of CLLS node variables:[1]

```
deftype "clls.var" {x1 x2 x3 x4 x5 x6 x7 x8 x9}
defentrytype {cons: string
              anchor: string
              roots: set("clls.var")
              dom: vec("sc.label" set("clls.var"))
              lam: vec("pa.label" set("clls.var"))
              var: vec("pa.label" set("clls.var"))}
```
(E.9)

The attributes consist of:

- `cons` is a string which represents the fragment assigned to the node, where we write `label(f x1 ... xn)` for the labeling constraint $X : f(X_1, \ldots, X_n)$, `dom(x1 x2)` for the dominance constraint $X_1 \lhd^* X_2$, `lambda(x1 x2)` for the binding constraint $\lambda(X_1) = X_2$, and use concatenation for conjoining constraints
- `anchor` is a string representing the anchor of the fragment
- `roots` is a set of node variables denoting the roots of the fragment
- `dom` is a vector used to map SC edge labels to sets of node variables. These node variables are typically leaves of the fragments, and are the startpoints of the dominance constraints corresponding to SC edges. The endpoints of these dominance constraints are always the roots of other fragments.
- `lam` and `var` are vectors used to map PA edge labels to sets of node variables. `var` maps PA edge labels like `ag` and `pat` to the corresponding node variables in the CLLS constraint. For example, in the CLLS constraint for *loves* in Figure E.2, `ag` corresponds to node X_3 and `pat` to X_5. These node variables are the startpoints of the binding constraints corresponding to PA edges. Their endpoints are lambda binders, whose position is specified by the attribute `lam`, a vector used to map PA edge labels to lambda binder node variables.

E.2.2. Lexical Classes

We use lexical classes to assign CLLS constraints to nodes. In this section, we describe only the lexical classes needed to account for the running example of this appendix. In the actual grammar, we have defined CLLS constraints for all other words as well.

Words without semantic content are assigned an empty CLLS constraint without a root and with anchor `A`:

```
defclass "clls_nocont" A {
  dim clls {cons: ""
            anchor: A
            roots: {}}}
```
(E.10)

[1] As the fragments in the example grammar have at most 9 nodes, `"clls.var"` contains the 9 variables `x1`, ..., `x9` representing $X_1, \ldots, X_9$.

To determiners, we assign a fragment defining only one node x1 labeled by the anchor A:[2]

```
defclass "clls_det" A {
  dim clls {cons: "label(x1 anchor)"
            anchor: A
            roots: {x1}}}
```
(E.11)

We describe common nouns with the following lexical class:

```
defclass "clls_cnoun" A {
  dim clls {cons: "label(x1 '@'(x2 x8)) label(x2 '@'(x3 x4))
                   dom(x4 x5) label(x5 '@'(x6 x7)) label(x6 anchor)
                   label(x7 var) label(x8 lambda(x9)) lam(x7 x8)"
            anchor: A
            roots: {x1}
            dom: {q: {x3}
                  r: {x4}
                  s: {x9}}
            lam: {ag: {x8}
                  pat: {x8}
                  addr: {x8}
                  agm: {x8}
                  patm: {x8}}}}
```
(E.12)

The CLLS constraint of the lexical class corresponds to that of (E.4), and graphically displayed in Figure E.1 above, except that its anchor (node variable x6) is variable. The root of the fragment is x1. By the dom attribute, its quantifier node is x3, its restriction x4 and its scope x9. By the lam attribute, the endpoint for binding constraints from verbs (ag, pat and addr) and modifiers of the noun (agm and patm) is x8.

Transitive verbs are described as follows:

```
defclass "clls_trans" A {
  dim clls {cons: "label(x1 '@'(x2 x3)) label(x2 '@'(x4 x5))
                   label(x4 anchor) label(x5 var) label(x3 var)"
            anchor: A
            roots: {x1}
            var: {ag: {x3}
                  pat: {x5}}}}
```
(E.13)

where the CLLS constraint corresponds to that of (E.5), graphically displayed in Figure E.2. The root of the fragment is x1. By the var attribute, its agent corresponds to node variable x3, and its patient to x5

E.3. CLLS Output Functor

The purpose of the *CLLS output functor* is to:

1. from a (possibly underspecified) PA/SC analysis, construct the corresponding CLLS constraint, utilizing the information provided by the lexical attributes on the CLLS dimension

[2] Before visualization, the CLLS output functor replaces all occurrences of anchor in the constructed CLLS constraint by the respective anchor of the node.

2. visualize the CLLS constraint in the graph visualizer *uDraw(Graph)* (Bernd Krieg-Brueckner's Group 2005)

The construction of the CLLS constraint to be visualized proceeds in four steps:

1. preprocessing the fragments of the nodes provided by the CLLS dimension
2. concatenating them
3. adding dominance constraints corresponding to edges and dominance edges on the SC dimension
4. adding binding constraints corresponding to the edges on the PA dimension

For the visualization itself, we apply the interface to uDraw(Graph) provided by Joachim Niehren.

As an example, we give a walkthrough of the construction of the CLLS constraint for *Every man loves a woman* from the underspecified PA/SC analysis shown in Figure E.7.

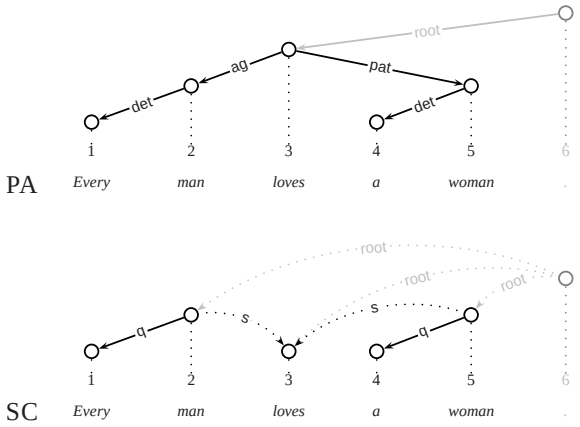


Figure E.7.: Example PA/SC/CLLS analysis

E.3.1. Preprocessing the Fragments

Preprocessing consists of two steps:

1. instantiating the anchor nodes of the fragments with the base form of the corresponding word
2. making the node variables of the fragments unique by amalgamating them with the corresponding node index

As an example, the `cons` value of the determiner *every* on the CLLS dimension is defined as follows by lexical class `clls_det` in (E.11) above:

```
"label(x1 anchor)"
```
(E.14)

After preprocessing, the anchor has been instantiated with `every`, and the node variable `x1` has been made unique by the suffix `_f1` (where `f` stands for "fragment"):

```
"label(x1_f1 every)"
```
(E.15)

E.3.2. Concatenating the Fragments

In the second step, the CLLS output functor concatenates the preprocessed fragments, yielding the following CLLS constraint, which we display graphically in Figure E.8.

```
"label(x1_f1 every)

label(x1_f2 '@'(x2_f2 x8_f2)) label(x2_f2 '@'(x3_f2 x4_f2))
dom(x4_f2 x5_f2) label(x5_f2 '@'(x6_f2 x7_f2)) label(x6_f2 man)
label(x7_f2 var) label(x8_f2 lambda(x9_f2)) lam(x7_f2 x8_f2)

label(x1_f3 '@'(x2_f3 x3_f3)) label(x2_f3 '@'(x4_f3 x5_f3))
label(x4_f3 love) label(x5_f3 var) label(x3_f3 var)

label(x1_f4 a)

label(x1_f5 '@'(x2_f5 x8_f5)) label(x2_f5 '@'(x3_f5 x4_f5))
dom(x4_f5 x5_f5) label(x5_f5 '@'(x6_f5 x7_f5)) label(x6_f5 woman)
label(x7_f5 var) label(x8_f5 lambda(x9_f5)) lam(x7_f5 x8_f5)"
```
(E.16)

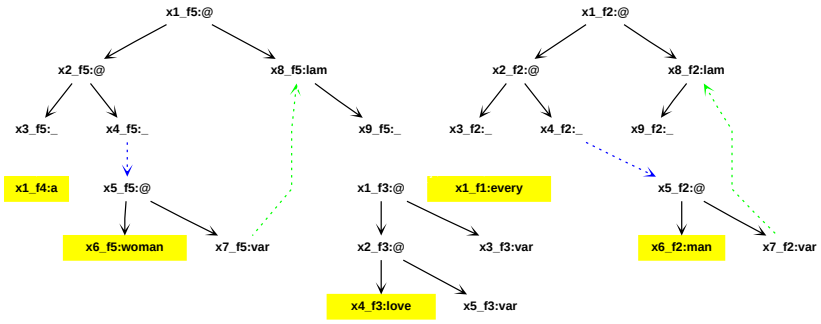


Figure E.8.: Graphical representation of CLLS constraint (E.16)

E.3.3. Adding Dominance Constraints

In the third step, the output functor adds dominance constraints corresponding to the edges and the dominance edges on the (possibly underspecified) SC dimension. For each edge or dominance edge from v to v' labeled l, the startpoint of the corresponding dominance constraint is specified by the lexical attribute `dom` for v and label l, and the endpoint by the lexical attribute `roots` for v'.

For example, consider the edge labeled q from *man* to *every* in the underspecified SC analysis in Figure E.7. As *man* is a common noun, it is characterized by lexical class `clls_cnoun` (E.12), and the set of startpoints of dominance constraints for *man* and label q is x3. The set of roots of the determiner *every*, characterized by lexical class `clls_det` (E.11), contains only x1. As a result, the dominance constraint corresponding to the edge goes from x3 of *man* (unique name x3_f2) to x1 of *every* (x1_f1). Similarly, the edge from *woman* to *a* induces a dominance constraint from x3_f5 to x1_f4. The two additional dominance constraints are displayed below:

```
"dom(x3_f2 x1_f1) dom(x3_f5 x1_f4)"
```
(E.17)

As another example, consider the dominance edges labeled s from *man* to *loves* and from *woman* to *loves* in Figure E.7. The startpoint for dominance edges from common nouns and label s is x9 (E.12), hence the startpoints of the dominance edges are x9_f2 for *man* and x9_f5 for *woman*. Both times, the endpoint is the root x1 of the fragment of *loves*, i.e., x1_f3. The result are the following two added dominance constraints:

```
"dom(x9_f2 x1_f3) dom(x9_f5 x1_f3)"
```
(E.18)

Together with the four additional dominance constraints corresponding to the two q edges and the two s dominance edges, the CLLS constraint (E.16) is graphically represented in Figure E.9.

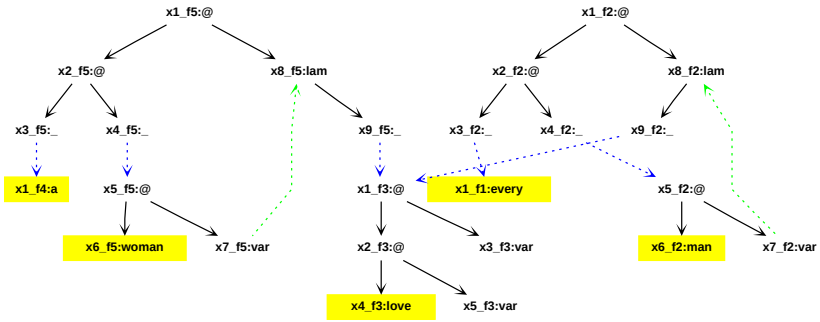


Figure E.9.: Graphical representation of CLLS constraint (E.16) with the additional dominance constraints (E.17) and (E.18)

In the output functor, we implement this idea of adding dominance constraints as follows. As explained in section C.1.2 of appendix C, the *output preparer* provides, among other things:

1. the list `NodeOLs` of the nodes of the analysis in OL syntax
2. the list `LEdges` of determined labeled edges `edge(I1 I2 LA)` of the analysis for each dimension

3. the list LDEdges of determined labeled dominance edges dom(I1 I2 LA) of the analysis for each dimension

where the list LDEdges of dominance edges excludes redundant dominance edges already entailed by "proper" edges or by transitivity. This ensures that the CLLS output functor does not add redundant dominance constraints.

The three lists are used in the function AddDomCons, which returns the list of dominance constraints to be added:

```
(  1) fun {AddDomCons NodeOLs LEdgesSC LDEdgesSC}
(  2)    for Edge in {Append LEdgesSC LDEdgesSC} collect:Collect do
(  3)       I1 = Edge.1
(  4)       I2 = Edge.2
(  5)       LA = Edge.3
(  6)
(  7)       NodeOL1 = {Nth NodeOLs I1}
(  8)       NodeOL2 = {Nth NodeOLs I2}
(  9)
( 10)       VarAs1 = NodeOL1.clls.entry.dom.LA
( 11)       VarAs2 = NodeOL2.clls.entry.roots
( 12)    in
( 13)       if {Length VarAs1==1} andthen {Length VarAs2==1} then
( 14)          VarA1 = {Nth VarAs1 1}#'_f'#I1
( 15)          VarA2 = {Nth VarAs2 1}#'_f'#I2
( 16)       in
( 17)          {Collect 'dom('#VarA1#' '#VarA2#')'}
( 18)       end
( 19)    end
( 20) end
( 21)
```

(E.19)

The function iterates over all determined labeled edges LEdgesSC and all determined labeled dominance edges LDEdgesSC on the SC dimension (line 2). The starting point of the edge/dominance edge on SC is I1, the endpoint I2, and the label LA (lines 3–5). In lines 7 and 8, we obtain the node records of the nodes I1 and I2 in OL syntax. In line 10, we access the lexical attribute dom for NodeOL1 and edge label LA on the CLLS dimension to obtain the list of atoms VarAs1, which is the OL representation of the set of startpoints of the dominance constraints for label LA. In line 11, we access roots of NodeOL2 to obtain VarAs2, the OL representation of the set of roots of the fragment of NodeOL2, which serve as the endpoints of the dominance constraints. If both lists contain precisely one node variable, line 17 adds the dominance constraint from the startpoint node variable VarA1 to the endpoint VarA2, where VarA1 is the first element of VarAs1, made unique by the suffix _f1 (line 14), and analogously for VarA2 (line 15).

E.3.4. Adding Binding Constraints

In the fourth and last step, we turn our attention to the PA dimension and add binding constraints corresponding to the determined edges on the (also possibly underspecified) PA dimension. For each edge (not dominance edge) from v to v' labeled l, the startpoint of the corresponding binding constraint is specified by the lexical attribute var for v and label l, and the endpoint by the lexical attribute lam for v' and l.

For instance, consider the edge labeled ag from *loves* to *man* in Figure E.7. As *loves* is a transitive verb, it is characterized by the lexical class clls_trans (E.13), stating that the startpoint of binding constraints for label ag is x3. *man* is characterized by clls_cnoun (E.12), stating that the endpoint of binding constraints for label ag is its node variable x8. As a result, the binding constraint corresponding to the edge goes from x3 of *loves* (x3_f3) to x8 of *man* (x8_f2). Similarly, the pat edge from *loves* to *woman* adds a binding constraint from x3_f3 to x8_f5:

```
"lambda(x3_f3 x8_f2) lambda(x5_f3 x8_f5)"
```
(E.20)

We show the resulting CLLS constraint in Figure E.10.

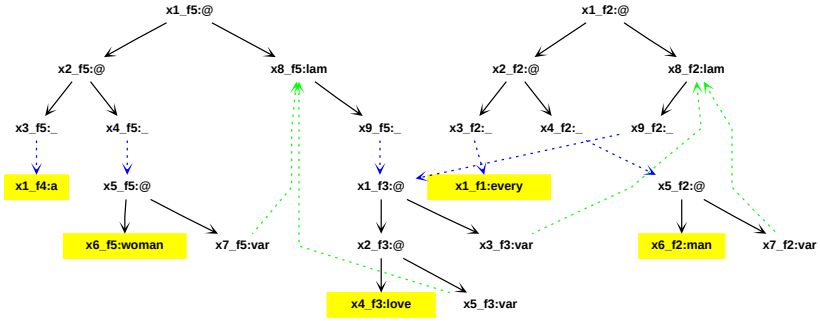


Figure E.10.: Graphical representation of CLLS constraint (E.16) with the additional dominance constraints (E.17) and (E.18), and the additional binding constraints (E.20)

We realize this idea as follows by the function AddBindingCons displayed below:

```
(  1) fun {AddBindingCons NodeOLs LEdgesPA}
(  2)    for edge(I1 I2 LA) in LEdgesPA collect:Collect do
(  3)       NodeOL1 = {Nth NodeOLs I1}
(  4)       NodeOL2 = {Nth NodeOLs I2}
(  5)
(  6)       VarAs1 = NodeOL1.clls.entry.var.LA
(  7)       VarAs2 = NodeOL2.clls.entry.lam.LA
(  8)    in
(  9)       if {Length VarAs1==1} andthen {Length VarAs2==1} then
( 10)          VarA1 = {Nth VarAs1 1}#'_f'#I1
( 11)          VarA2 = {Nth VarAs2 1}#'_f'#I2
( 12)       in
( 13)          {Collect 'lambda('#VarA1#' '#VarA2#')'}
( 14)       end
( 15)    end
( 16) end
```
(E.21)

The function iterates over the determined labeled edges edge(I1 I2 LA) in LEdgesPA on the PA dimension (line 2). It then obtains the startpoint and endpoints of the binding constraint corresponding to the edge (lines 3–7), and, if both are given, adds the binding constraint in line 13.

E.4. Summary

As the XDK constraint solver can selectively postpone the enumeration of readings on the individual dimensions, our approach supports scope underspecification out of the box, without any further stipulation. This has opened the door for an interface to CLLS, for which we have introduced the CLLS dimension to gather the necessary information to construct a CLLS constraint from a (possibly underspecified) PA/SC analysis. The CLLS constraint was then constructed by the CLLS output functor. By showing that it can be related to the state-of-the-art in underspecified semantics, we demonstrated that our model of semantics in terms of the two dimensions of predicate-argument structure and scope structure is not such a radical departure from state-of-the-art semantic representations as it might first have seemed.

Bibliography

Ajdukiewicz, K. (1935), Die Syntaktische Konnexität, *in* S. McCall, ed., 'Polish Logic 1920-1939', Oxford University Press, pp. 207–231. Translated from Studia Philosophica, 1, 1–27.

Alshawi, H. (1991), 'Resolving quasi logical forms', *Computational Linguistics* **16**(3), 133–144.

Andrews, P. B. (2002), *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Kluwer Academic Publishers.

Apt, K. R. (2003), *Principles of Constraint Programming*, Cambridge University Press.

Bader, R., Foeldesi, C., Pfeiffer, U. & Steigner, J. (2004), 'Modellierung grammatischer Phänomene der deutschen Sprache mit Topologischer Dependenzgrammatik'. Softwareprojekt, Saarland University.

Bar-Hillel, Y. (1953), 'A quasi-arithmetical notation for syntactic description', *Language* **29**, 47–58.

Barton, G. E., Berwick, R. & Ristad, E. S. (1987), *Computational Complexity and Natural Language*, MIT Press.

Becker, T., Rambow, O. & Niv, M. (1992), The derivational generative power, or, scrambling is beyond LCFRS, Technical report, University of Pennsylvania.

Beldiceanu, N. & Contjean, E. (1994), 'Introducing global constraints in CHIP', *Mathematical and Computer Modelling* pp. 97–123.

Bernd Krieg-Brueckner's Group (2005), 'uDraw(Graph)'. http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html.

Blackburn, P. & Gardent, C. (1995), A specification language for lexical functional grammars, *in* 'Proceedings of EACL 1995', Dublin/IE.

Böhmová, A., Hajič, J., Hajičová, E. & Hladká, B. (2001), The Prague Dependency Treebank: Three-level annotation scenario, *in* 'Treebanks: Building and Using Syntactically Annotated Corpora', Kluwer Academic Publishers.

Bojar, O. (2004), Problems of inducing large coverage constraint-based dependency grammar, *in* 'Proceedings of the International Workshop on Constraint Solving and Language Processing', Roskilde/DK.

Bos, J. (1996), Predicate logic unplugged, *in* 'Proceedings of the 10th Amsterdam Colloquium', pp. 133–143.

Brants, T. (1999), *Tagging and Parsing with Cascaded Markov Models — Automation of Corpus Annotation*, Saarbrücken Dissertations in Computational Linguistics and Language Technology, DFKI Saarbrücken.

Bresnan, J. (2001), *Lexical Functional Syntax*, Blackwell.

Bresnan, J. & Kaplan, R. (1982), Lexical-Functional Grammar: A formal system for grammatical representation, *in* J. Bresnan, ed., 'The Mental Representation of Grammatical Relations', The MIT Press, Cambridge/US, pp. 173–281.

Bresnan, J. W., Kaplan, R. M., Peters, S. & Zaenen, A. (1983), 'Cross-serial dependencies in dutch', *Linguistic Inquiry 13* pp. 173–281.

Bröker, N. (1999), *Eine Dependenzgrammatik zur Kopplung heterogener Wissensquellen*, Linguistische Arbeiten 405, Max Niemeyer Verlag, Tübingen/DE.

Brunklaus, T. (2000), Der Oz Inspector — Browsen: Interaktiver, einfacher, effizienter, Diploma thesis, Saarland University. http://www.ps.uni-sb.de/Papers/abstracts/OzInspector.html.

Butt, M. & King, T. H. (1998), Interfacing phonology with LFG, *in* 'Proceedings of the LFG98 Conference', Brisbane/AU.

Candito, M.-H. (1996), A principle-based hierarchical representation of LTAG, *in* 'Proceedings of COLING 1996', Kopenhagen/DK.

Candito, M.-H. & Kahane, S. (1998), Can the TAG derivation tree represent a semantic graph? An answer in the light of Meaning-Text Theory, *in* 'Fourth International Workshop on Tree Adjoining Grammars and Related Frameworks', University of Pennsylvania, Philadelphia/US, pp. 25–28.

Carpenter, B. (1992), *The Logic of Typed Feature Structures*, Cambridge Tracts in Theoretical Computer Science, 32 edn, Cambridge University Press.

Chomsky, N. (1957), *Syntactic Structures*, Janua linguarum, Mouton, The Hague/NL.

Chomsky, N. (1965), *Aspects of the Theory of Syntax*, MIT Press, Cambridge/US.

Chomsky, N. (1981), *Lectures on Government and Binding: The Pisa Lectures*, Foris Publications.

Church, A. (1940), 'A formulation of the simple theory of types', *Journal of Symbolic Logic* **5**, 56–68.

Clark, S. & Curran, J. R. (2004), The importance of supertagging for wide-coverage CCG parsing, *in* 'Proceedings of COLING 2004', pp. 282–288.

Copestake, A. & Flickinger, D. (2000), An open-source grammar development environment and broad-coverage English grammar using HPSG, *in* 'Conference on Language Resources and Evaluation', Athens/GR.

Copestake, A., Flickinger, D., Pollard, C. & Sag, I. (2004), 'Minimal recursion semantics. an introduction.', *Journal of Language and Computation* . To appear.

Crabbé, B. (2005), Grammatical development with XMG, *in* 'Proceedings of LACL 05', Bordeaux/FR.

Crabbé, B. & Duchier, D. (2004), Metagrammar redux, *in* 'Proceedings of the International Workshop on Constraint Solving and Language Processing', Roskilde/DK.

Dalrymple, M., Lamping, J., Pereira, F. & Saraswat, V. (1995), Linear Logic for meaning assembly, *in* 'Proceedings of the Workshop on Computational Logic for Natural Language Processing', Edinburgh/UK.

Debusmann, R. (2001), A declarative grammar formalism for dependency grammar, Diploma thesis, Saarland University. http://www.ps.uni-sb.de/Papers/abstracts/da.html.

Debusmann, R. (2004*a*), 'Modeling natural language with Topological Dependency Grammar'. Fortgeschrittenenpraktikum/Softwareprojekt, Wintersemester 2003/2004.

Debusmann, R. (2004*b*), Multiword expressions as dependency subgraphs, *in* 'Proceedings of the ACL 2004 Workshop on Multiword Expressions: Integrating Processing', Barcelona/ES.

Debusmann, R. & Duchier, D. (2002), Topological dependency analysis of the Dutch verb cluster, Technical report, Saarland University.

Debusmann, R. & Duchier, D. (2004), 'A comparative introduction to extensible dependency grammar'. Introductory course at the 16th European Summer School in Logic, Language and Information, ESSLLI 2004, Nancy, http://www.ps.uni-sb.de/~rade/talks.html.

Debusmann, R. & Duchier, D. (2006), 'XDG development kit'. http://www.mozart-oz.org/mogul/info/debusmann/xdk.html.

Debusmann, R., Duchier, D., Koller, A., Kuhlmann, M., Smolka, G. & Thater, S. (2004), A relational syntax-semantics interface based on dependency grammar, *in* 'Proceedings of COLING 2004', Geneva/CH.

Debusmann, R., Duchier, D. & Kruijff, G.-J. M. (2004), Extensible Dependency Grammar: A new methodology, *in* 'Proceedings of the COLING 2004 Workshop on Recent Advances in Dependency Grammar', Geneva/CH.

Debusmann, R., Duchier, D., Kuhlmann, M. & Thater, S. (2004), TAG as dependency grammar, *in* 'Proceedings of TAG+7', Vancouver/CA.

Debusmann, R., Postolache, O. & Traat, M. (2005), A modular account of information structure in Extensible Dependency Grammar, *in* 'Proceedings of the CICLING 2005 Conference', Springer, Mexico City/MX.

Dienes, P., Koller, A. & Kuhlmann, M. (2003), Statistical A* dependency parsing, *in* 'Prospects and Advances in the Syntax/Semantics Interface', Nancy/FR.

Dowty, D. R. (1989), On the semantic content of the notion of "thematic role", *in* G. Chierchia, B. H. Partee & R. Turner, eds, 'Properties, Types and Meanings', Vol. 2, Kluwer, Dordrecht/NL, pp. 69–129.

Duchier, D. (1999), Axiomatizing dependency parsing using set constraints, *in* 'Proceedings of MOL 6', Orlando/US.

Duchier, D. (2003), 'Configuration of labeled trees under lexicalized constraints and principles', *Research on Language and Computation* **1**(3–4), 307–336.

Duchier, D. & Debusmann, R. (2001), Topological dependency trees: A constraint-based account of linear precedence, *in* 'Proceedings of ACL 2001', Toulouse/FR.

Duchier, D., Le Roux, J. & Parmentier, Y. (2004), The Metagrammar compiler: An NLP application with a multi-paradigm architecture, *in* 'Proceedings of the MOZ04 Conference', Vol. 3389, Springer, Charleroi/BE.

Earley, J. (1970), 'An efficient context-free parsing algorithm', *Communications of the ACM* **13**(2), 451–455.

Egg, M., Koller, A. & Niehren, J. (2001), 'The Constraint Language for Lambda Structures', *Journal of Logic, Language, and Information* .

Erdmann, O. (1886), *Grundzüge der deutschen Syntax nach ihrer geschichtlichen Entwicklung dargestellt*, Erste Abteilung, Stuttgart/DE.

Frank, A. & Erk, K. (2004), Towards an LFG syntax-semantics interface for frame semantics annotation, *in* A. Gelbukh, ed., 'Computational Linguistics and Intelligent Text Processing', Lecture Notes in Computer Science, Springer Verlag.

Frank, A. & van Genabith, J. (2001), GlueTag. Linear Logic-based semantics for LTAG—and what it teaches us about LFG and LTAG, *in* M. Butt & T. H. King, eds, 'Proceedings of the LFG01 Conference', Hong Kong/HK.

Fuchss, R., Koller, A., Niehren, J. & Thater, S. (2004), Minimal recursion semantics as dominance constraints: Translation, evaluation, and analysis, *in* 'Proceedings of ACL 2004', Barcelona/ES.

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

Gardent, C. & Kallmeyer, L. (2003), Semantic construction in FTAG, *in* 'Proceedings of EACL 2003', Budapest/HU.

Gazdar, G., Klein, E., Pullum, G. & Sag, I. (1985), *Generalized Phrase Structure Grammar*, B. Blackwell, Oxford/UK.

Gerdes, K. & Kahane, S. (2001), Word order in German: A formal dependency grammar using a topological hierarchy, *in* 'ACL 2001 Proceedings', Toulouse/FR.

Goldsmith, J. (1979), Autosegmental Phonology, PhD thesis, MIT.

Goldsmith, J. (1990), *Autosegmental and Metrical Phonology*, Blackwell, Cambridge/US.

Grabowski, R., Kuhlmann, M. & Möhl, M. (2005), Lexicalised Configuration Grammars, *in* 'Proceedings of the Second International Workshop on Constraint Solving and Language Processing', Springer, Sitges/ES.

Gross, M. (1964), On the equivalence of models of language used in the fields of mechanical translation and information retrieval, *in* 'Information Storage and Retrieval', Harvard University, pp. 43–57.

Harary, F. (1994), *Graph Theory*, Addison-Wesley, Reading/US.

Harper, M. P., Hockema, S. A. & White, C. M. (1999), Enhanced constraint dependency parsers, *in* 'Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing', Honolulu/US.

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

Heinecke, J., Kunze, J., Menzel, W. & Schröder, I. (1998), Eliminative parsing with graded constraints, *in* 'Proceedings of COLING/ACL 1998', Montréal/CA, pp. 526–530.

Hentenryck, P. V. & Saraswat, V. (1996), 'Strategic directions in constraint programming', *ACM Computing Surveys* **28**(4), 701–726.

Henz, M., Müller, T. & Thiel, S. (2004), 'Global constraints for round robin tournament scheduling', *European Journal of Operational Research (EJORS)* .

Herling, S. (1821), 'Über die Topik der deutschen Sprache'.

Higgins, D. (1998), Parsing parallel grammatical representations, *in* 'Proceedings of COLING/ACL 1998', Montréal/CA.

Holan, T., Kubon, V., Oliva, K. & Platek, M. (2000), 'On complexity of word-order', *Journal t.a.l.* pp. 273–301.

Hudson, R. A. (1990), *English Word Grammar*, B. Blackwell, Oxford/UK.

Iordanskaja, L. & Mel'čuk, I. (2005), Towards establishing an inventory of surface-syntactic relations: Valency-controled surface-syntactic dependents of verb in french. to appear.

Jackendoff, R. (1977), *$\bar{X}$ Syntax: A Study of Phrase Structure*, number 2 *in* 'Linguistic Inquiry Monographs', MIT Press, Cambridge/US.

Jackendoff, R. (2002), *Foundations of Language*, Oxford University Press.

Jaffar, J. & Lassez, J.-L. (1988), From unification to constraints, *in* 'Proceedings of the 6th Conference on Logic programming '87', Springer, Tokyo/JP, pp. 1–18.

Jaffar, J. & Maher, M. M. (1994), 'Constraint Logic Programming: A survey', *The Journal of Logic Programming* **19/20**, 503–582. Special Issue: Ten Years of Logic Programming.

Joshi, A. K. (1987), An introduction to tree-adjoining grammars, *in* A. Manaster-Ramer, ed., 'Mathematics of Language', John Benjamins, Amsterdam/NL, pp. 87–115.

Joshi, A. K. & Bangalore, S. (1994), Disambiguation of super parts of speech (or supertags): Almost parsing, *in* 'Proceedings of COLING 1994', Kyoto/JP.

Joshi, A. K., Levy, L. & Takahashi, M. (1975), 'Tree Adjunct Grammars', *Journal of Computer and System Sciences* **10**(1).

Joshi, A. K. & Shanker, V. K. (1999), Compositional semantics with Lexicalized Tree Adjoining Grammar (LTAG): How much underspecification is necessary?, *in* H. C. Blunt & E. G. C. Thijsse, eds, 'Proceedings of the Third International Workshop on Computational Semantics (IWCS-3)', Tilburg/NL, pp. 131–145.

Kahane, S. (2001), 'A fully lexicalized grammar for french based on Meaning-Text Theory', *Computational Linguistics* .

Kallmeyer, L. & Joshi, A. K. (2003), 'Factoring predicate argument and scope semantics: Underspecified semantics with LTAG', *Research on Language and Computation* **1**(1–2), 3–58.

Kathol, A. (2000), *Linear Syntax*, Oxford University Press.

Kay, M. (1980), Algorithm schemata and data structures in syntactic processing, Technical report, Xerox Palo Alto Research Center. CSL-80-12.

Koller, A., Kuhlmann, M. & Thater, S. (2005), 'utool: The swiss army knife of underspecification'. http://utool.sourceforge.org/.

Koller, A., Niehren, J. & Thater, S. (2003), Bridging the gap between underspecification formalisms: Hole semantics as dominance constraints, *in* 'Proceedings of EACL 2003', Budapest/HU.

Koller, A. & Striegnitz, K. (2002), Generation as dependency parsing, *in* 'Proceedings of ACL 2002', Philadelphia/US.

Korthals, C. (2003), Unsupervised learning of word order rules, Master's thesis, Saarland University. Diploma thesis.

Kruijff, G.-J. M. & Baldridge, J. (2004), Generalizing dimensionality in Combinatory Categorial Grammar, *in* 'Proceedings of COLING 2004', Geneva/CH.

Kruijff, G.-J. M. & Duchier, D. (2003), Information structure in Topological Dependency Grammar, *in* 'Proceedings of EACL 2003', Budapest/HU.

Kubon, V. (2001), Problems of Robust Parsing of Czech, PhD thesis, Institute of Formal and Applied Linguistics, Prague/CZ.

Kunze, J. (1975), *Abhängigkeitsgrammatik*, Akademie Verlag, Berlin/DE.

Marcus, M. P., Hindle, D. & Fleck, M. M. (1983), D-theory: Talking about talking about trees, *in* 'Proceedings of ACL 1983', pp. 129–136.

Marcus, M. P., Santorini, B. & Marcinkiewicz, M. A. (1993), Building a large annotated corpus of English: the Penn Treebank, Technical report, University of Pennsylvania.

Maruyama, H. (1990), Structural disambiguation with constraint propagation, *in* 'Proceedings of ACL 1990', Pittsburgh/US, pp. 31–38.

McCawley, J. D. (1968), 'Concerning the base component of a Transformational Grammar', *Foundations of Language* **4**, 243–269.

Melamed, I. D., Satta, G. & Wellington, B. (2004), Generalized Multitext Grammars, *in* 'Proceedings of ACL 2004', Barcelona/ES.

Mel'čuk, I. (1988), *Dependency Syntax: Theory and Practice*, State Univ. Press of New York, Albany/US.

Mel'čuk, I. & Polguère, A. (1987), 'A formal lexicon in the Meaning-Text Theory (or how to do lexica with words)', *Computational Linguistics* **13**(3–4), 261–275.

Menzel, W. (1998), 'Constraint satisfaction for robust parsing of spoken language', *Journal of Experimental and Theoretical Artificial Intelligence* **10**(1), 77–89.

Menzel, W. & Schröder, I. (1998), Decision procedures for dependency parsing using graded constraints, *in* 'Proceedings of the COLING/ACL 1998 Workshop Processing of Dependency-based Grammars', Montréal/CA.

Möhl, M. (2004), 'Modellierung natürlicher Sprache mit Hilfe von Topologischer Dependenzgrammatik'. Fortgeschrittenenpraktikum, Saarland University, http://www.ps.uni-sb.de/ rade/papers/related/Moehl04.pdf.

Montague, R. (1974), The proper treatment of quantification in ordinary English, *in* R. Thomason, ed., 'Formal Philosophy: Selected Papers of Richard Montague', Yale University Press.

Montanari, U. (1970), Networks of constraints: Fundamental properties and application to picture processing, Technical report, Carnegie Mellon University.

Mozart Consortium (2006), 'The Mozart-Oz website'. http://www.mozart-oz.org/.

Narendranath, R. (2004), 'Evaluation of the stochastic extension of a constraint-based dependency parser'. Bachelorarbeit, Saarland University.

Odeh, M. (2004), 'Topologische Dependenzgrammatik fürs Arabische'. Forschungspraktikum, Saarland University.

Owens, J. (1988), *An Introduction to Medieval Arabic Grammatical Theory*, Studies in the History of Language Sciences, 45 edn, John Benjamins.

Panenová, J. (1974), 'On verbal frames in Functional Generative Description', *Prague Bulletin of Mathematical Linguistics* .

Peirce, C. S. (1898), *Reasoning and the Logic of Things: The Cambridge Conference Lectures 1898*, Harvard University Press, Cambridge/US. published 1992.

Pelizzoni, J. & das Gracas Volpe Nunes, M. (2005), N:M mapping in XDG - the case for upgrading groups, *in* 'Proceedings of the International Workshop on Constraint Solving and Language Processing', Sitges/ES.

Penn, G. (1999), A generalized-domain-based approach to serbo-croatian second-position clitic placement, *in* G. Bouma, E. Hinrichs, G.-J. M. Kruijff & R. Oehrle, eds, 'Constraints and Resources in Natural Language Syntax and Semantics', CSLI Publications, Stanford/US, pp. 119–136.

Pierce, B. (2002), *Types and Programming Languages*, MIT Press.

Pierrehumbert, J. (1980), The Phonetics and Phonology of English Intonation, PhD thesis, Massachusetts Institute of Technology, Bloomington/US.

Pollard, C. & Sag, I. A. (1987), *Information-Based Syntax and Semantics. Volume 1: Fundamentals*, CSLI, Stanford/US.

Pollard, C. & Sag, I. A. (1994), *Head-Driven Phrase Structure Grammar*, University of Chicago Press, Chicago/US.

Prevost, S. & Steedman, M. (1994), Information based intonation synthesis, *in* 'Proceedings of the ARPA Workshop on Human Language Technology', Princeton/US.

Pullum, G. K. & Scholz, B. C. (2001), On the distinction between model-theoretic and generative-enumerative syntactic frameworks, *in* P. de Groote, G. Morrill & C. Retoré, eds, 'Logical Aspect of Computational Linguistics: 4th International Conference', Lecture Notes in Artificial Intelligence, Springer, Berlin/DE, pp. 17–43.

Rogers, J. (1996), A model-theoretic framework for theories of syntax, *in* 'Proceedings of ACL 1996'.

Rogers, J. (1998), A descriptive characterization of tree-adjoining languages, *in* 'Proceedings of COLING/ACL 1998', Montréal/CA.

Ross, J. R. (1967), Constraints on Variables in Syntax, PhD thesis, MIT.

Sadock, J. M. (1991), *Autolexical Syntax,* University of Chicago Press.

Saraswat, V. (1993), *Concurrent Constraint Programming*, MIT Press.

Sarkar, A. (2000), Practical experiments in parsing using Tree Adjoining Grammars, *in* 'Proceedings of TAG+5', Paris/FR.

Schulte, C. (1997), Oz Explorer: A visual constraint programming tool, *in* L. Naish, ed., 'Proceedings of the Fourteenth International Conference on Logic Programming', MIT Press, Leuven/BE, pp. 286–300.

Schulte, C. (2002), *Programming Constraint Services*, Vol. 2302 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

Schulte, C. & Stuckey, P. J. (2004), Speeding up constraint propagation, *in* 'Tenth International Conference on Principles and Practice of Constraint Programming', Vol. 3258 of *Lecture Notes in Computer Science*, Springer-Verlag, Toronto/CA, pp. 619–633.

Sgall, P., Hajicova, E. & Panevova, J. (1986), *The Meaning of the Sentence in its Semantic and Pragmatic Aspects*, D. Reidel, Dordrecht/NL.

Shanker, V. K. & Weir, D. (1994), 'The equivalence of four extensions of context-free grammars', *Mathematical Systems Theory* **27**(6), 511–546.

Shieber, S. M. (1984), The design of a computer language for linguistic information, *in* 'Proceedings of COLING 1984', pp. 362–366.

Shieber, S. M. (1985), 'Evidence against the context-freeness of natural language', *Linguistics and Philosophy* **8**, 334–343.

Shieber, S. M. & Schabes, Y. (1990), Synchronous Tree Adjoining Grammars, *in* 'Proceedings of COLING 1990', Helsinki/FI.

Smolka, G. (1995), The Oz programming model, *in* J. van Leeuwen, ed., 'Computer Science Today', Lecture Notes in Computer Science, vol. 1000, Springer-Verlag, Berlin/DE, pp. 324–343.

Smolka, G. & Uszkoreit, H. (1996–2001), 'NEGRA project of the collaborative research centre (SFB) 378'. Saarland University.

Steedman, M. (2000*a*), 'Information structure and the syntax-phonology interface', *Linguistic Inquiry* **31**(4), 649–689.

Steedman, M. (2000*b*), *The Syntactic Process*, MIT Press, Cambridge/US.

Steele, S. M. (1978), Word order variation: A typological study, *in* J. Greenberg, ed., 'Universals of Human Language', Stanford University Press, Stanford/US, pp. 585–624.

Stys, M. & Zemke, S. (1995), Incorporating discourse aspects in English-polish MT: Towards robust implementation, *in* 'Recent Advances in NLP', Velingrad/BG.

Sutherland, I. E. (1963), Sketchpad: A man-machine graphical communication system, *in* E. C. Johnson, ed., 'Proceedings of the 1963 Spring Joint Computer Conference', Vol. 23 of *AFIPS Conference Proceedings*, American Federation of Information Processing Societies, Spartan Books, Baltimore/US, pp. 329–346.

Tack, G. (2002), IOzSeF - the integrated Oz search factory, Technical report, Saarland University.

Tesnière, L. (1959), *Eléments de Syntaxe Structurale*, Klincksiek, Paris/FR.

Trautwein, M. (1995), The complexity of structure sharing in unification-based Grammars, *in* W. Daelemans, G. Durieux & S. Gillis, eds, 'Computational Linguistics in the Netherlands 1995', pp. 165–179.

Valin, R. D. V. & LaPolla, R. (1997), *Syntax: Structure, Meaning and Function*, Cambridge University Press.

Wallace, M. (1996), 'Practical applications of constraint programming', *Constraints Journal* **1**(1).

Waltz, D. L. (1975), Understanding the line drawings of scenes with shadows, *in* P. Winston, ed., 'The Psychology of Computer Vision', McGraw-Hill.

Wechsler, S. (1995), The Semantic Basis of Argument Structure, PhD thesis, University of Chicago.

Weir, D. J. (1988), Characterizing Mildly Context-Sensitive Grammar Formalisms, PhD thesis, University of Pennsylvania.

White, M. (2004), Reining in CCG chart realization, *in* 'Proceedings of the 3rd International Conference on Natural Language Generation'.

XTAG Research Group (2001), A Lexicalized Tree Adjoining Grammar for English, Technical Report IRCS-01-03, IRCS, University of Pennsylvania.

Index