Manual of the XDG Development Kit

Ralph Debusmann and Denys Duchier and Jorge Marques Pelizzoni and Jochen Setz

28 December 2007

The XDG Development Kit (XDK) is a grammar development environment for the meta grammar formalism of Extensible Dependency Grammar (XDG). It was written by Ralph Debusmann (`rade@ps.uni-sb.de`), Denys Duchier (`duchier@ps.uni-sb.de`), Jorge Marques Pelizzoni (`jpeliz@icmc.usp.br`) and Jochen Setz (`info@jochensetz.de`). Contributors to XDG and the XDK are: Regine Bader, Ondrej Bojar, Vineet Chaitanya, Benoit Crabbe, Simone Debusmann, Christine Foeldesi, Ciprian Gerstenberger, Robert Grabowski, Alexander Koller, Christian Korthals, Geert-Jan Kruijff, Vladislav Kubon, Marco Kuhlmann, Pierre Lison, Mathias Moehl, Joachim Niehren, Marwan Odeh, Mark Pedersen, Ulrich Pfeiffer, Martin Platek, Oana Postolache, Gert Smolka, Jochen Steigner, Guido Tack, Stefan Thater and Maarika Traat.

Table of Contents

1 License

CeCILL FREE SOFTWARE LICENSE AGREEMENT

Notice

This Agreement is a Free Software license agreement that is the result of discussions between its authors in order to ensure compliance with the two main principles guiding its drafting:

* firstly, compliance with the principles governing the distribution of Free Software: access to source code, broad rights granted to users, * secondly, the election of a governing law, French law, with which it is conformant, both as regards the law of torts and intellectual property law, and the protection that it offers to both authors and holders of the economic rights over software.

The authors of the CeCILL (for Ce[a] C[nrs] I[nria] L[ogiciel] L[ibre]) license are:

Commissariat l'Energie Atomique - CEA, a public scientific, technical and industrial research establishment, having its principal place of business at 25 rue Leblanc, immeuble Le Ponant D, 75015 Paris, France.

Centre National de la Recherche Scientifique - CNRS, a public scientific and technological establishment, having its principal place of business at 3 rue Michel-Ange, 75794 Paris cedex 16, France.

Institut National de Recherche en Informatique et en Automatique - INRIA, a public scientific and technological establishment, having its principal place of business at Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay cedex, France.

Preamble

The purpose of this Free Software license agreement is to grant users the right to modify and redistribute the software governed by this license within the framework of an open source distribution model.

The exercising of these rights is conditional upon certain obligations for users so as to preserve this status for all subsequent redistributions.

In consideration of access to the source code and the rights to copy, modify and redistribute granted by the license, users are provided only with a limited warranty and the software's author, the holder of the economic rights, and the successive licensors only have limited liability.

In this respect, the risks associated with loading, using, modifying and/or developing or reproducing the software by the user are brought to the user's attention, given its Free Software status, which may make it complicated to use, with the result that its use is reserved for developers and experienced professionals having in-depth computer knowledge. Users are therefore encouraged to load and test the suitability of the software as regards their requirements in conditions enabling the security of their systems and/or data to be ensured and, more generally, to use and operate it in the same conditions of security. This Agreement may be freely reproduced and published, provided it is not altered, and that no provisions are either added or removed herefrom.

This Agreement may apply to any or all software for which the holder of the economic rights decides to submit the use thereof to its provisions.

Article 1 - DEFINITIONS

For the purpose of this Agreement, when the following expressions commence with a capital letter, they shall have the following meaning:

Agreement: means this license agreement, and its possible subsequent versions and annexes.

Software: means the software in its Object Code and/or Source Code form and, where applicable, its documentation, "as is" when the Licensee accepts the Agreement.

Initial Software: means the Software in its Source Code and possibly its Object Code form and, where applicable, its documentation, "as is" when it is first distributed under the terms and conditions of the Agreement.

Modified Software: means the Software modified by at least one Contribution.

Source Code: means all the Software's instructions and program lines to which access is required so as to modify the Software.

Object Code: means the binary files originating from the compilation of the Source Code.

Holder: means the holder(s) of the economic rights over the Initial Software.

Licensee: means the Software user(s) having accepted the Agreement.

Contributor: means a Licensee having made at least one Contribution.

Licensor: means the Holder, or any other individual or legal entity, who distributes the Software under the Agreement.

Contribution: means any or all modifications, corrections, translations, adaptations and/or new functions integrated into the Software by any or all Contributors, as well as any or all Internal Modules.

Module: means a set of sources files including their documentation that enables supplementary functions or services in addition to those offered by the Software.

External Module: means any or all Modules, not derived from the Software, so that this Module and the Software run in separate address spaces, with one calling the other when they are run.

Internal Module: means any or all Module, connected to the Software so that they both execute in the same address space.

GNU GPL: means the GNU General Public License version 2 or any subsequent version, as published by the Free Software Foundation Inc.

Parties: mean both the Licensee and the Licensor.

These expressions may be used both in singular and plural form.

Article 2 - PURPOSE

The purpose of the Agreement is the grant by the Licensor to the Licensee of a non-exclusive, transferable and worldwide license for the Software as set forth in Article 5 hereinafter for the whole term of the protection granted by the rights over said Software.

Article 3 - ACCEPTANCE

3.1 The Licensee shall be deemed as having accepted the terms and conditions of this Agreement upon the occurrence of the first of the following events:

* (i) loading the Software by any or all means, notably, by downloading from a remote server, or by loading from a physical medium; * (ii) the first time the Licensee exercises any of the rights granted hereunder.

3.2 One copy of the Agreement, containing a notice relating to the characteristics of the Software, to the limited warranty, and to the fact that its use is restricted to experienced users has been provided to the Licensee prior to its acceptance as set forth in Article 3.1 hereinabove, and the Licensee hereby acknowledges that it has read and understood it.

Article 4 - EFFECTIVE DATE AND TERM

4.1 EFFECTIVE DATE

The Agreement shall become effective on the date when it is accepted by the Licensee as set forth in Article 3.1.

4.2 TERM

The Agreement shall remain in force for the entire legal term of protection of the economic rights over the Software.

Article 5 - SCOPE OF RIGHTS GRANTED

The Licensor hereby grants to the Licensee, who accepts, the following rights over the Software for any or all use, and for the term of the Agreement, on the basis of the terms and conditions set forth hereinafter.

Besides, if the Licensor owns or comes to own one or more patents protecting all or part of the functions of the Software or of its components, the Licensor undertakes not to enforce the rights granted by these patents against successive Licensees using, exploiting or modifying the Software. If these patents are transferred, the Licensor undertakes to have the transferees subscribe to the obligations set forth in this paragraph.

5.1 RIGHT OF USE

The Licensee is authorized to use the Software, without any limitation as to its fields of application, with it being hereinafter specified that this comprises:

1. permanent or temporary reproduction of all or part of the Software by any or all means and in any or all form.

2. loading, displaying, running, or storing the Software on any or all medium.

3. entitlement to observe, study or test its operation so as to determine the ideas and principles behind any or all constituent elements of said Software. This shall apply when the Licensee carries out any or all loading, displaying, running, transmission or storage operation as regards the Software, that it is entitled to carry out hereunder.

5.2 ENTITLEMENT TO MAKE CONTRIBUTIONS

The right to make Contributions includes the right to translate, adapt, arrange, or make any or all modifications to the Software, and the right to reproduce the resulting software.

The Licensee is authorized to make any or all Contributions to the Software provided that it includes an explicit notice that it is the author of said Contribution and indicates the date of the creation thereof.

5.3 RIGHT OF DISTRIBUTION

In particular, the right of distribution includes the right to publish, transmit and communicate the Software to the general public on any or all medium, and by any or all means, and the right to market, either in consideration of a fee, or free of charge, one or more copies of the Software by any means.

The Licensee is further authorized to distribute copies of the modified or unmodified Software to third parties according to the terms and conditions set forth hereinafter.

5.3.1 DISTRIBUTION OF SOFTWARE WITHOUT MODIFICATION

The Licensee is authorized to distribute true copies of the Software in Source Code or Object Code form, provided that said distribution complies with all the provisions of the Agreement and is accompanied by:

1. a copy of the Agreement,

2. a notice relating to the limitation of both the Licensor's warranty and liability as set forth in Articles 8 and 9,

and that, in the event that only the Object Code of the Software is redistributed, the Licensee allows future Licensees unhindered access to the full Source Code of the Software by indicating how to access it, it being understood that the additional cost of acquiring the Source Code shall not exceed the cost of transferring the data.

5.3.2 DISTRIBUTION OF MODIFIED SOFTWARE

When the Licensee makes a Contribution to the Software, the terms and conditions for the distribution of the resulting Modified Software become subject to all the provisions of this Agreement.

The Licensee is authorized to distribute the Modified Software, in source code or object code form, provided that said distribution complies with all the provisions of the Agreement and is accompanied by:

1. a copy of the Agreement,

2. a notice relating to the limitation of both the Licensor's warranty and liability as set forth in Articles 8 and 9,

and that, in the event that only the object code of the Modified Software is redistributed, the Licensee allows future Licensees unhindered access to the full source code of the Modified Software by indicating how to access it, it being understood that the additional cost of acquiring the source code shall not exceed the cost of transferring the data.

5.3.3 DISTRIBUTION OF EXTERNAL MODULES

When the Licensee has developed an External Module, the terms and conditions of this Agreement do not apply to said External Module, that may be distributed under a separate license agreement.

5.3.4 COMPATIBILITY WITH THE GNU GPL

The Licensee can include a code that is subject to the provisions of one of the versions of the GNU GPL in the Modified or unmodified Software, and distribute that entire code under the terms of the same version of the GNU GPL.

The Licensee can include the Modified or unmodified Software in a code that is subject to the provisions of one of the versions of the GNU GPL, and distribute that entire code under the terms of the same version of the GNU GPL.

Article 6 - INTELLECTUAL PROPERTY

6.1 OVER THE INITIAL SOFTWARE

The Holder owns the economic rights over the Initial Software. Any or all use of the Initial Software is subject to compliance with the terms and conditions under which the Holder has elected to distribute its work and no one shall be entitled to modify the terms and conditions for the distribution of said Initial Software.

The Holder undertakes that the Initial Software will remain ruled at least by this Agreement, for the duration set forth in Article 4.2.

6.2 OVER THE CONTRIBUTIONS

The Licensee who develops a Contribution is the owner of the intellectual property rights over this Contribution as defined by applicable law.

6.3 OVER THE EXTERNAL MODULES

The Licensee who develops an External Module is the owner of the intellectual property rights over this External Module as defined by applicable law and is free to choose the type of agreement that shall govern its distribution.

6.4 JOINT PROVISIONS

The Licensee expressly undertakes:

1. not to remove, or modify, in any manner, the intellectual property notices attached to the Software;

2. to reproduce said notices, in an identical manner, in the copies of the Software modified or not.

The Licensee undertakes not to directly or indirectly infringe the intellectual property rights of the Holder and/or Contributors on the Software and to take, where applicable, vis--vis its staff, any and all measures required to ensure respect of said intellectual property rights of the Holder and/or Contributors.

Article 7 - RELATED SERVICES

7.1 Under no circumstances shall the Agreement oblige the Licensor to provide technical assistance or maintenance services for the Software.

However, the Licensor is entitled to offer this type of services. The terms and conditions of such technical assistance, and/or such maintenance, shall be set forth in a separate instrument. Only the Licensor offering said maintenance and/or technical assistance services shall incur liability therefor.

7.2 Similarly, any Licensor is entitled to offer to its licensees, under its sole responsibility, a warranty, that shall only be binding upon itself, for the redistribution of the Software and/or the Modified Software, under terms and conditions that it is free to decide. Said warranty, and the financial terms and conditions of its application, shall be subject of a separate instrument executed between the Licensor and the Licensee.

Article 8 - LIABILITY

8.1 Subject to the provisions of Article 8.2, the Licensee shall be entitled to claim compensation for any direct loss it may have suffered from the Software as a result of a fault on the part of the relevant Licensor, subject to providing evidence thereof.

8.2 The Licensor's liability is limited to the commitments made under this Agreement and shall not be incurred as a result of in particular: (i) loss due the Licensee's total or partial failure to fulfill its obligations, (ii) direct or consequential loss that is suffered by the Licensee due to the use or performance of the Software, and (iii) more generally, any consequential loss. In particular the Parties expressly agree that any or all pecuniary or business loss (i.e. loss of data, loss of profits, operating loss, loss of customers or orders, opportunity cost, any disturbance to business activities) or any or all legal proceedings instituted against the Licensee by a third party, shall constitute consequential loss and shall not provide entitlement to any or all compensation from the Licensor.

Article 9 - WARRANTY

9.1 The Licensee acknowledges that the scientific and technical state-of-the-art when the Software was distributed did not enable all possible uses to be tested and verified, nor for the presence of possible defects to be detected. In this respect, the Licensee's attention has been drawn to the risks associated with loading, using, modifying and/or developing and reproducing the Software which are reserved for experienced users.

The Licensee shall be responsible for verifying, by any or all means, the suitability of the product for its requirements, its good working order, and for ensuring that it shall not cause damage to either persons or properties.

9.2 The Licensor hereby represents, in good faith, that it is entitled to grant all the rights over the Software (including in particular the rights set forth in Article 5).

9.3 The Licensee acknowledges that the Software is supplied "as is" by the Licensor without any other express or tacit warranty, other than that provided for in Article 9.2 and, in particular, without any warranty as to its commercial value, its secured, safe, innovative or relevant nature.

Specifically, the Licensor does not warrant that the Software is free from any error, that it will operate without interruption, that it will be compatible with the Licensee's own equipment and software configuration, nor that it will meet the Licensee's requirements.

9.4 The Licensor does not either expressly or tacitly warrant that the Software does not infringe any third party intellectual property right relating to a patent, software or any other property right. Therefore, the Licensor disclaims any and all liability towards the Licensee arising out of any or all proceedings for infringement that may be instituted in respect of the use, modification and redistribution of the Software. Nevertheless, should such proceedings be instituted against the Licensee, the Licensor shall provide it with technical and legal assistance for its defense. Such technical and legal assistance shall be decided on a case-by-case basis between the relevant Licensor and the Licensee pursuant to a memorandum of understanding. The Licensor disclaims any and all liability as regards the Licensee's use of the name of the Software. No warranty is given as regards the existence of prior rights over the name of the Software or as regards the existence of a trademark.

Article 10 - TERMINATION

10.1 In the event of a breach by the Licensee of its obligations hereunder, the Licensor may automatically terminate this Agreement thirty (30) days after notice has been sent to the Licensee and has remained ineffective.

10.2 A Licensee whose Agreement is terminated shall no longer be authorized to use, modify or distribute the Software. However, any licenses that it may have granted prior to termination of the Agreement shall remain valid subject to their having been granted in compliance with the terms and conditions hereof.

Article 11 - MISCELLANEOUS

11.1 EXCUSABLE EVENTS

Neither Party shall be liable for any or all delay, or failure to perform the Agreement, that may be attributable to an event of force majeure, an act of God or an outside cause, such as defective functioning or interruptions of the electricity or telecommunications networks, network paralysis following a virus attack, intervention by government authorities, natural disasters, water damage, earthquakes, fire, explosions, strikes and labor unrest, war, etc.

11.2 Any failure by either Party, on one or more occasions, to invoke one or more of the provisions hereof, shall under no circumstances be interpreted as being a waiver by the interested Party of its right to invoke said provision(s) subsequently.

11.3 The Agreement cancels and replaces any or all previous agreements, whether written or oral, between the Parties and having the same purpose, and constitutes the entirety of the agreement between said Parties concerning said purpose. No supplement or modification to the terms and conditions hereof shall be effective as between the Parties unless it is made in writing and signed by their duly authorized representatives.

11.4 In the event that one or more of the provisions hereof were to conflict with a current or future applicable act or legislative text, said act or legislative text shall prevail, and the Parties shall make the necessary amendments so as to comply with said act or legislative text. All other provisions shall remain effective. Similarly, invalidity of a provision of the Agreement, for any reason whatsoever, shall not cause the Agreement as a whole to be invalid.

11.5 LANGUAGE

The Agreement is drafted in both French and English and both versions are deemed authentic.

Article 12 - NEW VERSIONS OF THE AGREEMENT

12.1 Any person is authorized to duplicate and distribute copies of this Agreement.

12.2 So as to ensure coherence, the wording of this Agreement is protected and may only be modified by the authors of the License, who reserve the right to periodically publish updates or new versions of the Agreement, each with a separate number. These subsequent versions may address new issues encountered by Free Software.

12.3 Any Software distributed under a given version of the Agreement may only be subsequently distributed under the same version of the Agreement or a subsequent version, subject to the provisions of Article 5.3.4.

Article 13 - GOVERNING LAW AND JURISDICTION

13.1 The Agreement is governed by French law. The Parties agree to endeavor to seek an amicable solution to any disagreements or disputes that may arise during the performance of the Agreement.

13.2 Failing an amicable solution within two (2) months as from their occurrence, and unless emergency proceedings are necessary, the disagreements or disputes shall be referred to the Paris Courts having jurisdiction, by the more diligent Party.

Version 2.0 dated 2006-09-05.

2 Overview

The XDG Development Kit (XDK) is a grammar development environment for the meta grammar formalism of Extensible Dependency Grammar (XDG). XDG is described most thoroughly in Ralph Debusmann's dissertation *Extensible Dependency Grammar - A Modular Grammar Formalism Based On Multigraph Description*. The XDK is published in *The XDG Grammar Development Kit* ([References], page 219).

Still more XDG material can be found here: `http://www.ps.uni-sb.de/~rade/xdg.html`. In particular, this page includes pointers to slides, e.g. from the ESSLLI 2004 course by Ralph Debusmann and Denys Duchier: `http://www.ps.uni-sb.de/~rade/talks.html`.

XDG is a meta grammar formalism: it must be instantiated to yield a particular grammar formalism. An *XDG instance* can have any number of *dimensions* and on these dimensions, it can use any number of *principles* which stipulate the well-formedness conditions of an analysis. An *XDG grammar* consists of two parts:

1. a specification of the used XDG instance
2. a lexicon

In the XDK, grammars are written as *XDK grammar files*. Grammar files include both 1. and 2., i.e. the specification of the particular XDG instance used, and a lexicon.

The XDK provides the following:

- compilers for grammar files in various grammar file input languages
- the solver for XDG
- output functors to visualize the solutions of the solver
- the principle compiler *Principle Writer* (Chapter 10 [PrincipleWriter], page 171) for automatic compilation of principles into Mozart/Oz constraint functors
- executable programs exposing the functionality of the system:
  - graphical user interface (`xdk`)
  - standalone grammar file compiler (`xdkc`)
  - standalone grammar file converter (`xdkconv`)
  - standalone solver (`xdks`)
- various useful shell-scripts
- a number of handcrafted example grammars
- a grammar generator to generate XDG grammars from the TAG grammar developed in the XTAG project

The structure of this manual is as follows. Chapter 3 [Installation], page 13 explains how to install the XDK. Chapter 4 [Compiler], page 17 explains how to write grammar files in one of the various grammar file input languages. Chapter 5 [Grammars], page 89 introduces the example grammars provided with the XDK. Chapter 6 [XTAG], page 97 describes the XTAG grammar generator which generates XDG grammars from the TAG grammar developed in the XTAG project. In Chapter 7 [Solver], page 99, we explain the core of the XDK, i.e. the constraint-based solver. Chapter 8 [Oracles], page 145 is about oracles to guide the search for solutions. Chapter 9 [Outputs], page 147 explains the various available outputs. Chapter 10 [PrincipleWriter], page 171 introduces the principle compiler, which can be used to extend the principle library. In Chapter 11 [Programs], page 173, we explain the executable programs provided by the XDK. Chapter 12 [Debug], page 183 illustrates how XDG grammars can be debugged. [References], page 219 contains pointers to relevant papers, and [Concept index], page 223 is the concept index of this document.

Throughout this manual, sections labeled “developers only” contain information useful only for people who wish to understand the inner workings of the XDK. Here, in Chapter 13 [Directories], page 185, we describe the directory structure of the XDK. In Chapter 14 [Exceptions], page 187 we specify the standard format for exceptions throughout the system. In Chapter 15 [Variable names], page 189, we describe the variable name conventions.

3 Installation

The newest version of the XDK is available at this URL:
`http://www.ps.uni-sb.de/~rade/mogul/publish/doc/debusmann-xdk/`.

The XDK is written in Mozart-Oz. The current version requires *Mozart-Oz 1.3.2*, which can be downloaded from the Mozart-Oz homepage at URL `http://www.mozart-oz.org/`. Mozart-Oz is available for all popular platforms, including Windows, Linux, and MacOS X.

3.1 Installing a binary release

To install a binary release, you need to do the following:

1. install Mozart
2. extract the archive

3.2 Installing a source release

To install a source release (archived in an *ozmake* package), you need to do the following:

1. install Mozart
2. (Windows only) install Cygwin.[1] Select at least the packages containing the C/C++ compiler "gcc"
3. (Windows only, optional) install Emacs[2]
4. extract the XDK package into the current directory: `ozmake --extract -p debusmann-xdk.pkg`
5. change directory to the XDK: `cd debusmann-xdk`
6. prepare installation[3] `sh scripts/prepinstall`
7. compile the XDK: `ozmake`

Optionally, you can now install the XDK globally (into `~/.oz`) by calling `ozmake -i` (first install) or `ozmake -U` (updating your already installed XDK).

Also optionally, you can add the path to the scripts to your path (e.g. in your `~/.bashrc` if you use bash). If installed locally, that path is `scripts/` (relative to the XDK directory), and if installed globally, `~/.oz/1.3.2/cache/x-ozlib/debusmann/xdk/scripts/`. In the latter case, you need to set the executable bit for the scripts:

```
chmod u+x ~/.oz/1.3.2/cache/x-ozlib/debusmann/xdk/scripts/*
```

3.3 Using the Emacs mode for User Language files

We provide an Emacs mode which implements syntax highlighting for User Language files.

---

[1] *Cygwin* is a Linux-like environment which you can obtain at URL `http://www.cygwin.com/`.

[2] *Emacs* is a powerful editor available as an additional package at URL `http://www.mozart-oz.org/`. It is well integrated into Mozart-Oz and is also very handy for using the XDK (e.g. you can use the UL Emacs mode).

[3] This sets the user executable bit for all the provided scripts.

3.3.1 Manual invocation

The Emacs mode can be invoked manually as follows:

1. M-x load-file (press ALT and x, then type "load-file")
2. select the file ul.el (in the XDK top directory if XDK is locally installed, otherwise ~/.oz/1.3.2/cache/x-ozlib/debusmann/xdk/ul.el
3. M-x ul-mode

3.3.2 Automatic invocation

The Emacs mode can be invoked automatically upon the launch of Emacs by adding the line:

```
(load-file "<path to ul.el>")
```

to your emacs configuration file (~/.emacs). <path to ul.el> corresponds to the path to the file ul.el.

3.4 Optional installation bits

3.4.1 IOzSeF

To use Guido Tack's *IOzSeF* exploration tool instead of the *Explorer* for the GUI, you need to install the appropriate package available in MOGUL at URL http://www.mozart-oz.org/mogul/info/tack/iozsef.html, and also the package *TkTreeWidget* available at http://www.mozart-oz.org/mogul/info/tack/TkTreeWidget.html. For your convenience, we have also included the two packages in the XDK distribution: tack-iozsef-1.1.pkg and tack-TkTreeWidget-0.7.pkg. To install, type:

```
ozmake -i -p tack-iozsef-1.1.pkg
ozmake -i -p tack-TkTreeWidget-0.7.pkg
```

3.4.2 CLLS output functor

The CLLS output functor requires Joachim Niehren's *DaVinci* package from MOGUL: http://www.mozart-oz.org/mogul/info/niehren/davinci.html, and all the things this package requires in turn for itself.

3.4.3 XTAG grammar generator

The XTAG grammar generator in subdirectory XTAG requires the following files in the directory XTAG/Grammar:

- syntax.flat
- treefams.dat
- treenames.dat
- xtag.trees.dat

The files are taken from the lem parser distribution available here: ftp://ftp.cis.upenn.edu/pub/xtag/lem/lem-0.14.0.tgz. Within the archive, the files can be found in the directory: lem-0.14.0/data/english. For best-fitting comparisons with the lem parser, use the preferences file: xtag.prefs, i.e., put it into its suitable directory, i.e., lem-0.14.0/lib/.

3.4.4 XTAG additional functionality

For the additional XTAG functionality (the function `Compare lem solutions` in `xdk.exe` and the output functor `output.xTAGDerivation`), the lem parser (`ftp://ftp.cis.upenn.edu/pub/xtag/lem/lem-0.14.0.tgz`) must be installed and its executables must be put into the search path.

4 Compiler

This chapter is about the *grammar file compiler* of the XDK. The grammar file compiler transforms a *grammar file* in one of the grammar file input languages (currently: *User Language (UL)* and *XML language*) into a *grammar* in the *Solver Language (SL)* which can be fed into the XDK solver. Basically, compilation of grammar files consists of expanding the lexical entries and encoding them for use in the XDK solver.

This chapter is chiefly about the structure of grammar files. In our explanations, we will often refer to the example grammar `Grammars/Acl01.ul`, which is written in the UL. The example grammar `Grammars/Acl01.xml` is an XML version of the same grammar.

The structure of this chapter is as follows: After an overview in Section 4.1 [Overview1], page 17, we explain how to define and use dimensions in Section 4.2 [Dimensions], page 19, and how to define types in Section 4.3 [Types], page 20. In Section 4.4 [Principles], page 24, we show how to use principles, and in Section 4.5 [Outputs1], page 30 how to use output functors. In Section 4.6 [Lexicon], page 31, we explain how to write the lexicon. In Section 4.7 [Lattices], page 36, we introduce lattices which correspond to the types. Section 4.8 [Merge], page 40 explains how grammar files can be merged.

The next sections are for reference purposes only: In Section 4.9 [Types reference], page 40 we give a list of all types and their lattices. In Section 4.10 [UL syntax], page 47, we present the full syntax of the User Language, and in Section 4.11 [XML syntax], page 53 of the XML language.

Developers only: In Section 4.12 [IL syntax], page 62, we explain the syntax of the Intermediate Language, and in Section 4.13 [SL syntax], page 74 the syntax of the Solver Language. The *grammar record* is the internal representation of a compiled grammar. We explicate it in Section 4.14 [Grammar record], page 79. Finally, we explain the lattice functors corresponding to the lattices, and used in various places in the XDK, in Section 4.15 [Lattice functors], page 83.

4.1 Overview

The purpose of the *grammar file compiler* is to compile *grammar files* in one of the various grammar file input languages into a grammar usable by the XDK solver.

4.1.1 Grammar files

Grammar files serve two purposes:

1. they specify the particular *XDG instance*,
2. they specify the *lexicon*.

For 1., each grammar file defines the *dimensions* used by the instance, and on each of these dimensions, the *principles* and parameters which apply to it. Additionally, the grammar file specifies on each dimension which *outputs* apply to it. The XDK includes libraries of predefined principles (*principle library*) and of predefined outputs (*output library*). These libraries can be extended by new principles and new outputs according to the interfaces described in the developer sections.

For 2., each grammar file defines a number of *lexical classes* and a number of unexpanded *lexical entries*.

4.1.2 Grammar file languages

Grammar files can be written in various languages. In the current version of the XDK, you can choose between the *User Language* (UL), the *XML language*, and the *Intermediate Language (IL)*. We recommend the UL for handcrafted grammar development. The XML language is better suited for electronic processing of grammars. The IL is designed to be the internal language for the grammar file compiler - we do not recommend to write grammars in that language. It is rather hard to read, and, more importantly, if you write your grammars in the IL directly, the grammar file compiler cannot tell you the whereabouts of an error. The grammar file compiler first converts each UL and XML grammar file into an IL grammar file, which can be processed uniformly from then on.

All programs of XDK can open grammar files in the following formats (with the following default file name suffixes):

- the UL (default suffix: `ul`)
- the XML language (`xml`)
- IL *Oz pickles*, (`ilp`)
- IL *Oz functors* exporting the grammar under the key `grammar` (`ozf`)
- SL *Oz pickles* in the *Solver Language (SL)* (`slp`).

4.1.3 Type checking

Grammar files are *statically typed*. The grammar file compiler performs *type checking* on used principles and lexical classes and entries.

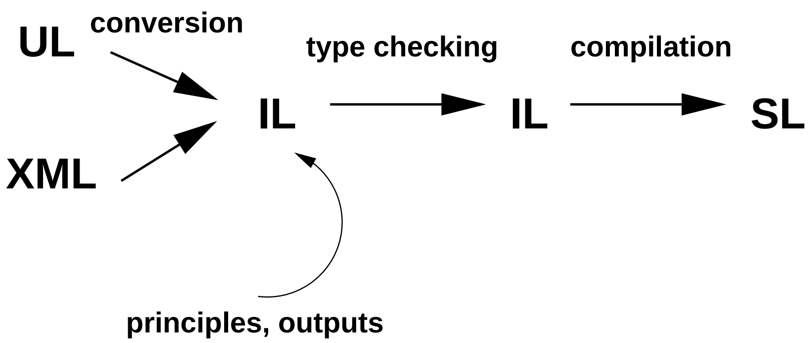
4.1.4 Compilation

After type checking, the grammar file compiler transforms the type checked IL grammar file into a grammar file in the *Solver Language (SL)* which can be fed into the XDK solver. The lexicon of the SL grammar file does not contain lexical classes anymore: it contains only the lexical entries which result from compiling out the lexical entries in the IL grammar file.

An arbitrary number of grammars can be *merged* into one. The prerequisite for merging is that all the grammars must share the same type definitions.

4.1.5 Summary

In the picture below, we summarize of the stages of processing the grammar file compiler performs:



That is:

1. it converts grammar files in either the User Language (UL) or the XML language into an IL grammar file
2. it adds the definitions of the predefined principles and outputs to the IL grammar file
3. it performs type checking on the IL grammar file
4. it compiles the IL grammar file into a SL grammar which can be fed into the XDK solver

4.1.6 Developers only

Internally, the functors `Compiler/UL/Parser.oz` and then `Compiler/UL/2ILConverter.oz` convert UL grammar files into IL grammar files.

The functors `Compiler/XML/Parser.oz` and then `Compiler/XML/2ILConverter.oz` convert XML language grammar files into IL grammar files.

Then, the functor `Compiler/TypeCollector.oz` collects all the types defined in the IL grammar file.

Then, the functor `Compiler/TypeChecker.oz` type-checks the IL grammar file.

Then, the functor `Compiler/Encoder.oz` encodes the IL grammar file using the lattice functors corresponding to the types (defined in `Compiler/Lattices/`). This step yields a stateless *Solver Language (SL) grammar record* which can be saved to disk and loaded again (i.e. *pickled* in Oz terminology).

Finally, the functor `Compiler/Compiler.oz` compiles the stateless SL grammar into a stateful SL grammar record which can be used by the XDK solver.

4.2 Dimensions

4.2.1 Defining dimensions

Each grammar file defines a set of *dimensions*. A dimension definition consists of:

1. type definitions (see Section 4.3 [Types], page 20)
2. a specification of used principles on that dimension (see Section 4.4 [Principles], page 24)
3. a specification of used outputs on that dimension (see Section 4.5 [Outputs1], page 30)

In the UL, a dimension definition is written:

```
defdim <constant> {<dimension definition>}
```

where <constant> is the dimension identifier for the defined dimension.

Notice that the XDK supports a special dimension: `lex`. The `lex` dimension serves to assign a word to each lexical entry, and must be defined in each grammar file. It cannot be turned off - the lexicon is always used.

4.2.2 Using dimensions

From the set of defined dimensions, each grammar file uses a subset. To use a dimension, you write:

```
usedim <constant>
```

where <constant> is the dimension identifier of the used dimension.

4.3 Types

The types used throughout a grammar file are defined inside dimension definitions. Each dimension defines at least three types:

1. its *attributes type*
2. its *entry type*
3. its *label type*

You can define an arbitrary number of additional types.

We give a list of all types in Section 4.9 [Types reference], page 40.

4.3.1 Attributes type

The attributes type is a record specifying the type of the *attributes record* of the currently defined dimension. The attributes record includes a set of additional features for each node on the defined dimension. If you do not provide an attribute type, the grammar file compiler assumes it to be the empty record.

In the UL, the attributes type is defined as follows:

```
defattrstype <type>
```

4.3.2 Entry type

The entry type is a record specifying the type of the *entry record* on the currently defined dimension. The entry record includes all features of a lexical entry on the defined dimension. If you do not provide an entry type, the grammar file compiler assumes it to be the empty record.

In the UL, the entry type is defined as follows:

```
defentrytype <type>
```

4.3.3 Label type

The label type is a domain specifying the type of labels on the currently defined dimension. The labels correspond to the edge labels on the defined dimension. If you do not provide a label type, the grammar file compiler assumes it to be the empty domain.

In the UL, the label type is defined as follows:

```
deflabeltype <type>
```

4.3.4 Additional types

You can define an arbitrary number of additional types, e.g. to ease the construction of more complex types.

In the UL, you define additional types as follows:

```
deftype <constant> <type>
```

where the constant corresponds to the type identifier[1] of the type.

[1] The grammar file compiler considers all identifiers, and therefore also type identifiers, to be global to the grammar file. For instance, you cannot define two types with the same identifier, even if the definitions are contained in different dimension definitions. On the other hand, you can freely access types defined on a different dimension than the currently defined one.

4.3.5 Example (id dimension)

Here are the type definitions on the id dimension of our example grammar file `Grammars/Acl01.ul`:

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% define dimension id
defdim id {
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% define types
  deftype "id.label" {det subj obj vbse vprt vinf prt}
  deftype "id.person" {first second third}
  deftype "id.number" {sg pl}
  deftype "id.gender" {masc fem neut}
  deftype "id.case" {nom gen dat acc}
  deftype "id.def" {def indef undef}
  deftype "id.agr" tuple("id.person" "id.number" "id.gender" "id.case" "id.def")
  deftype "id.agrs" iset("id.agr")

  deflabeltype "id.label"
  defattrstype {agr: "id.agr"}
  defentrytype {in: valency("id.label")
                out: valency("id.label")
                agrs: "id.agrs"
                agree: set("id.label")
                govern: vec("id.label" "id.agrs")}
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% use principles
  ...
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% use and choose outputs
  ...
}
```

`defdim id` indicates the definition of a dimension with identifier `id`.

`deftype "id.label" {det subj obj vbse vprt vinf prt}` defines a type with identifier `id.label`, a domain with consisting of the constants `det`, `subj`, `obj`, `vbse`, `vprt`, `vinf` and `prt`.

`deftype "id.person" {first second third}` defines a type with identifier `id.person`, a *domain* consisting of the constants `first`, `second` and `third`.

`deftype "id.number" {sg pl}` defines a type with identifier `id.number`, a domain consisting of the constants `sg` and `pl`.

`deftype "id.gender" {masc fem neut}` defines a type with identifier `id.gender`, a domain consisting of the constants `masc`, `fem` and `neut`.

`deftype "id.case" {nom gen dat acc}` defines a type with identifier `id.case`, a domain consisting of the constants `nom`, `gen`, `dat` and `acc`.

`deftype "id.def" {def indef undef}` defines a type with identifier `id.def`, a domain consisting of the constants `def`, `indef` and `undef`.

`deftype "id.agr" tuple("id.person" "id.number" "id.gender" "id.case" "id.def")` defines a type with identifier `id.agr`, a *tuple* with the projections `"id.person"`, `"id.number"`, `"id.gender"`, `"id.case"`, and `"id.def"`.

`deftype "id.agrs" iset("id.agr")` defines a type with identifier `id.agrs`, an *intersective set* with domain `id.agr`.

`deflabeltype "id.label"` states that the label type is *type reference* to the type with identifier `id.label`.

`defattrstype {agr: "id.agr"}` states that the attributes type is a *record* with field `agr` of type `id.agr`.

The lines starting with `defentrytype` define the entry type, a record with fields `in`, `out`, `agrs`, `agree` and `govern`. `in` has type `valency("id.label")`, a *valency* with domain `"id.label"`. `out` has type `valency("id.label")`, a valency with domain `"id.label"`. `agrs` has type `"id.agrs"`. `agree` is an *accumulative set* with domain `"id.label"`. `govern` has type `vec("id.label" "id.agrs")`, a *map* with domain `"id.label"` and co-domain `"id.agrs"`.

4.3.6 Example (lp dimension)

Here are the type definitions on the lp dimension of our example grammar file `Grammars/Acl01.ul`:

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% define dimension lp
defdim lp {
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%
  %% define types
  deftype "lp.label" {d df n mf vcf p pf v vxf}
  deflabeltype "lp.label"
  defentrytype {in: valency("lp.label")
                out: valency("lp.label")
                on: iset("lp.label")}
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%
  %% use principles
  ...
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%
  %% use and choose outputs
  ...
}
```

`defdim lp` indicates the definition of a dimension named `lp`.

`deftype "lp.label" {d df n mf vcf p pf v vxf}` defines a type with identifier `lp.label`, a domain with consisting of the constants `d`, `df`, `n`, `mf`, `vcf`, `p`, `pf`, `v` and `vxf`.

`deflabeltype "lp.label"` states that the label type is a reference to `lp.label`.

We omit the definition of the attributes type on the lp dimension, thus the grammar file compiler assumes it to be the empty record.

The lines starting with `defentrytype` define the entry type, a record with fields `in`, `out` and `on`. `in` has type `valency("lp.label")`, a valency with domain `"lp.label"`.

out has type valency("lp.label"), a valency with domain "lp.label". on has type iset("lp.label"), an intersective set with domain "lp.label".

4.3.7 Example (idlp dimension)

Here are the type definitions on the idlp dimension of our example grammar file Grammars/Acl01.ul:

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% define dimension idlp
defdim idlp {
  defentrytype {blocks: set("id.label")
                end: vec("lp.label" set("id.label"))}
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% use principles
  ...
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% use and choose outputs
  ...
}
```

defdim idlp indicates the definition of a dimension named idlp.

The lines starting with defentrytype define the entry type, a record with fields blocks and link. blocks has type set("id.label"), an *accumulative set* with domain id.label.[2] link has type vec("lp.label" set("id.label"), a map with domain lp.label and co-domain set("id.label").

4.3.8 Example (lex dimension)

Here are the type definitions on the *lex dimension* of our example grammar file Grammars/Acl01.ul:

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% define dimension lex
defdim lex {
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% define types
  defentrytype {word: string}
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% use principles
  ...
}
```

defentrytype {word: string} states that the entry type is a record with only the field word of type string.

The lex dimension must be defined in each grammar file, and its entry type must at least include the field word of type string. The XDK compiler collects all lexical entries with identical word values on the lex dimension in sets of lexical entries, assigned to this word value.

[2] Note that we refer here to a type which is defined on another dimension (viz. the id dimension).

Notice that the lex dimension is very different from the “full-blown” id and lp dimensions: it serves only to add lexical information such as the word form (and optionally further information) to each lexical entry.

4.4 Principles

Each dimension states which principles it uses, and their parameters. The principles are taken from the predefined *principle library* of the XDK. The principle library includes the definitions of the available principles. Developers can add new principles using the interface described in the developer sections.

To use a principle, you need to specify two mappings:

1. the *dimension mapping*
2. the *argument mapping*

4.4.1 Dimension mapping

The dimension mapping binds *dimension variables* to dimension identifiers. Each principle definition introduces a set of dimension variables which must be bound upon principle use. `Lex` and `This` are special dimension variables, always bound to dimensions `lex` and to the currently defined dimension, respectively.

4.4.2 Argument mapping

The argument mapping binds *argument variables* (or just *arguments* for short) to values. Each principle definition introduces a set of arguments, their types, and (optionally) their default values. Each argument which is not provided upon principle use gets its default value. If it does not have a default value, the XDK grammar file compiler raises an exception.

In the UL, the expression to use a principle is:

```
useprinciple <constant> {dims { <variable_1>:<constant_1>
                                ...
                                <variable_n>:<constant_n> }
                         args { <variable_1>:<term_1>
                                ...
                                <variable_m>:<term_m> }
                         }
```

Here, `<constant>` is the principle identifier. In the `dims` part, you specify the dimension mapping, and in the `args` part, you specify the argument mapping.

In the following, we give a set of example of how principles are used in our example grammar `Grammars/Acl01.ul`. Note that this manual contains detailed descriptions of all principles in the predefined principle library in Section 7.2 [Principles list], page 103.

4.4.3 Example (principle.graph)

Here is how the principle `principle.graph` is used on the id dimension of our example grammar file:

```
useprinciple "principle.graph" {
  dims {D: id}}
```

The identifier of the principle is `principle.graph`. The dimension mapping maps the dimension variable `D` to the dimension identifier `id`. The argument mapping is empty.

4.4.3.1 Graph principle

The `principle.graph` principle posits that the structure on the dimension bound to the dimension variable `D` is a graph[3]. It has no arguments.

In the example, the principle posits that the id dimension is a graph.

4.4.4 Example (principle.tree)

Here is how the principle `principle.tree` is used on the id dimension of our example grammar file:

```
useprinciple "principle.tree" {
  dims {D: id}}
```

The identifier of the principle is `principle.tree`. The dimension mapping maps the dimension variable `D` to the dimension identifier `id`. The argument mapping is empty.

4.4.4.1 Tree principle

The `principle.tree` principle posits that the structure on dimension `D` is a tree. The principle does not have any arguments.

In the example, the principle posits that the id dimension is a tree.

4.4.5 Example (principle.valency)

Here is how the principle `principle.valency` is used on the id dimension of our example grammar file:

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

The identifier of the principle is `principle.valency`. The dimension mapping maps the dimension variable `D` to the dimension identifier `id`. The argument mapping is empty.

4.4.5.1 Valency principle

The `principle.valency` principle constrains the incoming and outgoing edges of each node on dimension `D`. The `In` and `Out` arguments each specify a *valency*. The default values of the `In` and `Out` arguments are the *feature paths* `_.D.entry.in` and `_.D.entry.out`, respectively.

In the example, the values of the `In` and `Out` arguments are not provided, thus the grammar file compiler uses the default values `_.D.entry.in` and `_.D.entry.out`, respectively. That is, for each node v, `In` equals the the value of the field `in` of the entry of v on the id dimension, and `Out` equals the value of the field `out` of the entry of v.

4.4.5.2 Feature paths

Before we proceed, here is a short introduction of *feature paths*. Feature paths are used to access fields in the attributes or in the entry of a node. In the UL, the syntax for a feature path is:

---

[3] Currently, the principle library includes two graph principles: `principle.graph` and `principle.graph1`. The latter leads to faster solving than the former, but it is cannot be used together with several principles of the principle library, and is thus quite obsolete.

```
<root var>.<dim var>.(attrs|entry).<field_1>.....<field_n>
```

Feature paths start with a *root variable* (`<root var>`) which states which node shall be accessed. The root variable is either "up" or "down" (in the UL, this corresponds to `^` or `_`). Each principle can bind "up" and "down" to arbitrary nodes. By convention, "up" means "mother", and "down" means "daughter" (for constraints on edges), and "down" means "myself" (for constraints on nodes). As the valency principle states a constraint on nodes, the root variable `_` in the example `_.D.entry.out` denotes "myself".

The second argument of a feature path is a dimension variable (`<dim var>`) specifying the dimension of the value which is eventually accessed. In the example `_.D.entry.out`, the dimension variable is `D`.

The third argument of a feature path is one of the special constants `attrs` or `entry`. If you choose `attrs`, you access the attributes, and if you choose `entry`, you access the lexical entry. In the example `_.D.entry.out`, the lexical entry is accessed. Thus, the principle is lexicalized.

The fourth argument of a feature path is a list (separated by dots) of fields describing the path to the accessed value. In the example `_.D.entry.out`, the list consists only of the field `out`.

4.4.6 Example (principle.agr)

Here is how the principle `principle.agr` is used on the id dimension of our example grammar file:

```
useprinciple "principle.agr" {
  dims {D: id}
  args {Agr: _.D.attrs.agr
        Agrs: _.D.entry.agrs}}
```

The identifier of the principle is `principle.agr`. The dimension mapping maps the dimension variable `D` to the dimension identifier `id`. The argument mapping maps the argument `Agr` to the feature path `_.D.attrs.agr`, and `Agrs` to `_.D.entry.agrs`.

4.4.6.1 Agr principle

The `principle.agr` principle has the two arguments `Agr` and `Agrs`. The agr principle posits the constraint that for all nodes `Agr` is an element of `Agrs`.

The resulting constraint here is the following: for all nodes v, the value of the node attribute field `agr` is an element of the value of the lexical entry field `agrs`.

4.4.7 Example (principle.agreement)

Here is how the principle `principle.agreement` is used on the id dimension of our example grammar file:

```
useprinciple "principle.agreement" {
  dims {D: id}
  args {Agr1: ^.D.attrs.agr
        Agr2: _.D.entry.agr
        Agree: ^.D.entry.agree}}
```

The identifier of the principle is `principle.agreement`. The dimension mapping maps the dimension variable `D` to the dimension identifier `id`. The argument mapping maps the argument Agr1 to the feature path ^.D.attrs.agr, Agr2 to _.D.attrs.agr, and Agree to ^.D.entry.agree.

4.4.7.1 Agreement principle

The principle.agreement principle has the three arguments Agr1, Agr2 and Agree. It posits the constraint that for all edges from mother v to daughter v' labeled by l, if l is in the set described by Agree, Agr1 must equal Agr2.

In the example, this constraint amounts to the stipulation that for all edges from mother v to daughter v' labeled by l, if l is in the set lexically specified by the agree feature of the mother on the id dimension (feature path ^.D.entry.agree), then the node attribute agr must be equal for both the mother v (feature path ^.D.attrs.agr and the daughter v' (feature path _.D.attrs.agr.

4.4.8 Example (principle.government)

Here is how the principle principle.government is used on the id dimension of our example grammar file:

```
useprinciple "principle.government" {
  dims {D: id}
  args {Agr2: _.D.attrs.agr
        Govern: ^.D.entry.govern}}
```

The identifier of the principle is principle.agreement. The dimension mapping maps the dimension variable D to the dimension identifier id. The argument mapping maps the argument Agr2 to the feature path _.D.attrs.agr, and Govern to ^.D.entry.govern.

4.4.8.1 Government principle

The principle.government principle has the two arguments Agr2 and Government. It posits the constraint that for all edges from mother v to daughter v' labeled by l, Agr2 must be in the set prescribed by Govern for label l.

In the example, this constraint amounts to the stipulation that for all edges from mother v to daughter v' labeled by l, the value of the agr field of the node attributes of the daughter v' (feature path _.D.attrs.agr) must be in the set of labels prescribed by the field govern of the lexical entry of the mother v (feature path ^.D.entry.govern).

4.4.9 Example (principle.order)

Here is how the principle principle.order is used on the lp dimension of our example grammar file:

```
useprinciple "principle.order" {
  dims {D: lp}
  args {Order: [d df n mf vcf p pf v vxf]
        On: _.D.entry.on
        Yields: true}}
```

Here, the principle identifier is principle.order. The dimension mapping maps dimension variable D to dimension identifier lp. The argument mapping maps the argument Order to the list [d df n mf vcf p pf v vxf], On to the feature path _.D.entry.on, and Yields to true.

4.4.9.1 Order principle

The principle.order principle constrains the linear order of the nodes. In particular, it orders the yields or daughters of each node according to their edge label. The Order argument specifies a total order on a subset of the set of edge labels (as a list). The On argument specifies the set of possible *node labels* for each node used to position it with respect to its daughters. The domain of node labels is the same as the domain of edge labels. The Yields argument can be either true or false, depending on whether for each node, its entire subgraphs shall be ordered (true), or just its daughters (false). If the order principle is used in conjunction with the principle.projectivity, which is most frequently the case, then setting Yields to true is just an optimization for solving, but does not change the number of solutions.

The default value of the Order argument is the empty list. The default value of the On argument is the feature path _.D.entry.on. The default value of the Yields argument is false.

In the example, the total order on the set of edge labels is [d df n mf vcf p pf v vxf], i.e. all daughters with edge label (or the mother with node label) d precede all those with edge label df, and so on. The On argument is set to _.D.entry.on, i.e., for each node v, the set of possible node labels of v equals the value of the field on of the entry of v on the lp dimension. The Yields argument is set to true.

4.4.10 Example (principle.projectivity)

Here is how the principle principle.projectivity is used on the lp dimension of our example grammar file:

```
useprinciple "principle.projectivity" {
  dims {D: lp}}
```

Here, the principle identifier is principle.projectivity. The dimension mapping maps dimension variable D to dimension identifier lp. The argument mapping is empty.

4.4.10.1 Projectivity principle

The principle.projectivity principle constrains the analysis on the dimension D to be projective.

In the example, the principle is used on the lp dimension.

4.4.11 Example (principle.climbing)

Here is how the principle principle.climbing is used on the idlp dimension of our example grammar file:

```
useprinciple "principle.climbing" {
  dims {D1: lp
        D2: id}}
```

Here, the principle identifier is principle.climbing. The dimension mapping maps dimension variable D1 to dimension identifier lp, and dimension variable D2 to dimension id. The argument mapping is empty.

4.4.11.1 Climbing principle

The `principle.climbing` principle posits that the graph on dimension `D1` must be flattening of the graph on dimension `D2`. It is called "climbing" since this flattening is the result of nodes "climbing up" metaphorically from the deep dimension `D2` to the flat dimension `D1`. The principle introduces two arguments. The `Subgraphs` argument is either `true` or `false`, depending on whether each node is required to take its entire subgraph along when migrating upwards (`true`), or not (`false`). The `MotherCards` argument specifies whether for each node, the cardinalities of the sets of mothers on `D1` and `D2` must be equal (`true`), or not (`false`). This is an optimization for the case that both `D1` and `D2` are trees. If any of the two is not a tree, `MotherCards` should be set to `false`.

The default value of `Subgraphs` and `MotherCards` is `true`.

In the example, the principle posits that the graph on the lp dimension must be a flattening of the graph on the id dimension. By default, `Subgraphs` and `MotherCards` are set to `true`, i.e. each node must take its entire subgraph along when climbing up, and the cardinalities of its sets of mothers on the id and lp dimensions are equal.

4.4.12 Example (principle.barriers)

Here is how the principle `principle.barriers` is used on the idlp dimension of our example grammar file:

```
useprinciple "principle.barriers" {
 dims {D1: lp
       D2: id
       D3: idlp}}
```

Here, the principle identifier is `principle.barriers`. The dimension mapping maps dimension variable `D1` to dimension identifier `lp`, `D2` to dimension `id` and `D3` to `idlp`. The argument mapping binds the argument variable `Blocks` to the feature path `_.D3.entry.blocks`.

4.4.12.1 Barriers principle

The `principle.barriers` principle is a specialization of the climbing principle. Its purpose is to "block" nodes in the "deep" dimension `D2` from climbing up and appearing higher up in the "flat" dimension `D1`. The principle introduces an argument `Blocks` whose default value is `_.D3.entry.blocks`. The value of `Blocks` is a set edge labels on the "deep" dimension `D2`. For each node v, all nodes below v which have one of these incoming edge labels on the "deep" dimension `D2` must stay below v on the "flat" dimension `D1`.

In the example, the "flat" dimension is the lp dimension, and the "deep" dimension is the id dimension. The value of the `Blocks` argument is lexicalized: for each node v, it equals the value of the `blocks` field of the entry of v on the idlp dimension.

4.4.13 Example (principle.linkingEnd)

Here is how the principle `principle.linkingEnd` is used on the idlp dimension of our example grammar file:

```
useprinciple "principle.linkingEnd" {
 dims {D1: lp
       D2: id}}
```

Here, the principle identifier is `principle.linkingEnd`. The dimension mapping maps dimension variable `D1` to dimension identifier `lp`, and dimension variable `D2` to dimension `id`. The argument mapping is empty.

4.4.13.1 LinkingEnd principle

The `principle.linkingEnd` principle constrains all outgoing edges from node $v1$ to node $v2$ labeled l on dimension `D1` with respect to the incoming edge label l' of $v2$ on dimension `D2`. The principle introduces the argument `End` whose value is a function from the set of edge labels on `D1` to sets of edge labels on `D2`. The default value of the `End` is `^.D3.entry.end`.

An edge $v1$ to node $v2$ labeled l on dimension `D1` is only licensed if the incoming edge label l' of $v2$ on dimension `D2` is an element of the set specified by the applying the function in the `End` argument to label l.

In the example, the `End` argument is not provided. By default, it is lexicalized and equals for each node v_1 the value of the `end` field of the entry of v_1 on the idlp dimension.

4.4.14 Example (principle.entries)

Here is how the principle `principle.entries` is used on the lex dimension of our example grammar file:

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%
%% use principles
useprinciple "principle.entries" {}
```

Here, the principle identifier is `principle.entries`. The dimension mapping is empty. The argument mapping is also empty.

4.4.14.1 Entries principle

The purpose of the `principle.entries` principle is to ensure that for each node, precisely one lexical entry is selected. If you do not use the entries principle, and there are two identical lexical entries for a word in the input, the XDK solver does not select one of the two. If you do use it, it does select one, i.e. it enumerates all possible lexical entries for a word in the input.

4.5 Outputs

In a dimension definition, you can specify a set of outputs to visualize the solutions on that dimension. The outputs must be taken from the predefined *output library*, of the XDK. Using outputs proceeds in two steps:

1. choose outputs
2. use outputs

4.5.1 Choosing outputs

First, you *choose* the subset of *chosen outputs* from the outputs available in the output library. All of these outputs must be able to visualize the solution on the currently defined dimension.

In the UL, you choose an output as follows:

```
output <constant>
```

where the `<constant>` is an output identifier.

4.5.2 Using outputs

Then, you *use* a subset of the chosen outputs (the *used outputs*) which are actually utilized by the XDK to visualize the individual solutions of a solution for that dimension.

In the UL, you use an output as follows:

```
useoutput <constant>
```

where the `<constant>` is an output identifier.

This rather artificial difference between choosing and using outputs has its roots in the GUI of the XDK. Here, all the chosen outputs are available in the `Outputs` menu (so that you can still use them on demand), but only the used outputs are actually used for visualizing solutions.

4.5.3 Example

In our example grammar file, only the output `output.pretty` is chosen on both the id and the lp dimension (but not used):

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%
%% use and choose outputs
output "output.pretty"
```

The multi-dimensional outputs `output.dags1` and `output.latexs1` are used on the lex dimension, but only the former is chosen:

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%
%% use and choose outputs
output "output.dags1"
output "output.latexs1"
useoutput "output.dags1"
```

4.6 Lexicon

In this section, we explain how to write the *lexicon* of an XDG grammar. The XDG lexicon is a mapping from words to sets of *lexical entries* for that word. Lexical entries can be constructed using a hierarchy of parametrized *lexical classes*.

4.6.1 Disjunction

The grammar file compiler supports a kind of *disjunction* to express lexical generalizations. The idea stems from Marie-Helene Candito's work on *Metagrammar* (*Generating an LTAG out of a Principle-based Hierarchical Representation*, [References], page 219) for LTAG. Disjunction corresponds to Candito's notion of *crossings*. The Metagrammar approach is pursued further by Benoit Crabbe and Denys Duchier (*A Metagrammatical Formalism for Lexicalized TAGs*, *Lexical Classes for Structuring the Lexicon of a TAG*, *Metagrammar Redux*), and Denys Duchier actually had the idea to incorporate crossings under the disguise of disjunction into the XDK grammar file compiler ([References], page 219).

4.6.2 Defining lexical entries

A lexical entry is divided into *entry dimensions* corresponding to the individual used dimensions. The type of an entry dimension for dimension d equals the entry type for d. In the following, we call the value of entry dimension for a dimension d the *d entry*.

Obligatorily, each lexical entry must define the `word` feature on the lex dimension. This is the *key* of the lexical entry.

In the UL, a lexical entry is written as follows:

```
defentry {
  dim d_1 <term_1>
  ...
  dim d_n <term_n> }
```

where `<term_i>` is the value of the entry dimension for dimension `d_i`, i.e. the `d_i` entry (`1<=i<=n`).

4.6.2.1 Example (lexical entry)

Below, we show an example lexical entry. It follows the type definitions of our example grammar file `Grammars/Acl01.ul`:

```
defentry {
  dim id {in: {det}
          out: {}
          agrs: ($ fem & (dat|gen) & sg & def)
          agree: {}
          govern: {det: $ ()
                   subj: $ ()
                   obj: $ ()
                   vbse: $ ()
                   vprt: $ ()
                   vinf: $ ()
                   prt: $ ()}}
  dim lp {in: {df}
          out: {}
          on: {d}}
  dim idlp {blocks: {}
            end: {d: {}
                  df: {}
                  n: {}
                  mf: {}
                  vcf: {}
                  p: {}
                  pf: {}
                  v: {}
                  vxf: {}}
  dim lex {word: "der"}}
```

The id entry sets the `in` field to value `{det}`, i.e. a singleton set containing the constant `det`. It sets the `out` field to value `{}`, i.e. the empty set. The `agrs` field is set to `($ fem & (dat|gen) & sg & def)` which is a *set generator expression*. We explain set generator expressions in Section 4.9 [Types reference], page 40. Suffice to say here that set generator expressions describe sets of tuples of a certain type, using set generator conjunction `&` and set generator disjunction `|`. Here, the set generator expression describes all tuples with feminine gender (`fem`), either dative or genitive case (`(dat|gen)`), singular (`sg`), and definite (`def`).

The agree field is set the empty set, and the govern field to a record which maps each edge label on the id dimension to an empty set generator expression. The empty set generator expression denotes all possible tuples of the corresponding type.

The lp entry sets the in field to value {df}, out to {}, and on to {d}.

The idlp entry sets the blocks field to {}. The link field is set to a record which maps each edge label on the lp dimension to the empty set.

The value of word on the lex dimension is “der”; i.e. the dimension record for the lex dimension sets the key for the entire lexical entry to “der”

4.6.3 Defining lexical classes

Lexical entries can be build more conveniently using *lexical classes*. A lexical class is a lexical entry with the difference that the value of the feature word on the lex dimension does not have to be defined. Instead each lexical class has its unique *class identifier*.

In the UL, a lexical class is written as follows:

```
defclass <constant> {
  dim d_1 <term_1>
  ...
  dim d_n <term_n> }
```

where the constant is the class identifier, and <term_i> is the d_i entry (1<=i<=n).

4.6.3.1 Example (lexical class)

Here is an example lexical class:

```
defclass "det" {
  dim id {in: {det}
          out: {}
          agrs: ($ fem & (dat|gen) & sg & def)
          agree: {}
          govern: {det: $ ()
                   subj: $ ()
                   obj: $ ()
                   vbse: $ ()
                   vprt: $ ()
                   vinf: $ ()
                   prt: $ ()}}
  dim lp {in: {df}
          out: {}
          on: {d}}
  dim idlp {blocks: {}
            end: {d: {}
                  df: {}
                  n: {}
                  mf: {}
                  vcf: {}
                  p: {}
                  pf: {}
```

```
                  v: {}
                  vxf: {}}
      dim lex {word: "der"}}
```

The only difference to the lexical entry above is that the class has the identifier "det" in addition to its key "der".

4.6.3.2 Class parameters

Classes can introduce an arbitrary number of variables called *class parameters*.

In the UL, class parameters are introduced after the class identifier and must begin with an upper case letter:

```
defclass <constant> <variable_1> ... <variable_m> {
  dim d_1 <term_1>
  ...
  dim d_n <term_n> }
```

where `<variable_j>` (`1<=j<=m`) correspond to the class parameters.

4.6.3.3 Example (lexical class with parameters)

Here is an example of a class with class parameters:

```
defclass "det" Word Agrs {
  dim id {in: {det}
          out: {}
          agrs: Agrs
          agree: {}
          govern: {det: $ ()
                   subj: $ ()
                   obj: $ ()
                   vbse: $ ()
                   vprt: $ ()
                   vinf: $ ()
                   prt: $ ()}}
  dim lp {in: {df}
          out: {}
          on: {d}}
  dim idlp {blocks: {}
            link: {d: {}
                   df: {}
                   n: {}
                   mf: {}
                   vcf: {}
                   p: {}
                   pf: {}
                   v: {}
                   vxf: {}}
  dim lex {word: Word}}
```

The lexical class has two parameters, `Word` and `Agrs`. `Word` is the value of the `word` feature on the lex dimension, and `Agrs` is the value of the `agrs` feature on the id dimension.

4.6.4 Using lexical classes

Lexical classes can used to construct other lexical classes or to construct lexical entries. All parameters of a class must be instantiated upon use.

In the UL, a class use is written as follows:

```
useclass <constant> {
  <variable_1> : <term_1>
  ...
  <variable_m> : <term_m> }
```

where the constant is the class identifier, and parameter <variable_j> is bound to <term_j> (1<=j<=m).

Notice that you can omit the `useclass` keyword for convenience.

4.6.4.1 Example (class use)

In the example below, we construct a lexical entry for the word “der” using the lexical class `det` defined above (note that we omit the `useclass` keyword here):

```
defentry {
  "det" {Word: "der"
         Agrs: ($ fem & (dat|gen) & sg & def)}}
```

The resulting lexical entry is identical to the lexical entry in the example given above.

4.6.5 Disjunction

The XDK grammar file compiler supports the use of *disjunction*, as a powerful tool to model lexical generalizations. If a value can be either A or B, you write that down: $A or B$. In the resulting lexicon, the XDK grammar file compiler compiles out all possibilities into separate lexical entries.

In the UL, disjunction is written using the `|` operator.

4.6.5.1 Example (disjunction of set generator expressions)

In the example below, we use disjunction to express that the determiner “der” in German can have three different agreement values:

```
defentry {
  "det" {Word: "der"
        Agrs: (($ masc & nom & sg & def) |
               ($ fem & (dat|gen) & sg & def) |
               ($ gen & pl & def))}}
```

I.e., the agreement value is either `($ masc & nom & sg & def)`, `($ fem & (dat|gen) & sg & def)`, or `($ gen & pl & def)`. In the resulting lexicon, the expression above yields three lexical entries, differing only in the value of their agreement (i.e. the value of the feature `agrs` on the id dimension). Notice that the `|` operator within the second set generator expression (`($ fem & (dat|gen) & sg & def)` stands for set generator disjunction which is a different form of disjunction inside set generators. Set generator disjunction does not yield additional lexical entries.

4.7 Lattices

The XDK grammar file compiler uses lattices to model the notions of *default values* and *conjunction*. Each type corresponds to a *lattice* having a *top value*, a *bottom value*, a *greatest lower bound operation*, and a *least upper bound operation*. The XDK grammar file compiler only utilizes the top value, bottom value, and the greatest lower bound operation; it does not use the least upper bound operation. I.e., lattices are only traversed downwards, not upwards.

4.7.1 Top values

The XDK grammar file compiler sets values which are not provided in a lexical class or lexical to the top value of the type. As we give a complete list of types and their corresponding lattices in Section 4.9 [Types reference], page 40, for the present purposes, it suffices to say that:

- the top value of an intersective set is the full set (containing all elements of the domain of the set)
- the top value of an accumulative set is the empty set
- the top value of a valency is the empty set
- the top value of a record is defined recursively: it is the record where each feature has its appropriate top value

Currently, top values serve two purposes:

1. they provide *default values* for omitted values
2. they are used to make the output of the *pretty* output functor more readable (values with top values are abbreviated to `top`)

In the UL, the top value can be obtained by 1) writing `top`, 2) omitting the value.

4.7.1.1 Example (top values)

We come back to the example lexical entry for "der" in Section 4.6 [Lexicon], page 31 above, repeated below:

```
defentry {
  dim id {in: {det}
          out: {}
          agrs: ($ fem & (dat|gen) & sg & def)
          agree: {}
          govern: {det: $ ()
                   subj: $ ()
                   obj: $ ()
                   vbse: $ ()
                   vprt: $ ()
                   vinf: $ ()
                   prt: $ ()}}
  dim lp {in: {df}
          out: {}
          on: {d}}
  dim idlp {blocks: {}
```

```
                 end: {d: {}
                       df: {}
                       n: {}
                       mf: {}
                       vcf: {}
                       p: {}
                       pf: {}
                       v: {}
                       vxf: {}
       dim lex {word: "der"}}
```

First, we replace all values which are identical to the default values by `top`:

```
defentry {
  dim id {in: {det}
          out: top
          agrs: ($ fem & (dat|gen) & sg & def)
          agree: top
          govern: top}
  dim lp {in: {df}
          out: top
          on: {d}}
  dim idlp {blocks: top
            end: top}
  dim lex {word: "der"}}
```

Next, we remove all these features. What we end up with is a much more succinct lexical entry:

```
defentry {
  dim id {in: {det}
          agrs: ($ fem & (dat|gen) & sg & def)
  dim lp {in: {df}
          on: {d}
  dim lex {word: "der"}}
```

4.7.2 Bottom values

All types have bottom values. As we give a complete list of types and their corresponding lattices in Section 4.9 [Types reference], page 40, it suffices to say here that the bottom value of an accumulative set is the full set (containing all elements of the domain of the set).

Currently, bottom values serve two purposes:

1. for accumulative sets, they can be used as an abbreviation replacing an explicit specification of the full set
2. they are used to make the output of the *pretty* output functor more readable (values with bottom values are abbreviated to `bot`)

In the UL, the bottom value can be obtained by writing `bot`.

4.7.2.1 Example (bottom values)

As an example, consider the following definition of the class `fin`:

```
defclass "fin" Word Agrs {
  dim id {in: {}
          out: {subj}
          agrs: Agrs
          agree: {subj}
          govern: {subj: $ nom}}
  dim lp {in: {}
          out: {mf* vxf?}
          on: {v}}
  dim idlp {blocks: {det subj obj vbse vprt vinf prt}}
  dim lex {word: Word}}
```

The value of `blocks` on the idlp dimension includes all elements of the domain of the set (`{det subj obj vbse vprt vinf prt}`). The type of `blocks` is *accumulative set*; i.e. its bottom value is the full set. Thus, we can replace the value of `blocks` by its bottom value to obtain a more succinct lexical class:

```
defclass "fin" Word Agrs {
  dim id {in: {}
          out: {subj}
          agrs: Agrs
          agree: {subj}
          govern: {subj: $ nom}}
  dim lp {in: {}
          out: {mf* vxf?}
          on: {v}}
  dim idlp {blocks: bot}
  dim lex {word: Word}}
```

4.7.3 Greatest lower bound operation

As we give a complete list of types and their corresponding lattices in Section 4.9 [Types reference], page 40, for the present purposes, it suffices to say here that

- the greatest lower bound of two intersective sets is their intersection
- the greatest lower bound of two accumulative sets is their union

The greatest lower bound operation has only the purpose of specializing values. It can be thought of as what is called *lexical inheritance* in other grammar formalisms.

In the following, we will call the greatest lower bound operation *conjunction*

4.7.3.1 Example (greatest lower bound operation; accumulative sets)

Here is a second example. First, we define the two lexical classes `block_subj` and `block_obj`:

```
defclass "blocks_subj" {
  dim idlp {blocks: {subj}}}
```

```
defclass "blocks_obj" {
  dim idlp {blocks: {obj}}}
```

Next, we use the greatest lower bound operation (conjunction) to combine the two lexical classes:

```
defclass "blocks_subj_obj" {
  "blocks_subj" &
  "blocks_obj"}
```

The type of the `blocks` feature on the idlp dimension is *accumulative set*. Hence in the resulting lexical class, the `blocks` value on the lp dimension is the *union* of the `blocks` value of the class `blocks_subj` (`{subj}`) and the `blocks` value of the class `blocks_obj` (`{obj}`), i.e. `{subj obj}`.

If the type of the feature would be *intersective set*, we would have used *intersection* instead of union.

4.7.3.2 Example (combining conjunction and disjunction)

Conjunction and disjunction can be combined. Here is an example. First, we define five lexical classes:

```
defclass "class_1" { ... }

defclass "class_2" { ... }

defclass "class_3" { ... }

defclass "class_4" { ... }

defclass "class_5" { ... }
```

Next, we combine the classes using both conjunction and disjunction in a `defentry` expression:

```
defentry {
  ("class_1" |
   "class_2" |
   "class_3") &
  ("class_4" |
   "class_5")}
```

The expression defines six lexical entries which we could write less succinctly without disjunction as:

```
defentry {
  "class_1" &
  "class_4"}

defentry {
  "class_1" &
  "class_5"}
```

```
defentry {
  "class_2" &
  "class_4"}

defentry {
  "class_2" &
  "class_5"}

defentry {
  "class_3" &
  "class_4"}

defentry {
  "class_3" &
  "class_5"}
```

4.8 Merge

The XDK grammar file compiler allows to *merge* an arbitrary number of grammars into one. Two grammars g_1 and g_2 can be merged into g_3 if they have the same type definitions.

The merge operation is defined as follows:

- the lexicon of g_3 includes all the lexical entries of g_1 and g_2 (i.e. g_3 maps each word to the union of the set of lexical entries for this word in g_1 and g_2)
- the dimension definitions of g_3 are taken from g_2

If more than two grammars g_i ($1 <= i <= n$, and $n > 2$) are merged, the last grammar in the sequence determines the dimension definitions of the merged grammar.

4.9 Types reference

This section lists all types and their corresponding lattices, including their top values, bottom values, and greatest lower bound operations. *Type synonyms* (`Bool`, `Ints`, `Map`, `Valency`) are included.

4.9.1 Bool

Bool is a type synonym for a Section 4.9.3 [Domain], page 41 including only the two constants `false` and `true`.[4]

4.9.1.1 Example

Here is an example bool type definition:

```
deftype "bool" bool
```

4.9.2 Card

Card is a specialized set of integers used for cardinality sets. Usually, such types do not show up explicitly but are introduced via the type synonym Section 4.9.13 [Valency], page 47.

A cardinality set can be specified in various ways:

[4] `false` is encoded by the integer 1, and `true` by the integer 2 since `false` alphabetically precedes `true`.

- set specification: <constant>#{<integer_1> ... <integer_n>} specifies a set of integers
- interval specification: <constant>#[<integer_1> <integer_2>] specifies the closed interval between the integers <integer_1> and <integer_2>
- <constant> is equivalent to <constant>#{1}
- <constant>! is equivalent to <constant>#{1}
- <constant>? is equivalent to <constant>#{0 1}
- <constant>* is equivalent to <constant>#[0 infty]} (where infty corresponds to "infinity")
- <constant>+ is equivalent to <constant>#[1 infty]}

4.9.2.1 Top value

The set {0}.

4.9.2.2 Bottom value

See domain type (Section 4.9.3 [Domain], page 41).

4.9.2.3 Greatest lower bound operation

The greatest lower bound of two cardinality sets S_1 and S_2 is defined as follows:

- S_2 if S_1 is top
- S_1 if S_2 is top
- S_1 if S_1 and S_2 are equal
- the intersection of S_1 and S_2 otherwise

4.9.2.4 Example

Here is an example card type definition:

```
deftype "card" card
```

4.9.3 Domain

4.9.3.1 Description

A constant from a finite domain of constants.

4.9.3.2 Top value

flat_top (undefined)

4.9.3.3 Bottom value

flat_bot (undefined)

4.9.3.4 Greatest lower bound operation

The greatest lower bound of two values A and B yields:

- B if A is top
- A if B is top

- A if A and B are equal
- bottom otherwise

4.9.3.5 Example

Here is an example domain type definition including the constants `constant_1`, `constant_2` and `constant_3`:

```
deftype "domain" {constant_1 constant_2 constant_3}
```

4.9.4 Integer

An integer.

4.9.4.1 Top value

See domain type (Section 4.9.3 [Domain], page 41).

4.9.4.2 Bottom value

See domain type (Section 4.9.3 [Domain], page 41).

4.9.4.3 Greatest lower bound operation

See domain type (Section 4.9.3 [Domain], page 41).

4.9.4.4 Example

Here is an example of an integer type definition:

```
deftype "integer" int
```

4.9.5 Integers

A set of integers. This is a type synonym for the type `set(int)` (Section 4.9.9 [Set (accumulative)], page 44).

4.9.5.1 Example

Here is an example of an integers type definition:

```
deftype "integers" ints
```

4.9.6 List

A list over a domain of any type.

4.9.6.1 Top value

See domain type (Section 4.9.3 [Domain], page 41).

4.9.6.2 Bottom value

See domain type (Section 4.9.3 [Domain], page 41).

4.9.6.3 Greatest lower bound operation

See domain type (Section 4.9.3 [Domain], page 41).

4.9.6.4 Example

Here is an example list type definition with domain `ref("domain")`:

```
deftype "list" ref("domain")
```

4.9.7 Map

A *map* models a total function from a domain to a co-domain by a record whose arity is the domain, and whose type at all fields is the co-domain. The domain must be a finite domain of constants, while the co-domain can have any type.

This is a type synonym for the Section 4.9.8 [Record], page 43 type:

```
field_1:type,...,field_n:type
```

where `field_1,...,field_n` are the elements of the domain of the function, and `type` is the type of the co-domain.

4.9.7.1 Example

Here is an example map type definition from domain `ref("domain")` to co-domain `ref("codomain")`:

```
deftype "map" vec(ref("domain") ref("codomain"))
```

4.9.8 Record

A record over n features, where each of these features consists of a *field* $field_i$ and a *value* $term_i$ of type $type_i$ ($1 <= i <= n$). We call the set of fields of a record its *arity*.

4.9.8.1 Top value

For each field $field_i$ in the arity of the record, its value is the top value of the corresponding type $type_i$.

4.9.8.2 Bottom value

For each field $field_i$ in the arity of the record, its value is the bottom value of the corresponding type $type_i$.

4.9.8.3 Greatest lower bound operation

Recursively, the greatest lower bound of two records A and B is the record where the value of each field $field_i$ is the result of the greatest lower bound of the value at $field_i$ in record A, and the value $field_i$ in record B.

Notice that you can also use the greatest lower bound operation (conjunction) on the features in a record specification. The conjunction of two features $field_1 : term_1$ and $field_2 : term_2$ is defined as:

- If $field_1$ and $field_2$ are equal, then replace the two features by $field_1 : term$ in the record specification, where $term$ is the conjunction of $term_1$ and $term_2$.
- Otherwise, keep the two features in the record specification.

4.9.8.4 Example

Here is an example record type definition with three fields in its arity (`field_1`, `field_2` and `field_3`). These have types `ref("type_1")`, `ref("type_2")` and `ref("type_3")`, respectively:

```
deftype "record" {field_1: ref("type_1")
                  field_2: ref("type_2")
                  field_3: ref("type_3")}
```

4.9.9 Set (accumulative)

A set over a domain. Different lattices depending on the domain, which can be:

1. a finite domain of constants
2. integer
3. a tuple of which all projections are finite domains of constants

4.9.9.1 Set generator expressions

Values of accumulative sets over tuples of finite domains of constants (b) can be specified using *set generator expressions*. Set generator expressions are explained in the previous section for intersective sets.

4.9.9.2 Top value

1. empty set
2. empty set
3. empty set

4.9.9.3 Bottom value

1. the full set (containing all constants in the domain)
2. the set of all integers
3. the full set (containing all tuples in the domain)

4.9.9.4 Greatest lower bound operation

1. set union
2. set union
3. set union

4.9.9.5 Example

Here is an example accumulative set type definition with domain type `ref("type")`:

```
deftype "set" ref("type")
```

4.9.10 Intersective set

A set over a domain. Different lattices depending on the domain, which can be:

1. a finite domain of constants
2. integer
3. a tuple of which all projections are finite domains of constants

4.9.10.1 Set generator expressions

If the domain of the intersective set is a tuple of which all projections are finite domains of constants (case 2), the set can be specified using a *set generator expression*. Set generator expressions describe sets of tuples over finite domains of constants, using *set generator conjunction* (`&` operator) and *set generator disjunction* (`|` operator).[5] Here is the semantics of set generator conjunction:

- a constant in a set generator conjunction must be in the appropriate projection of all described tuples
- a constant in a set generator disjunction must be in the appropriate projection of at least one described tuple

4.9.10.2 Example (set generator expressions)

As an example from our grammar file `Grammars/Acl01.ul`, consider the set generator expression `($ fem & (dat|gen) & sg & def)`. The corresponding type has identifier `id.agrs`, and corresponds to the type definitions below:

```
deftype "id.person" {first second third}
deftype "id.number" {sg pl}
deftype "id.gender" {masc fem neut}
deftype "id.case" {nom gen dat acc}
deftype "id.def" {def indef undef}
deftype "id.agr" tuple(ref("id.person")
                       ref("id.number")
                       ref("id.gender")
                       ref("id.case")
                       ref("id.def"))
deftype "id.agrs" iset(ref("id.agr"))
```

The set generator expression `($ fem & (dat|gen) & sg & def)` describes the set of all tuples with constant `fem` at the third projection (corresponding to the finite domain `id.gender`), either `dat` or `gen` at the fourth projection (`id.case`), `sg` a the second projection (`id.number`), and `def` at the fifth projection (`id.def`). The first projection (`id.person`) is not specified, i.e. it can be any of the constants in the domain (`first`, `second`, or `third`).

4.9.10.3 Top value

1. the full set (containing all constants in the domain)
2. the set of all integers
3. the full set (containing all tuples in the domain)

4.9.10.4 Bottom value

1. empty set

[5] Set generator conjunction `&` and set generator disjunction `|` are different from conjunction and disjunction in the lexicon. Set generator disjunction is restricted to set generator expressions and set generator disjunction does not lead to an increase of the the number of lexical entries. On the other hand, conjunction and disjunction in the lexicon can be used for all terms, and disjunction leads to an increase of the number of lexical entries.

2. empty set
3. empty set

4.9.10.5 Greatest lower bound operation

1. set intersection
2. set intersection
3. set intersection

4.9.10.6 Example

Here is an example intersective set type definition with domain type `ref("type")`:

```
deftype "iset" ref("type")
```

4.9.11 String

A string.

4.9.11.1 Top value

See domain type (Section 4.9.3 [Domain], page 41).

4.9.11.2 Bottom value

See domain type (Section 4.9.3 [Domain], page 41).

4.9.11.3 Greatest lower bound operation

See domain type (Section 4.9.3 [Domain], page 41).

4.9.11.4 Example

Here is an example of a string type definition:

```
deftype "string" string
```

4.9.12 Tuple

A tuple over n finite domains of constants.

4.9.12.1 Top value

see domain type (Section 4.9.3 [Domain], page 41).

4.9.12.2 Bottom value

see domain type (Section 4.9.3 [Domain], page 41).

4.9.12.3 Greatest lower bound operation

see domain type (Section 4.9.3 [Domain], page 41).

4.9.12.4 Example

Here is an example tuple type definition whose first projection has type `ref("type_1")`. The second projection has type `ref("type_2")` and the third projection type `ref("type_3")`:

```
deftype "tuple" tuple(ref("type_1") ref("type_2") ref("type_3"))
```

4.9.13 Valency

A valency is a record whose arity is a finite domain of constants, and whose type at all fields is a set of integers called *cardinality set*.

This is a type synonym for a Section 4.9.7 [Map], page 43 type with the same domain and co-domain `card`.

4.9.13.1 Example

Here is an example valency type definition with domain type `ref("domain")`:

```
deftype "valency" ref("domain")
```

4.10 UL syntax

In this section, we describe the syntax of *User Language (UL)* grammar files, using the *Extended Backus Naur Form (EBNF)* as defined in the XML specification of the W3C (see `http://www.w3.org/TR/REC-xml#sec-notation`).

4.10.1 UL lexical syntax

In this section, we lay out the lexical syntax of the UL.

4.10.1.1 Keywords

Here are the keywords of the UL:

```
<keyword> ::= args | attrs |
              bool | bot |
              card |
              defattrstype | defclass | defdim | defentry |
              defentrytype | defgrammar | deflabeltype | deftype |
              dim | dims |
              entry |
              infty | int | ints | iset |
              label | list |
              vec |
              output |
              ref |
              set | string |
              top | tuple | tv |
              useclass | usedim | useoutput | useprinciple |
              valency
```

4.10.1.2 Operators

Here are the operators of the UL:

```
<operator> ::= { | } | ( | ) | * | & | '| | '@ | [ | ] | < | > |
               $ | . | :: | _ | ^ | ! | ? | + | # | :
```

4.10.1.3 Identifiers

Identifiers consist of letters and the underscore:

```
<id> ::= [a-zA-Z_]+
```

4.10.1.4 Integers

Integers consist of numbers:

```
<int> ::= [0-9]+
```

4.10.1.5 Strings

Strings can be quoted using single quotes (`<sstring>`), double quotes (`<dstring>`), or guillemet quotes (`<gstring>`). You can freely choose between the different kinds of quotes. Inside the quotes, you can write strings using any characters from the ISO 8859-1 character set. We write `.` for "any character from the ISO 8859-1 character set":

```
<sstring> ::= '.+'

<dstring> ::= ".+"

<gstring> ::= .+
```

4.10.1.6 End of line comments

End of line comments are written using the percent symbol `%`.

4.10.1.7 Balanced comments

Balanced comments start with `/*` and end with `*/`.

4.10.1.8 Includes

Files can be included using the `\input` directive. For example to include the file `Chorus_header.ul`, you write:

```
\include "Chorus_header.ul"
```

4.10.2 UL context-free syntax

In this section, we lay out the context-free syntax of the UL. We write all keywords in lower case, and all non-terminals in upper case letters. We use single quotes to escape the meta characters `(`, `)`, `[`, `]`, `?`, `*`, `+`, `#`, `|`, and `.`.

4.10.2.1 Start symbol (S)

The start symbol of our context-free grammar is `S`:

```
S ::= Defgrammar*
```

4.10.2.2 Grammar definitions (Defgrammar)

Here is the UL Syntax for grammar definitions:

```
Defgrammar ::= defdim Constant { Defdim* }
            |   defclass Constant Constant* { Class }
            |   defentry { Class* }
            |   usedim Constant
```

`defdim Constant { Defdim* }` defines a dimension with identifier `Constant`, and dimension definitions `Defdim*`.

`defclass Constant Constant* { Class }` defines a lexical class with identifier `Constant`, class variables `Constant*`, and class body `Class`.

`defentry { Class* }` defines a lexical entry defined by class bodies `Class*`.

`usedim Constant` uses the dimension with identifier `Constant`.

4.10.2.3 Dimension definitions (Defdim)

Here is the UL syntax for dimension definitions:

```
Defdim ::= defattrstype Type
        |   defentrytype Type
        |   deflabeltype Type
        |   deftype Constant Type
        |   useprinciple Constant { Useprinciple* }
        |   output Constant
        |   useoutput Constant
```

`defattrstype Type` defines the attributes type `Type`.

`defentrytype Type` defines the entry type `Type`.

`deflabeltype Type` defines the label type `Type`.

`deftype Constant Type` defines the type `Type` with identifier `Constant`.

`useprinciple Constant { Useprinciple* }` uses the principle with identifier `Constant` and dimension and argument mappings `Useprinciple*`.

`output Constant` chooses output `Constant`.

`useoutput Constant` uses output `Constant`.

4.10.2.4 Principle use instructions (Useprinciple)

Here is the UL syntax for principle use instructions:

```
Useprinciple ::= dims { VarTermFeat* }
              |   args { VarTermFeat* }
```

`dims { VarTermFeat* }` is the dimension mapping `VarTermFeat*`.

`args { VarTermFeat* }` is the argument mapping `VarTermFeat*`.

4.10.2.5 Types (Type)

This is the UL syntax of types:

```
Type ::= { Constant* }
      |   set '(' Type ')'
      |   iset '(' Type ')'
      |   tuple '(' Type* ')'
      |   list '(' Type ')'
      |   valency '(' Type ')'
      |   { TypeFeat+ }
      |   { : }
      |   vec '(' Type_1 Type_2 ')'
      |   card
      |   int
```

```
          |   ints
          |   string
          |   bool
          |   ref '(' Constant ')'
          |   Constant
          |   label '(' Constant ')'
          |   tv '(' Constant ')'
          |   '(' Type ')'
```

`{ Constant* }` is a finite domain consisting of the constants `Constant*`.

`set '(' Type ')'` is a accumulative set with domain `Type`.

`iset '(' Type ')'` is a intersective set with domain `Type`.

`tuple '(' Type* ')'` is a tuple with projections `Type*`.

`list '(' Type ')'` is a list with domain `Type`.

`valency '(' Type ')'` is a valency with domain `Type`.

`{ TypeFeat+ }` is a record with features `TypeFeat+`.

`{ : }` is the empty record.

`vec '(' Type_1 Type_2 ')'` is a vector with fields `Type_1` and values of type `Type_2`.

`card` is a cardinality set.

`int` is an integer.

`ints` is a set of integers.

`string` is a string.

`bool` is a boolean.

`ref '(' Constant ')'` is a type reference to the type with identifier `Constant`.

`Constant` is a shortcut for `ref '(' Constant ')'`.

`label '(' Constant ')'` is an label reference to the label type on the dimension referred to by dimension variable `Constant`.

`tv '(' Constant ')'` is a type variable.

`'(' Type ')'` encapsulates type `Type`.

4.10.2.6 Class bodies (Class)

Here is the UL syntax of a lexical class body:

```
Class ::= dim Constant Term
       |   useclass Constant
       |   useclass Constant { VarTermFeat* }
       |   Constant
       |   Constant { VarTermFeat* }
       |   Class_1 & Class_2
       |   Class_1 '|' Class_2
       |   '(' Class ')'
```

`dim Constant Term` defines the entry `Term` for the dimension with identifier `Constant`.

`useclass Constant` uses the lexical class with identifier `Constant`.

`Constant` is a shortcut for `useclass Constant`.

`useclass Constant { VarTermFeat* }` uses the lexical class with identifier `Constant` and class parameters `VarTermFeat*`.

`Constant { VarTermFeat* }` is a shortcut for `useclass Constant { VarTermFeat* }`.

`Class & Class` is the conjunction of `Class_1` and `Class_2`.

`Class '|' Class` is the disjunction of `Class_1` and `Class_2`.

`'(' Class ')'` brackets class `Class`.

4.10.2.7 Terms (Term)

Here is the UL syntax of terms:

```
Term ::= Constant
      |   Integer
      |   top
      |   bot
      |   Featurepath
      |   CardFeat
      |   { Term* }
      |   '[' Term* ']'
      |   { Recspec+ }
      |   { : }
      |   $ Setgen
      |   $ '(' ')'
      |   Term :: Type
      |   Term_1 & Term_2
      |   Term_1 '|' Term_2
      |   Term_1 @ Term_2
      |   '<' Term* '>'
      |   '(' Term ')'
```

`Constant` is a constant.

`Integer` is an integer.

`top` is lattice top.

`bot` is lattice bottom.

`Featurepath` is a feature path.

`CardFeat` is a cardinality specification.

`{ Term* }` is a set of the elements `Term*`.

`'[' Term* ']'` is a list of the elements `Term*` (in this order).

`{ Recspec+ }` is a record with specification `Recspec+`.

`{ : }` is the empty record.

`$ Setgen` introduces set generator expression with set generator expression body `Setgen`.

`$ '(' ')'` is the empty set generator expression.

`Term :: Type` is a type annotation of term `Term` with type `Type`.

`Term_1 & Term_2` is the conjunction of `Term_1` and `Term_2`.

`Term_1 '|' Term_2` is the disjunction of `Term_1` and `Term_2`.

Term_1 @ Term_2 is the *concatenation* of Term_1 and Term_2. Concatenation is restricted to strings.

'<' Term* '>' is an order generator specification of a list of elements Term*.

'(' Term ')' brackets term Term.

4.10.2.8 Feature paths (Featurepath)

Here is the UL syntax of feature paths:

```
Featurepath ::= Root '.' Constant '.' Aspect ('.' Constant)+

Root ::= _|^

Aspect ::= attrs|entry
```

Root '.' Constant '.' Aspect ('.' Constant)+ is a feature path with root variable Root, dimension variable Constant, aspect Aspect, and the list fields ('.'Constant)+.

4.10.2.9 Record specifications (Recspec)

Here is the UL syntax of record specifications:

```
Recspec ::= TermFeat
         |   Recspec_1 & Recspec_2
         |   Recspec_1 '|' Recspec_2
         |   '(' Recspec ')'
```

TermFeat is a feature.

Recspec_1 & Recspec_2 is the conjunction of Recspec_1 and Recspec_2.

Recspec_1 '|' Recspec_2 is the disjunction of Recspec_1 and Recspec_2.

'(' Recspec ')' brackets record specification Recspec.

4.10.2.10 Set generator expression bodies (Setgen)

Here is the UL syntax of set generator expression bodies:

```
Setgen ::= Constant
        |   Setgen_1 & Setgen_2
        |   Setgen_1 '|' Setgen_2
        |   '(' Setgen ')'
```

Constant is a constant.

Setgen_1 & Setgen_2 is the conjunction of Setgen_1 and Setgen_2.

Setgen_1 '|' Setgen_2 is the disjunction of Setgen_1 and Setgen_2.

'(' Setgen ')' brackets set generator expression body Setgen.

4.10.2.11 Constants (Constant)

Here is the UL syntax of constants:

```
Constant ::= <id> | <sstring> | <dstring> | <gstring>
```

I.e. a constant is either an identifier (<id>), a single quoted string (<sstring>), a double quoted string (<dstring>), or a guillemot quoted string (<gstring>).

4.10.2.12 Integers (Integer)

Here is the UL syntax of constants:

```
Integer ::= <int> | infty
```

I.e. an integer is either an integer (`<int>`) or the keyword for "infinity" (`infty`).

4.10.2.13 Features (ConstantFeat, TermFeat, VarTermFeat, and CardFeat)

Here is the UL syntax of features:

```
ConstantFeat ::= Constant_1 : Constant_2

TermFeat ::= Constant : Term

VarTermFeat ::= Constant : Term

TypeFeat ::= Constant : Type

CardFeat ::= Constant Card
```

`ConstantFeat` is a feature with field `Constant_1` and value `Constant_2`.

`TermFeat` and `VarTermFeat` are features with field `Constant` and value `Term`.

`TypeFeat` is a feature with field `Constant` and value `Type`.

`CardFeat` is a cardinality specification with field `Constant` and cardinality set `Card`.

4.10.2.14 Cardinality sets (Card)

Here is the UL syntax of cardinality sets:

```
Card ::= !
      |   '?'
      |   '*'
      |   '+'
      |   '#' { Integer* }
      |   '#' '[' Integer_1 Integer_2 ']'
```

`!` is cardinality set $\{0\}$.

`'?'` is the cardinality set $\{0, 1\}$.

`'*'` is the cardinality set $\{0, \ldots, infty\}$ where $infty$ means "infinity".

`'+'` is the cardinality set $\{1, \ldots, infty\}$.

`'#' { Integer* }` is the cardinality set including the integers `Integer*`.

`'#' '[' Integer_1 Integer_2 ']'` is the cardinality set including the closed interval between `Integer_1` and `Integer_2`.

4.11 XML syntax

In this section, we describe the syntax of the XML language by going through the *Document Type Declaration (DTD)* of it. To validate your own XML language files, we recommend the use of an XML validator such as the free `rxp` (available here: http://www.cogsci.ed.ac.uk/~richard/rxp.html). To understand the XML terminology, we recommend to read the W3C XML specification (available here: http://www.w3.org/TR/REC-xml).

Note that we provide an example grammar file in the XML language in Grammars/Acl01.xml. The grammar file defines exactly the same grammar as Grammars/Acl01.ul, so that it is easy to compare the two grammar file input languages.

4.11.1 Parameter entities

The DTD for the XML language begins with the definition of a couple of *parameter entities*:

```
<!ENTITY % type "(typeDomain|typeSet|typeISet|typeTuple|typeList|
                  typeRecord|typeValency|typeCard|typeVec|
                  typeInt|typeInts|typeString|typeBool|typeRef|
                  typeLabelRef|typeVariable)">
<!ENTITY % types "(%type;*)">
<!ENTITY % term "(constant|variable|integer|top|bot|
                  constantCard|constantCardSet|constantCardInterval|
                  variableCard|variableCardSet|variableCardInterval|
                  set|list|record|setGen|featurePath|annotation|
                  conj|disj|concat|order|feature|varFeature)">
<!ENTITY % terms "(%term;*)">
<!ENTITY % class "(classDimension|useClass|classConj|classDisj)">
<!ENTITY % classes "(%class;*)">
<!ENTITY % recSpec "(feature|varFeature|recordConj|recordDisj)">
<!ENTITY % recSpecs "(%recSpec;*)">
<!ENTITY % setGenSpec "(constant|setGenConj|setGenDisj)">
<!ENTITY % setGenSpecs "(%setGenSpec;*)">

<!ENTITY principleDefs SYSTEM "../../Solver/Principles/principles.xml">
<!ENTITY outputDefs SYSTEM "../../Outputs/outputs.xml">
```

The parameter entity type corresponds to the *enumerated type* encompassing the *elements* typeDomain, typeSet, typeISet, typeTuple, typeList, typeRecord, typeValency, typeCard, typeVec, typeInt, typeInts, typeString, typeBool, typeRef, typeLabelRef, and typeVariable.

The parameter entity types corresponds to zero or more occurrences of the parameter entity type.

The parameter entity term corresponds to the enumerated type encompassing the elements constant, variable, integer, top, bot, constantCard, constantCardSet, constantCardInterval, variableCard, variableCardSet, variableCardInterval, set, list, record, setGen, featurePath, annotation, conj, disj, concat, order, feature and varFeature.

The parameter entity terms corresponds to zero or more occurrences of the parameter entity term.

The parameter entity class corresponds to the enumerated type encompassing the elements classDimension, useClass, classConj, classDisj.

The parameter entity `classes` corresponds to zero or more occurrences of the parameter entity `class`.

The parameter entity `recSpec` corresponds to the enumerated type encompassing the elements `feature`, `varFeature`, `recordConj`, `recordDisj`.

The parameter entity `recSpecs` corresponds to zero or more occurrences of the parameter entity `recSpec`.

The parameter entity `setGenSpec` corresponds to the enumerated type encompassing the elements `constant`, `setGenConj`, `setGenDisj`.

The parameter entity `setGenSpecs` corresponds to zero or more occurrences of the parameter entity `setGenSpec`.

The parameter entity `principleDefs` corresponds to the *system identifier* `"../../Solver/Principles/principles.xml"`, an XML file which declares all available principle identifiers. Since XDK grammar files do not contain principle definitions but only principle uses, this is how the XML grammar file "knows" the principle identifiers which can be used. Note that you can adapt the path of the system identifier to a more suitable one (whether it is suitable will depend on your XML validator).

The parameter entity `outputDefs` corresponds to the *system identifier* `"../../Outputs/outputs.xml"`, an XML file which declares all available output identifiers. Since XDK grammar files do not contain output definitions but only output uses, this is how the XML grammar file "knows" the output identifiers which can be used. Note that you can adapt the path of the system identifier to a more suitable one (whether it is suitable will depend on your XML validator).

4.11.2 Elements

4.11.2.1 Root element (grammar)

The *root element type* of the XML language is `grammar`, defined as follows:

```
<!ELEMENT grammar (principleDef*,outputDef*,useDimension*,
                   dimension*,classDef*,entry*)>
```

I.e. a grammar file in the XML language starts with zero or more principle definitions (`principleDef*`), then zero or more output definitions (`outputDef*`), then zero or more dimension uses (`useDimension*`), then zero or more dimension definitions (`dimension*`), then zero or more lexical class definitions (`classDef*`), and finally zero or more lexical entry definitions (`entry*`).

4.11.2.2 Principle definitions (principleDef)

XDK grammar files do not include principle definitions. The principle definitions in the XML language only introduce the principle identifiers to enable the file to be validated properly:

```
<!ELEMENT principleDef EMPTY>
<!ATTLIST principleDef id ID #REQUIRED>
```

The `principleDef` element has the required *attribute* `id` which is an XML ID corresponding to the principle identifier.

4.11.2.3 Output definitions (outputDef)

XDK grammar files do not include output definitions. The output definitions in the XML language only introduce the output identifiers to enable the file to be validated properly:

```
<!ELEMENT outputDef EMPTY>
<!ATTLIST outputDef id ID #REQUIRED>
```

The `outputDef` element has the required attribute `id` which is an XML ID corresponding to the output identifier.

4.11.2.4 Dimension use (useDimension)

Here is the syntax for using dimensions:

```
<!ELEMENT useDimension EMPTY>
<!ATTLIST useDimension idref IDREF #REQUIRED>
```

The `useDimension` element has the required attribute `idref` which is an XML ID reference corresponding to the dimension identifier.

4.11.2.5 Dimension definition (dimension)

Here is the syntax for defining dimensions:

```
<!ELEMENT dimension (attrsType?,entryType?,labelType?,typeDef*,
                     usePrinciple*,output*,useOutput*)>
<!ATTLIST dimension id ID #REQUIRED>
```

I.e. a dimension definition starts with zero or one definitions of the attributes type (`attrsType?`), then with zero or one definitions of the entry type (`entryType?`), then with zero or one definitions of the label type (`labelType?`). Then, it continues with zero or more additional type definitions (`typeDef*`), then zero or more used principle (`usePrinciple*`), zero or more chosen outputs (`output*`), and finally zero or more used outputs (`useOutput*`).

It has the required attribute `id` which is an XML ID corresponding to the dimension identifier.

4.11.2.6 Attributes type (attrsType)

Here is the syntax for defining the attributes type:

```
<!ELEMENT attrsType %type;>
```

I.e. the `attrsType` element has one obligatory child which is a type.

4.11.2.7 Entry type (entryType)

Here is the syntax for defining the entry type:

```
<!ELEMENT entryType %type;>
```

I.e. the `entryType` element has one obligatory child which is a type.

4.11.2.8 Label type (labelType)

Here is the syntax for defining the label type:

```
<!ELEMENT labelType %type;>
```

I.e. the `labelType` element has one obligatory child which is a type.

4.11.2.9 Choosing an output (output)

Here is the syntax for choosing an output:

```
<!ELEMENT output EMPTY>
<!ATTLIST output idref IDREF #REQUIRED>
```

The `output` element has the required attribute `idref` which is an XML ID reference corresponding to the output identifier.

4.11.2.10 Using an output (useOutput)

Here is the syntax for using an output:

```
<!ELEMENT useOutput EMPTY>
<!ATTLIST useOutput idref IDREF #REQUIRED>
```

The `useOutput` element has the required attribute `idref` which is an XML ID reference corresponding to the output identifier.

4.11.2.11 Type definition (typeDef)

Here is the syntax for defining a type:

```
<!ELEMENT typeDef %type;>
<!ATTLIST typeDef id ID #REQUIRED>
```

I.e. the `typeDef` element has one child which is a type.

It has the required attribute `id` which is an XML ID corresponding to the type identifier.

4.11.2.12 Types (type parameter entity)

Here is the syntax of types:

```
<!ELEMENT typeDomain (constant*)>
<!ELEMENT typeSet %type;>
<!ELEMENT typeISet %type;>
<!ELEMENT typeTuple %types;>
<!ELEMENT typeList %type;>
<!ELEMENT typeRecord (typeFeature*)>
<!ELEMENT typeValency %type;>
<!ELEMENT typeCard EMPTY>
<!ELEMENT typeVec (%type;,%type;)>
<!ELEMENT typeInt EMPTY>
<!ELEMENT typeInts EMPTY>
<!ELEMENT typeString EMPTY>
<!ELEMENT typeBool EMPTY>
<!ELEMENT typeRef EMPTY>
<!ATTLIST typeRef idref IDREF #REQUIRED>
<!ELEMENT typeLabelRef EMPTY>
<!ATTLIST typeLabelRef data NMTOKEN #REQUIRED>
<!ELEMENT typeVariable EMPTY>
```

`typeDomain` is a finite domain of constants `constant*`.

`typeSet` is an accumulative set with domain `%type;`.

`typeISet` is an intersective set with domain `%type;`.

`typeTuple` is a tuple with projections `%types;`.

`typeList` is a list with domain `%type;`.

`typeRecord` is a record with features `typeFeature*`.

`typeValency` is a valency with domain `%type;`.

`typeCard` is a cardinality set.

`typeVec` is a vector with fields and value type `(%type;,%type;)`.

`typeInt` is an integer.

`typeInts` is a set of integers.

`typeString` is a string.

`typeBool` is a boolean.

`typeRef` is a type reference to the type identifier specified by its required `idref` attribute (an XML ID reference).

`typeLabelRef` is a reference to the label type of the dimension variable specified by the required `data` attribute (an XML *name token*).

`typeVariable` is a type variable.

4.11.2.13 Features (typeFeature and feature)

Here is the syntax for type features:

```
<!ELEMENT typeFeature %type;>
<!ATTLIST typeFeature
          data NMTOKEN #REQUIRED>
<!ELEMENT feature %term;>
<!ATTLIST feature
          data NMTOKEN #REQUIRED>
<!ELEMENT varFeature %term;>
<!ATTLIST varFeature
          data NMTOKEN #REQUIRED>
```

The `typeFeature` element has one child which is a type (`%type;`), and the required attribute `data` (an XML name token) which is its field.

The `feature` element has one child which is a term (`%term;`), and the required attribute `data` (an XML name token) which is its field.

The `varFeature` element has one child which is a term (`%term;`), and the required attribute `data` (an XML name token) which is its field.

4.11.2.14 Principle use (usePrinciple)

Here is the syntax for using principles:

```
<!ELEMENT usePrinciple (dim*,arg*)>
<!ATTLIST usePrinciple idref IDREF #REQUIRED>

<!ELEMENT dim EMPTY>
<!ATTLIST dim
          var NMTOKEN #REQUIRED
          idref IDREF #REQUIRED>
```

```
<!ELEMENT arg %term;>
<!ATTLIST arg
          var NMTOKEN #REQUIRED>
```

The usePrinciple element has zero or more dim children which establish the dimension mapping, followed by zero or more arg children which establish the argument mapping. It has the required attribute idref which is an XML ID reference to the used principle identifier.

The dim element has the required attributes var (an XML name token), and idref (an XML ID reference). var is the dimension variable, and idref is the dimension ID to which the former is bound.

The arg element has one child which is a term (%term;). It has the required attribute var (an XML name token). var is the argument variable to which the term is bound.

4.11.2.15 Class definitions (classDef)

Here is the syntax for class definitions:

```
<!ELEMENT classDef (variable*,%classes;)>
<!ATTLIST classDef
          id ID #REQUIRED>
```

I.e. the classDef element has zero or more variable children, and one child corresponding to the parameter entity classes (%classes;).

It has the required attribute id, an XML ID corresponding to the class identifier.

4.11.2.16 Class bodies

Here is the syntax for class bodies:

The parameter entity classes corresponds to either of the elements classDimension, useClass, classConj, or classDisj:

```
<!ELEMENT classDimension %term;>
<!ATTLIST classDimension idref IDREF #REQUIRED>

<!ELEMENT useClass (feature*)>
<!ATTLIST useClass idref IDREF #REQUIRED>

<!ELEMENT classConj %classes;>

<!ELEMENT classDisj %classes;>
```

The classDimension element specifies a dimension entry (%term;) for the dimension with the identifier given by the required attribute idref (an XML ID reference).

The useClass element specifies the use of a lexical class with the class identifier given by the required attribute idref (an XML ID reference). The parameters of this class are specified as a list of features (features*).

The classConj element specifies the conjunction of its children.

The classDisj element specifies the disjunction of its children.

4.11.2.17 Lexical entries (entry)

Here is the syntax for lexical entries:

```
<!ELEMENT entry %classes;>
```

I.e. the `entry` element specifies a lexical entry as a list of class bodies (`%classes;`).

4.11.2.18 Terms (term parameter entity)

Here is the syntax for terms:

```
<!ELEMENT constant EMPTY>
<!ATTLIST constant data NMTOKEN #REQUIRED>
<!ELEMENT integer EMPTY>
<!ATTLIST integer data NMTOKEN #REQUIRED>
<!ELEMENT top EMPTY>
<!ELEMENT bot EMPTY>
<!ELEMENT variable EMPTY>
<!ATTLIST variable data NMTOKEN #REQUIRED>
<!ELEMENT constantCard EMPTY>
<!ATTLIST constantCard
          data NMTOKEN #REQUIRED
  card (one|opt|any|geone) "one">
<!ELEMENT constantCardSet (integer*)>
<!ATTLIST constantCardSet
          data NMTOKEN #REQUIRED>
<!ELEMENT constantCardInterval (integer,integer)>
<!ATTLIST constantCardInterval
          data NMTOKEN #REQUIRED>
<!ELEMENT variableCard EMPTY>
<!ATTLIST variableCard
          data NMTOKEN #REQUIRED
  card (one|opt|any|geone) "one">
<!ELEMENT variableCardSet (integer*)>
<!ATTLIST variableCardSet
          data NMTOKEN #REQUIRED>
<!ELEMENT variableCardInterval (integer,integer)>
<!ATTLIST variableCardInterval
          data NMTOKEN #REQUIRED>
<!ELEMENT set %terms;>
<!ELEMENT list %terms;>
<!ELEMENT record %recSpecs;>
<!ELEMENT recordConj %recSpecs;>
<!ELEMENT recordDisj %recSpecs;>
<!ELEMENT setGen %setGenSpecs;>
<!ELEMENT setGenConj %setGenSpecs;>
<!ELEMENT setGenDisj %setGenSpecs;>
<!ELEMENT featurePath (constant*)>
<!ATTLIST featurePath
          root (down|up) #REQUIRED
```

```
              dimension NMTOKEN #REQUIRED
              aspect (entry|attrs) #REQUIRED>
    <!ELEMENT annotation (%term;,%type;)>
    <!ELEMENT conj %terms;>
    <!ELEMENT disj %terms;>
    <!ELEMENT concat %terms;>
```

The `constant` element defines a constant. It has the required attribute `data` (an XML name token) which is the constant itself

The `integer` element defines an integer. It has the required attribute `data` (an XML name token) which is the integer itself.

The `top` element corresponds to lattice top.

The `bot` element corresponds to lattice bottom.

The `variable` element defines a variable. It has the required attribute `data` (an XML name token) which is the variable itself.

The `constantCard` element defines a cardinality specification. It has the attributes `data` (an XML name token) and `card`, of which `data` is required and `card` is optional (with *attribute default* `one`). `data` corresponds to the field of the cardinality specification, and `card` to the cardinality set. Here, `one` corresponds to `!` in the UL, `opt` to `?`, `any` to `*`, and `geone` to `+`.

The `constantCardSet` element also defines a cardinality specification. It has zero or more `integer` children and the required attribute `data` (an XML name token). `data` is the field of the cardinality specification. The `integer` children the set of integers in the cardinality set.

The `constantCardInterval` element also defines a cardinality specification. It has two `children` and the required attribute `data` (an XML name token). `data` is the field of the cardinality specification. The two integers define the cardinality set by a closed interval.

`variableCard`, `variableCardSet` and `variableCardInterval` have variable instead of constant features.

The `set` element specifies a set of terms (`%terms;`).

The `list` element specifies a list of terms (`%terms;`).

The `record` element specifies a record. Therefore, it utilizes record specifications (`%recSpecs;`). A record specification is either a feature (`feature`), a variable feature (`varFeature`), a conjunction of record specifications (`recordConj`), or a disjunction of record specifications (`recordDisj`).

The `setGen` element specifies a set generator expression. The body of a set generator expression is a list of specifications (`%setGenSpecs;`). A set generator expression specification is either a constant (`constant`), a conjunction of set generator expression specifications (`setGenConj`), or a disjunction of set generator expression specifications (`setGenDisj`).

The `featurePath` element specifies a feature path. The required attribute `root` (`down` or `up`) corresponds to the root variabe of the feature path, the required attribute `dimension` to the dimension variable, and the required attribute `aspect` to the aspect (`entry` or `attrs`). The `constant` children of the `featurePath` element correspond to the fields of the feature path. Note that the root variable value `down` corresponds to `_` in the UL, and `up` to `^`.

The `annotation` element specifies a type annotation for a term. Its first child is a term (`%term;`), and its second child a type (`%type;`).

The `conj` element specifies the conjunction of a list of terms (`%terms;`).

The `disj` element specifies the disjunction of a list of terms (`%terms;`).

The `concat` element specifies the *concatenation* of a list of terms (`%terms;`). Concatenation is restricted to strings.

The `order` element specifies an *order generator* for a list of terms (`%terms;`).

4.12 IL syntax

In this section, we describe the syntax of *Intermediate Language (IL)* grammar files. The IL is a record language in Mozart-Oz syntax. It is designed specifically to be dealt with easily in Mozart-Oz.

We describe the syntax of the IL in a notation similar to the *Extended Backus Naur Form (EBNF)*.

We use the record syntax of Mozart-Oz, where records look like this:

```
<constant>(<constant_1>:<value_1>
           ...
           <constant_n>:<value_n>)
```

where `<constant>` is the record name, and `<constant_i>:<value_i>` is a feature with field `<constant_i>` and value `<value_i>` (`1<=i<=n`).

4.12.1 Descriptions

We write down the syntax of the IL as a mapping from *description identifiers* to *descriptions*. This mapping is written as a Mozart-Oz record as follows:

```
o(<description id_1>:<description_1>
  ...
  <description id_n>:<description_n>)
```

A description can be one of the following:

- tuple of `n` descriptions: `<description_1>#...#<description_n>`
- list of descriptions: `'*'(<description>)` (if such a description is not given, the empty list is assumed)
- optional description: `'?'(<description>)` (if such a description is not given, a default depending on the description is assumed)
- a reference to a description identifier: `<description_id>`
- a disjunction of `n` descriptions:

```
disj(<description_1>
     ...
     <description_n>)
```

- a complex description (an element):

```
elem(tag: <constant>
     <constant_1>: <description_1>
     ...
     <constant_n>: <description_n>)
```

- a simple description, i.e. either:
  - an atom defining a unique identifier: `'ID'`
  - an atom referring to a unique identifier: `'IDREF'`
  - character data: `'CDATA'`
  - an atom: `'ATOM'`
  - an integer: `'INT'`

We use single quotes to escape Mozart-Oz keywords (e.g. `functor`), tokens starting with an upper case letter (Mozart-Oz variables), and tokens containing dots (e.g. `'principle.tree'`).

4.12.2 Syntax checker

The XDK includes the Mozart-Oz functor `Compiler/SyntaxChecker.ozf` whose exported procedure `Check` can be used to check whether an file fulfills its syntax specifications. The IL syntax specification is in the functor `Compiler/ILSyntax.ozf`. The file `Compiler/SyntaxCheckerTest.oz`, which can be fed in the *Oz Programming Interface (OPI)*, demonstrates how to use the syntax checker to check whether the output of the UL and XML language frontends are syntactically correct.

The syntax checker will be useful if you decide to design a new grammar file input language in addition to the UL and the XML language.

4.12.3 Start symbol ('S')

We continue with giving an overview of the syntax of the IL. The start symbol of the IL syntax is 'S':

```
'S': 'GRAMMAR'
```

4.12.4 Grammar definition ('GRAMMAR')

A grammar definition has tag `grammar`:

```
'GRAMMAR': elem(tag: grammar
                principles: '*'('PRINCIPLEDEF')
                outputs: '*'('OUTPUTDEF')
                usedimensions: '*'('CONSTANT')
                dimensions: '*'('DIMENSION')
                classes: '*'('CLASSDEF')
                entries: '*'('ENTRY'))
```

The `principles` feature corresponds to a list of principle definitions (`'*'('PRINCIPLEDEF')`).

The `outputs` feature corresponds to a list of output definitions (`'*'('OUTPUTDEF')`).

The `usedimensions` feature corresponds to a list of constants which represent the identifiers of the used dimensions (`'*'('CONSTANT')`).

The `dimensions` feature corresponds to a list of dimension definitions (`'*'('DIMENSION')`).

The `classes` feature corresponds to a list of class definitions (`'*'('CLASSDEF')`).

The `entries` feature corresponds to a list of lexical entries (`'*'('ENTRY')`).

4.12.5 Principle definitions ('PRINCIPLEDEF')

A principle definition has tag `principledef`. Notice that principle definitions can only be written in the IL since they are closed for the user. They cannot be written in the UL or the XML language:

```
'PRINCIPLEDEF': elem(tag: principledef
                     id: 'CONSTANT'
                     dimensions: '*'('VARIABLE')
                     args: '*'('VARIABLE'#'TYPE')
                     defaults: '*'('VARIABLE'#'TERM')
                     node: '?'('TYPE')
                     constraints: '*'('CONSTANT'#'INTEGER'))
```

The `id` feature corresponds to the principle identifier (`'CONSTANT'`).

The `dimensions` feature corresponds to a list of dimension variables (`'*'('VARIABLE')`), the dimension variables which are introduced by the principle.

The `args` feature corresponds to a list of pairs of argument variables and their types (`'*'('VARIABLE'#'TYPE')`).

The `defaults` feature corresponds to a list of pairs of argument variables and their default values (`'*'('VARIABLE'#'TERM')`).

The `node` feature corresponds to an optional model record type `'?'('TYPE')`. If this feature is not given, the empty record is assumed.

The `constraints` feature corresponds to a list of pairs of constraint names and their priorities (`'*'('CONSTANT'#'INTEGER')`).

4.12.6 Output definitions

An output definition has tag `outputdef`. Notice that output definitions can only be written in the IL since they are closed for the user. They cannot be written in the UL or the XML language.

```
'OUTPUTDEF': elem(tag: outputdef
                  id: 'CONSTANT'
                  'functor': 'CONSTANT')
```

The `id` feature corresponds to the output identifier (`'CONSTANT'`).

The `'functor'` feature corresponds to the functor name of the output (`'CONSTANT'`).

4.12.7 Dimension definitions ('DIMENSION')

A dimension definition has tag `dimension`:

```
'DIMENSION': elem(tag: dimension
                  id: 'CONSTANT'
                  attrs: '?'('TYPE')
                  entry: '?'('TYPE')
                  label: '?'('TYPE')
                  types: '*'('TYPEDEF')
                  principles: '*'('USEPRINCIPLE')
                  outputs: '*'('OUTPUT')
                  useoutputs: '*'('USEOUTPUT'))
```

The `id` feature corresponds to the dimension identifier (`'CONSTANT'`).

The `attrs` feature corresponds to an optional attributes type (`'?'('TYPE')`). The default for this description is the empty record.

The `entry` feature corresponds to an optional entry type (`'?'('TYPE')`). The default for this description is the empty record.

The `label` feature corresponds to an optional label type (`'?'('TYPE')`). The default for this description is the empty domain.

The `types` feature corresponds to a list of type definitions (`'*'('TYPEDEF')`).

The `principles` feature corresponds to a list of principle uses (`'*'('USEPRINCIPLE')`).

The `outputs` feature corresponds to a list of chosen outputs (`'*'('OUTPUT')`).

The `useoutputs` feature corresponds to a list of used outputs (`'*'('USEOUTPUT')`).

4.12.8 Output chooses ('OUTPUT')

An output chooses has tag `output`:

```
'OUTPUT': elem(tag: output
               idref: 'CONSTANT')
```

The `idref` feature corresponds to the chosen output identifier (`'CONSTANT'`).

4.12.9 Output uses ('USEOUTPUT')

An output choosing has tag `useoutput`:

```
'USEOUTPUT': elem(tag: useoutput
                  idref: 'CONSTANT')
```

The `idref` feature corresponds to the used output identifier (`'CONSTANT'`).

4.12.10 Type definitions ('TYPEDEF')

A type definition has tag `typedef`:

```
'TYPEDEF': elem(tag: typedef
                id: 'CONSTANT'
                type: 'TYPE')
```

The `id` feature corresponds to the type identifier (`'CONSTANT'`).

The `type` feature corresponds to the type (`'TYPE'`).

4.12.11 Types ('TYPE')

The description identifier `'TYPE'` corresponds to the following:

```
'TYPE': disj(elem(tag: 'type.domain'
                  args: '*'('CONSTANT'))
             elem(tag: 'type.set'
                  arg: 'TYPE')
             elem(tag: 'type.iset'
                  arg: 'TYPE')
             elem(tag: 'type.tuple'
                  args: '*'('TYPE'))
             elem(tag: 'type.list'
```

```
     arg: 'TYPE')
elem(tag: 'type.record'
     args: '*'('CONSTANT'#'TYPE'))
elem(tag: 'type.valency'
     arg: 'TYPE')
elem(tag: 'type.card')
elem(tag: 'type.vec'
     arg1: 'TYPE'
     arg2: 'TYPE')
elem(tag: 'type.int')
elem(tag: 'type.ints')
elem(tag: 'type.string')
elem(tag: 'type.bool')
elem(tag: 'type.ref'
     idref: 'CONSTANT')
elem(tag: 'type.labelref'
     arg: 'VARIABLE')
elem(tag: 'type.variable'
     data: 'ATOM'))
```

4.12.11.1 Domain types ('type.domain')

A domain type has tag `'type.domain'`. The `args` feature corresponds to the set of constants in the domain (`'*'('CONSTANT')`).

4.12.11.2 Accumulative set types ('type.set')

An accumulative set type has tag `'type.set'`. The `arg` feature corresponds to the type of the domain of the set (`'TYPE'`).

4.12.11.3 Intersective set types ('type.iset')

An intersective set type has tag `'type.iset'`. The `arg` feature corresponds to the type of the domain of the set (`'TYPE'`).

4.12.11.4 Tuple types ('type.tuple')

A tuple type has tag `'type.tuple'`. The `args` feature corresponds to the types of the projections of the tuple (`'*'('TYPE')`).

4.12.11.5 List types ('type.list')

A list type has tag `'type.record'`. The `arg` feature corresponds to the type of the domain of the list (`'TYPE'`).

4.12.11.6 Record types ('type.record')

A record type has tag `'type.list'`. The `args` feature corresponds to a list of pairs of the record fields and their types (`'*'('CONSTANT'#'TYPE')`).

4.12.11.7 Valency types ('type.valency')

A valency type has tag `'type.valency'`. The `arg` feature corresponds to the type of the domain of the valency (`'TYPE'`).

4.12.11.8 Vector types ('type.vec')

A vector type has tag `'type.vec'`. The `arg1` feature corresponds to the domain of the fields of the vector (`'TYPE'`), and the `arg2` feature to the type of the values (`'TYPE'`).

4.12.11.9 Integer types ('type.int')

An integer type has tag `'type.int'`.

4.12.11.10 Set of integers types ('type.ints')

A set of integers type has tag `'type.ints'`.

4.12.11.11 String types ('type.string')

A string type has tag `'type.string'`.

4.12.11.12 Bool types ('type.bool')

A boolean type has tag `'type.bool'`.

4.12.11.13 Type reference types ('type.ref')

A type reference has tag `'type.ref'`. The `idref` feature corresponds to the identifier of the referenced type (`'CONSTANT'`).

4.12.11.14 Label reference types ('type.labelref')

A label reference has tag `'type.labelref'`. The `arg` feature corresponds to the dimension variable whose set of edge labels is referenced (`'VARIABLE'`).

4.12.11.15 Type variable ('type.variable')

A type variable has tag `'type.variable'`.

4.12.12 Principle uses ('USEPRINCIPLE')

A principle use has tag `useprinciple`:

```
'USEPRINCIPLE': elem(tag: useprinciple
                     idref: 'CONSTANT'
                     dimensions: '*'('VARIABLE'#'CONSTANT')
                     args: '*'('VARIABLE'#'TERM'))
```

The `idref` feature corresponds to the principle identifier of the used principle (`'CONSTANT'`).

The `dimensions` feature corresponds to the dimension mapping, a list of pairs of dimension variables and dimension identifiers (`'*'('VARIABLE'#'CONSTANT')`).

The `args` feature corresponds to the argument mapping, a list of pairs of argument variables and their values (`'*'('VARIABLE'#'TERM')`).

4.12.13 Class definitions ('CLASSDEF')

A class definition has tag `classdef`:

```
'CLASSDEF': elem(tag: classdef
                 id: 'CONSTANT'
                 vars: '*'('VARIABLE')
                 body: 'CLASS')
```

The `id` feature corresponds to the class identifier (`'CONSTANT'`).

The `vars` feature corresponds to the list of variables which are bound by the class (`'*'('VARIABLE')`).

The `body` feature corresponds to the class body (`'CLASS'`).

4.12.14 Class bodies ('CLASS')

The description identifier `'CLASS'` corresponds to the following:

```
'CLASS': disj(elem(tag: 'class.dimension'
                   idref: 'CONSTANT'
                   arg: 'TERM')
              elem(tag: 'class.ref'
                   idref: 'CONSTANT'
                   args: '*'('VARIABLE'#'TERM'))
              elem(tag: conj
                   args: '*'('CLASS'))
              elem(tag: disj
                   args: '*'('CLASS')))
```

4.12.14.1 Entry dimension ('class.dimension')

An entry dimension has tag `'class.dimension'`. The `idref` feature corresponds to the dimension identifier (`'CONSTANT'`), and the `arg` feature corresponds to the entry dimension itself (`'TERM'`).

4.12.14.2 Class reference ('class.ref')

A class reference has tag `'class.ref'`. The `idref` feature corresponds to the class identifier of the referenced class. The `args` feature corresponds to the list of pairs of variables and values that specify the variable binding of the class reference (`'*'('VARIABLE'#'TERM')`).

4.12.14.3 Conjunction (conj)

A conjunction of class bodies has tag `conj`. The `args` feature corresponds to the list of class bodies which are combined by conjunction (`'*'('CLASS')`).

4.12.14.4 Disjunction (disj)

A disjunction of class bodies has tag `disj`. The `args` feature corresponds to the list of class bodies which are combined by disjunction (`'*'('CLASS')`).

4.12.15 Lexical entries ('ENTRY')

A lexical entry has tag `entry`.

```
'ENTRY': elem(tag: entry
              body: 'CLASS')
```

The body feature corresponds to the class body which specifies the lexical entry ('CLASS').

4.12.16 Terms ('TERM')

The description identifier 'TERM' corresponds to the following:

```
'TERM': disj('CONSTANT'
             'VARIABLE'
             'INTEGER'
             'CARD'
             'CONSTANT'#'CARD'
             'VARIABLE'#'CARD'
             elem(tag: top)
             elem(tag: bot)
             elem(tag: set
                  args: '*'('TERM'))
             elem(tag: list
                  args: '*'('TERM'))
             elem(tag: record
                  args: '*'('RECSPEC'))
             elem(tag: setgen
                  arg: 'SETGEN')
             elem(tag: featurepath
                  root: 'ROOT'
                  dimension: 'VARIABLE'
                  aspect: 'ASPECT'
                  fields: '*'('CONSTANT'))
             elem(tag: annotation
                  arg1: 'TERM'
                  arg2: 'TYPE')
             elem(tag: conj
                  args: '*'('TERM'))
             elem(tag: disj
                  args: '*'('TERM'))
             elem(tag: concat
                  args: '*'('TERM'))
             elem(tag: order
                  args: '*'('TERM')))

'ROOT': disj('_' '^')
'ASPECT': disj('entry' 'attrs')
```

4.12.16.1 Constants ('CONSTANT')

A constant has tag constant:

```
'CONSTANT': elem(tag: constant
```

```
                data: 'ATOM')
```

The data feature corresponds to the constant itself.

4.12.16.2 Variables ('VARIABLE')

A variable has tag variable.

```
'VARIABLE': elem(tag: variable
                 data: 'ATOM')
```

The data feature corresponds to the variable itself.

4.12.16.3 Integers ('INTEGER')

An integer has tag integer.

```
'INTEGER': disj(elem(tag: integer
                     data: 'INT')
                elem(tag: integer
                     data: 'infty'))
```

An integer is either an integer or infinity.

The data feature of an integer corresponds to the integer itself or to the special constant infty (for "infinity").

4.12.16.4 Cardinality sets ('CARD')

The description identifier CARD corresponds to the following:

```
'CARD': disj(elem(tag: 'card.wild'
                  arg: 'WILD')
             elem(tag: 'card.set'
                  args: '*'('INTEGER'))
             elem(tag: 'card.interval'
                  arg1: 'INTEGER'
                  arg2: 'INTEGER'))

'WILD': disj('!' '?' '*' '+')
```

A wild card cardinality set has tag 'card.wild'. The arg feature corresponds to one of the wild cards '!', '?', '*', or '+'.

A cardinality set has tag 'card.set'. The args feature corresponds to a list of integers which specify the set ('*'('INTEGER')).

A cardinality interval has tag 'card.interval'. The arg1 feature corresponds to the left endpoint of the closed interval ('INTEGER'), and the arg2 feature to the right endpoint ('INTEGER').

4.12.16.5 Cardinality specifications ('CONSTANT'#'CARD' or 'VARIABLE'#'CARD')

Cardinality specifications have the syntax 'CONSTANT'#'CARD', or 'VARIABLE'#'CARD'.

4.12.16.6 Top values (top)

Top values have tag top.

4.12.16.7 Bottom values (bot)

Bottom values have tag bot.

4.12.16.8 Sets (set)

Sets have tag set. The args feature corresponds to the set elements ('*'('TERM')).

4.12.16.9 Lists (list)

Lists have tag list. The args feature corresponds to the list elements ('*'('TERM')).

4.12.16.10 Records (record)

Records have tag record. The args feature corresponds to the list of record specifications for this record '*'('RECSPEC').

A record specification (description identifier 'RECSPEC') has the following syntax:

```
'RECSPEC': disj('CONSTANT'#'TERM'
                'VARIABLE'#'TERM'
                elem(tag: conj
                     args: '*'('RECSPEC'))
                elem(tag: disj
                     args: '*'('RECSPEC')))
```

I.e. a record specification is either a pair of a field and a value ('CONSTANT'#'TERM'), of a variable and a value ('VARIABLE'#'TERM'), or a conjunction of record specifications (tag: conj), or a disjunction of record specifications (tag: disj).

4.12.16.11 Set generator expressions (setgen)

A set generator expression has tag setgen. The arg feature is the set generator expression body (description identifier 'SETGEN'):

```
'SETGEN': disj('CONSTANT'
               elem(tag: conj
                    args: '*'('SETGEN'))
               elem(tag: disj
                    args: '*'('SETGEN')))
```

I.e. a set generator expression body is either a constant, or a conjunction of set generator expression bodies (tag: conj), or a disjunction of set generator expression bodies (tag: disj).

4.12.16.12 Feature paths (featurepath)

A feature path has tag featurepath.

The root feature corresponds to the root variable of the feature path, either '_' or '^'.

The dimension feature corresponds to the dimension variable of the feature path ('VARIABLE').

The aspect feature corresponds to the aspect of the feature path, either 'entry' or 'attrs'.

The fields feature corresponds to the fields of the feature path ('*'('CONSTANT')).

4.12.16.13 Type annotations (`annotation`)

A type annotation has tag `annotation`.

The `arg1` feature corresponds to the term (`'TERM'`), and the `arg2` feature to the annotated type (`'TYPE'`).

4.12.16.14 Conjunction (`conj`)

A conjunction has tag `conj`.

The `args` feature corresponds to the list of terms which are combined by conjunction.

4.12.16.15 Disjunction (`disj`)

A disjunction has tag `disj`.

The `args` feature corresponds to the list of terms which are combined by disjunction.

4.12.16.16 Concatenation (`concat`)

A concatenation has tag `concat`.

The `args` feature corresponds to the list of terms which are to be *concatenated*. Concatenation is restricted to strings.

4.12.16.17 Order (`order`)

An *order generator* has tag `order`.

The `args` feature corresponds to the list of terms from which a set of pairs representing an order relation is generated.

4.12.17 Undetermined values

The XDK solver can also yield *partial solutions* in which not all values in the node record are determined; instead some of the values are still variables. In the following, we show how these variables are represented in the IL.

4.12.17.1 Undetermined cardinality sets

This is the IL syntax for undetermined cardinality sets (i.e. cardinality set variables) in valencies:

```
elem(tag: '_'
     args: [IL1 IL2])
```

`IL1` is the cardinality set representing the set of integers which are already known to be in the cardinality set variable.

`IL2` is the cardinality set representing the set of integers which can still be bound to the cardinality set variable.

4.12.17.2 Undetermined constants

This is the IL syntax for undetermined constants (i.e. constant variables):

```
elem(tag: '_'
     args: [DSpec])
```

DSpec is a *domain specification*[6], representing the set of constants which can still be bound to the constant variable.

4.12.17.3 Undetermined integers

This is the IL syntax for undetermined integers (i.e. integer variables):

```
elem(tag: '_'
     args: [DSpec])
```

DSpec is a domain specification representing the set of integers which can still be bound to the integer variable.

4.12.17.4 Undetermined lists

This is the IL syntax for undetermined lists (i.e. list variables):

```
elem(tag: '_'
     args: nil)
```

4.12.17.5 Undetermined sets

The IL syntax for undetermined sets (i.e. set variables) differs depending on the domain of the set:

1. a finite domain of constants or a tuple of which all projections are finite domains of constants
2. any other type

1. The IL syntax for undetermined sets whose domain is either a finite domain of constants or a tuple of which all projections are finite domains of constants is given below:

   ```
   elem(tag: '_'
        args: [MSpec1 MSpec2 DSpec])
   ```

   MSpec1 is a set specification representing the set of constants which are already known to be in the set variable.

   MSpec2 is a set specification representing the set of constants which could still end up in the set variable.

   DSpec is a domain specification representing the set of integers which can still be bound to the integer variable representing the cardinality of the set variable.
2. The IL syntax for undetermined sets over any other domain is given below:

   ```
   elem(tag: '_'
        args: nil)
   ```

4.12.17.6 Undetermined strings

This is the IL syntax for undetermined strings (i.e. string variables):

```
elem(tag: '_'
     args: nil)
```

---

[6] A domain specification can be a list of values (or pairs of values), or just a value (or a pair of values) describing the domain. Pairs of values denote closed intervals within the set.

4.12.17.7 Undetermined tuples

The IL syntax for undetermined tuples (i.e. tuple variables) differs depending on the types of the projections of the tuple:

1. all projections are finite domains of constants
2. any of the projections has a type other than finite domain of constants

1. This is the IL syntax for undetermined tuples of which all projections are finite domains of constants:

   ```
   elem(tag: '_'
        args: [DSpec])
   ```

   DSpec is a domain specification representing the set of tuples which can still be bound to the tuple variable.

2. This is the IL syntax for undetermined tuples of which any of the projections has a type other than finite domain of constants:

   ```
   elem(tag: '_'
        args: nil)
   ```

4.13 SL syntax

In this section, we describe how to transform *Intermediate Language (IL)* expressions into *Solver Language (SL)* expressions which are used in the XDK solver.

4.13.1 Feature path

Here is the syntax of IL feature paths:

```
elem(tag: featurepath
     root: RootA
     dimension: VIL
     dimension_idref: IDA
     aspect: AspectA
     fields: FieldCILs)
```

RootA is an Oz atom corresponding to the root variable, VIL is an IL variable corresponding to the dimension variable, IDA is an Oz atom corresponding to the dimension identifier, AspectA is an Oz atom corresponding to the aspect, and FieldCILs is a list of IL constants corresponding to the fields of the feature path.

And here is the corresponding SL expression:

```
featurepath(root: RootA
            dimension: DVA
            dimension_idref: IDA
            aspect: AspectA
            fields: FieldAs)
```

RootA, IDA and AspectA stay the same. DVA is an Oz atom corresponding to VIL, and FieldAs is a list of Oz atoms corresponding to FieldCILs.

4.13.2 Cardinality set

Here is the syntax of IL cardinalities:

```
elem(tag: 'card.wild'
     arg: IL)

elem(tag: 'card.set'
     args: IILs)

elem(tag: 'card.interval'
     arg1: IIL1
     arg2: IIL2)
```

And here is the corresponding SL expression:

```
M
```

where `M` is the Oz finite set encoding the cardinality.

4.13.3 Domain

Here is the syntax of IL constants:

```
elem(tag: constant
     data: A)
```

`A` is an Oz atom corresponding to the constant itself.

And here is the corresponding SL expression:

```
I
```

`I` is an Oz integer encoding `A`.

4.13.4 Integer

Here is the syntax of IL integers:

```
elem(tag: integer
     data: I)
```

`I` is an Oz integer corresponding to the integer.

And here is the corresponding SL expression:

```
I
```

`I` stays the same.

4.13.5 List

Here is the syntax of IL lists:

```
elem(tag: list
     args: ILs)
```

`ILs` is an Oz list of IL expressions.

And here is the corresponding SL expression:

```
SLs
```

`SLs` is an Oz list of SL expressions encoding `ILs`.

4.13.6 Record

Here is the syntax of IL records:

```
elem(tag: record
     args: [CIL1#IL1
            ...
            CILn#ILn])
```

The value of the `args` feature is a list of pairs `CILi#ILi` of an IL constant and an IL expression (`1<=i<=n`).

And here is the corresponding SL expression:

```
o(A1:SL1
  ...
  An:SLn)
```

`Ai` is the Oz atom encoding `CILi`, and `SLi` the SL expression encoding `ILi` (`1<=i<=n`).

4.13.7 Set

Here is the syntax of IL sets:

```
elem(tag: set
     args: ILs)
```

`ILs` is an Oz list of IL expressions.

The corresponding SL expression is different depending on the type of the domain of the set:

1. a finite domain of constants or a tuple whose projections are all finite domains of constants
2. integer
3. any other type

1. Here is the corresponding SL expression:

   ```
   M
   ```

   `M` is an Oz finite set of integers encoding the constants in the set.
2. See 1.
3. Here is the corresponding SL expression:

   ```
   SLs
   ```

   `SLs` is an Oz list of SL expressions encoding `ILs`.

4.13.8 Order

Here is the syntax of order generators:

```
elem(tag: order
     args: ILs)
```

`ILs` is an Oz list of IL expressions.

The corresponding SL expression is the encoding of the set of all tuples representing the order relation described by `ILs`. For instance, the encoding of:

```
elem(tag: order
     args: [elem(tag: constant
                 data: a)
            elem(tag: constant
                 data: b)
            elem(tag: constant
                 data: c)])
```

is:

```
elem(tag: set
     args: [elem(tag: list
                 args: [elem(tag: constant
                             data: a)
                        elem(tag: constant
                             data: b)])
            elem(tag: list
                 args: [elem(tag: constant
                             data: a)
                        elem(tag: constant
                             data: c)])
            elem(tag: list
                 args: [elem(tag: constant
                             data: b)
                        elem(tag: constant
                             data: c)])])
```

4.13.9 String

Here is the syntax of IL strings:

```
elem(tag: constant
     data: A)
```

`A` is an Oz atom corresponding to the string.

And here is the corresponding SL expression:

```
A
```

`A` stays the same.

4.13.10 Concat

Here is the syntax of concatenation:

```
elem(tag: concat
     args: ILs)
```

`ILs` is an Oz list of IL expressions.

The corresponding SL expression is the concatenation of the IL expressions `ILs`. Concatenation is restricted to strings.

4.13.11 Tuple

Here is the syntax of IL tuples:

```
elem(tag: list
     args: ILs)
```

The corresponding SL expression is different depending on the type of the projections of the tuple:

1. all projections are finite domains of constants
2. at least one projection is not a finite domain of constants

1. Here is the corresponding SL expression:

   ```
   I
   ```

   `I` is an Oz integer encoding the tuple.
2. Here is the corresponding SL expression:

   ```
   SLs
   ```

   `SLs` is an Oz list of SL expressions encoding `ILs`.

4.13.12 Undetermined values

The XDK solver can also yield *partial solutions* in which not all values in the node record are determined; instead some of the values are still undetermined variables. In the following, we show how these variables are represented in the SL.

4.13.12.1 Undetermined cardinality sets

This is the SL syntax for undetermined cardinality sets (i.e. cardinality set variables) in valencies:

```
MSpec1#MSpec2#DSpec
```

`MSpec1` is a set specification representing the set of integers which are already known to be in the cardinality set variable.

`MSpec2` is a set specification representing the set of integers which could still end up in the cardinality set variable.

`DSpec` is a domain specification representing the set of integers which can still be bound to the integer variable representing the cardinality of the cardinality set variable.

4.13.12.2 Undetermined constants

This is the SL syntax for undetermined constants (i.e. constant variables):

```
DSpec
```

`DSpec` is a domain specification, representing the set of constants which can still be bound to the constant variable.

4.13.12.3 Undetermined integers

This is the SL syntax for undetermined integers (i.e. integer variables):

```
DSpec
```

`DSpec` is a domain specification representing the set of integers which can still be bound to the integer variable.

4.13.12.4 Undetermined lists

This is the SL syntax for undetermined lists (i.e. list variables):

```
_
```

4.13.12.5 Undetermined sets

The SL syntax for undetermined sets (i.e. set variables) differs depending on the domain of the set:

1. a finite domain of constants or a tuple of which all projections are finite domains of constants
2. any other type

1. Below, we show the SL syntax for undetermined sets over finite domains of constants or tuples of which all projections are finite domains of constants:

   ```
   MSpec1#MSpec2#DSpec
   ```

   `MSpec1` is a set specification, representing the set of constants which are already known to be in the set variable.

   `MSpec2` is a set specification representing the set of constants which could still end up in the set variable.

   `DSpec` is a domain specification representing the set of integers which can still be bound to the integer variable representing the cardinality of the set variable.
2. Below, we show the SL syntax for undetermined sets over any other domain:

   ```
   _
   ```

4.13.12.6 Undetermined strings

This is the SL syntax for undetermined strings (i.e. string variables):

```
_
```

4.13.12.7 Undetermined tuples

The SL syntax for undetermined tuples (i.e. tuple variables) differs depending on the projections of the tuple:

1. all projections are finite domains of constants
2. at least one of the projections is not a finite domain of constants

1. Below, we show the SL syntax for undetermined tuples where all projections are finite domains of constants:

   ```
   DSpec
   ```

   `DSpec` is a domain specification representing the set of tuples which can still be bound to the tuple variable.

   2) Below, we show the SL syntax for undetermined tuples where at least one projection is not a finite domain of constants:

   ```
   _
   ```

4.14 Grammar record

The *grammar record* is the result of compilation of a grammar file, and represents all the information contained in the grammar file. The XDK makes use of two different kinds of grammar records:

1. stateless grammar records
2. stateful grammar records

Stateless grammar records can be saved to disk. Before the XDK solver can actually use the grammar for solving, it has to be transformed into a stateful grammar record (an extension of the stateless grammar record).

4.14.1 Stateless grammar record

Below, we display the type definition of the stateless grammar record:

```
grammar(
   usedDIDAs: DIDAs
   allDIDAs: DIDAs
   dIDADimension1Rec: DIDASLERec
   pnPrinciple1Rec: PnSLERec
   usedPns: Pns
   chosenPns: Pns
   pnCAPriITups: PnCAITups
   dIDAUsedOnsRec: DIDAOnsRec
   onOutput1Rec: OnSLERec
   chosenOns: Ons
   usedOns: Ons
   aEntriesRec: ASLERec
   as: As
   entriesI: I
   entry1Tn: Tn
   nodeTn: Tn
   node1Tn: Tn
   tnTypeRec: TnILTCoRec)
```

The value of the `usedDIDAs` feature is a list of dimension identifiers (`DIDAs`) which are the used dimensions.

The value of the `allDIDAs` feature is a list of dimension identifiers (`DIDAs`) which are all the dimensions defined in the grammar.

The value of the `dIDADimension1Rec` feature is a record mapping dimension identifiers (`DIDA`) to encoded dimension definitions (`SLE`).

The value of the `pnPrinciple1Rec` feature is a record mapping *principle names* (`Pn`) to encoded principle uses (`SLE`). A principle name is a unique name for a principle chosen on a dimension.[7]

The value of the `usedPns` feature is a list of principle names (`Pns`) which are the used principles.

The value of the `chosenPns` feature is a list of principle names (`Pns`) which are all the principles chosen in the grammar.[8]

The value of the `pnCAPriITups` feature is a list of tuples `Pn#CA#I` of a principle name (`Pn`), a principle constraint name `CA`, and a priority (`I`). These are all principle constraints.

[7] Contrary to principle names, principle identifiers are not unique because the same principle can be used on several dimensions.

[8] The distinction between chosen and used principles makes sense for the graphical user interface: here, the `Principles` pull-down menu displays all chosen principles, of which only the selected principles are actually used for solving.

The value of the `dIDAUsedOnsRec` feature is a record mapping dimension identifiers (`DIDA`) to the *output names* of the outputs used used on that dimension. An output name is a unique name for an output chosen on a dimension.[9]

The value of the `onOutput1Rec` feature is a record mapping output names (`On`) to encoded output definitions (`SLE`).

The value of the `chosenOns` feature is a list of the chosen output names.

The value of the `usedOns` feature is a list of the used output names.

The value of the `aEntriesRec` feature is a record mapping words (`A`) to their corresponding encoded lexical entries (`SLE`). This record is empty if the lexicon is stored into a database.

The value of the `as` feature is the list of all words (`As`) in the lexicon.

The value of the `entriesI` feature is an integer denoting the number of lexical entries of the grammar.

The value of the `entry1Tn` feature is the type name (`Tn`) corresponding to the type of a lexical entry.

The value of the `nodeTn` feature is the type name (`Tn`) corresponding to the type of a node record, including the features `word`, `index`, `nodeSet` and `entryIndex`, and excluding the subrecords corresponding to the dimensions.

The value of the `node1Tn` feature is the type name (`Tn`) corresponding to the type of a complete node record, including the features of the type `nodeTn` plus the `attrs`, `entry` and `model` features from the individual dimensions.

The value of the `tnTypeRec` feature is a record mapping type names (`Tn`) to their corresponding types (`ILTCo`).

Notice that `ILTCo` types are different form `IL` types in various respects:

- all type references are resolved
- all label type references are resolved
- in all domain types, all constants in the `args` of the type are converted from IL constants to Oz atoms
- in all record types, all a record type specification consisting of pairs `CIL_i#IL_i` (`1<=i<=n`) is converted to an Oz record containing features `A_i:IL_i` (where Oz atom `A_i` corresponds to IL constant `CIL_i`).

4.14.2 Stateful grammar record

Below, we display the type definition of the features extending a stateless grammar record into a stateful grammar record:

```
grammar(
   dIDADimensionRec: DIDASLCRec
   dIDA2AttrsLat: DIDA2Lat
   dIDA2EntryLat: DIDA2Lat
   dIDA2LabelLat: DIDA2Lat
```

[9] Like principles, contrary to output names, output identifiers are not unique because the same output can be chosen on several dimensions.

```
pnPrincipleRec: PnSLCRec
pn2Principle: Pn2SLC
pn2ModelLat: Pn2Lat
pn2DIDA: Pn2DIDA
pnIsActive: PnIsActive
procProcPnCAPriITups: ProcProcPnCAITups
onOutputRec: OnSLCRec
onOutputTups: OnSLCTups
checkAsInEntries: As2U
as2ABRec: As2ABRec
as2AEntriesRec: As2AEntriesRec
entry1Lat: Lat
nodeLat: Lat
node1Lat: Lat)
```

The value of the `dIDADimensionRec` feature is a record mapping dimension identifiers (`DIDA`) to compiled dimension definitions (`SLC`).

The value of the `dIDA2AttrsLat` feature is a function from dimension identifiers (`DIDA`) to the lattice corresponding to the attributes type on that dimension (`Lat`).

The value of the `dIDA2EntryLat` feature is a function from dimension identifiers (`DIDA`) to the lattice corresponding to the entry type on that dimension (`Lat`).

The value of the `dIDA2LabelLat` feature is a function from dimension identifiers (`DIDA`) to the lattice corresponding to the label type on that dimension (`Lat`).

The value of the `pnPrincipleRec` feature is a record mapping principle names (`Pn`) to compiled principle uses (`SLC`).

The value of the `pn2Principle` feature is a function from principle names (`Pn`) to compiled principle uses (`SLC`).

The value of the `pn2ModelLat` feature is a function from principle names (`Pn`) to the lattice corresponding to the type of the model record introduced by that principle (`Lat`).

The value of the `pn2DIDA` feature is a function from principle names (`Pn`) to dimension identifiers corresponding to the dimension on which the principle is used (`DIDA`).

The value of the `pnIsActive` feature is the function `PnIsActive: Pn UsedDIDAs UsedPns -> B`, returning for principle `Pn`, whether it is active given used dimension IDs `UsedDIDAs` and used principle names `UsedPns`.

The value of the `procProcPnCAPriITups` feature is a list of tuples `Proc#Proc1#Pn#CA#I` of a constraint procedure (`Proc`), a profile procedure (`Proc1`), a principle name (`Pn`), a principle constraint name (`CA`) and a priority (`I`). These are all principle constraints.

The value of the `onOutputRec` feature is a record mapping output names (`On`) to compiled output uses (`SLC`).

The value of the `onOutputTups` feature is a list of tuples `On#Output` of an output name (`On`) and an output (`Output`). These tuples represent the mapping already contained in the value of the `onOutputRec` feature. The difference is that they are ordered alphabetically (by their name and their dimension name) to match the order of the outputs in the Outputs menu in the GUI.

The value of the `checkAsInEntries` feature is a function from lists of words (`As`) to unit (`U`), raising an exception if any word `A` in `As` is not contained in the lexicon.

The value of the `as2ABRec` feature is a function from lists of words (`As`) to records mapping words (`A`) to bool (`B`). For all `A` in `As`, `B` is true if `A` is contained in the lexicon, and false otherwise.

The value of the `as2Entries` feature is a function from lists of words (`As`) to records mapping words (`A`) to lists of compiled lexical entries for that word (`Entries`).

The value of the `entry1Lat` feature is the lattice (`Lat`) corresponding to the type of a lexical entry.

The value of the `nodeLat` feature is the lattice (`Lat`) corresponding to the type of a node record, including the features `word`, `index`, `nodeSet` and `entryIndex`, and excluding the subrecords corresponding to the dimensions.

The value of the `node1Lat` feature is the lattice (`Lat`) corresponding to the type of a node record, including the features `word`, `index`, `nodeSet` and `entryIndex`, and the subrecords corresponding to the dimensions.

4.15 Lattice functors

Lattices are implemented as *abstract data types (ADTs)*. Each abstract data type is implemented as an Oz *functor*.

In this section, we start with an overview of the lattice functors in Section 4.15.1 [Lat-Overview], page 83, before we explain the individual lattice functors.[10]

4.15.1 Overview

The lattice functors are located in `Compiler/Lattices`. Each lattice functor is a record with at least the following features making up its interface:

```
elem(tag: A
     top: SL
     bot: SL
     glb: SLSL2SL
     select: SLsD2SL
     makeVar: U2SL
     encode: ILOpti2SL
     decode: SL2IL
     pretty: SLAbbrB2OL)
     count: U2Co
```

The `tag` feature corresponds to an atom (A)[11] denoting the name of the lattice.

The `top` feature corresponds to the top value of the lattice in the solver language (SL).

[10] In the explanation of the individual lattice functors, we omit the `top`, `bot`, `glb`, `encode`, `decode` and `pretty` features because their function explained elsewhere in this manual: `top`, `bot` and `glb` are explained Section 4.9 [Types reference], page 40. `encode`, `decode` and `pretty` are explained in Section 4.12 [IL syntax], page 62, Section 4.13 [SL syntax], page 74, and Section 9.36 [OL syntax], page 161 — they convert back and forth between these languages.

[11] We use the type abbreviations defined in Chapter 15 [Variable names], page 189.

The `bot` feature corresponds to the bottom value of the lattice in the solver language (SL).

The `glb` feature corresponds to the greatest lower bound function of the lattice. Its type is `SL SL -> SL`, i.e. it takes two SL values and yields their greatest lower bound (as a SL value). All greatest lower bound functions are commutative, thereby maintaining monotonicity.

The `select` feature corresponds to the selection function of the lattice which is used by the XDK solver to select one lexical entry from the set of lexical entries for a word. Its type is `SLs D -> SL`, i.e. it takes a list of SL values and a selector (an Oz finite domain variable), and yields a SL value.

The `makeVar` feature corresponds to the make variable function of the lattice which is used by the XDK solver to create variables of values. Its type is `U -> SL`, i.e. it takes no argument (as indicated by the type `U` for "unit") and yields an undetermined SL value.

The `encode` feature corresponds to the encoding function of the lattice. Its type is `IL Opti -> SL`, i.e. it takes an IL value and an optimization record `Opti` (introduced for efficiency reasons), and yields a SL value.

The `decode` feature corresponds to the decoding function of the lattice. Its type is `SL -> IL`, i.e. it takes a SL value and yields an IL value.

The `pretty` feature corresponds to the pretty function of the lattice. Its type is `SL AbbrB -> OL`, i.e. it takes a SL value and a boolean (`AbbrB`), and yields an OL value. `AbbrB` specifies whether the yielded OL value shall be abbreviated (`AbbrB==true`) or not (`AbbrB==false`).

The `count` feature corresponds to the counting function of the lattice. Its type is `U -> Co`, i.e. it takes no argument and yields a fd/fs variable count. This is used for profiling.

4.15.2 Card functor

4.15.2.1 Select function

The select function uses the selection constraint for Oz finite set variables to select one set of integers from the list of sets of integers.

4.15.2.2 Make variable function

The make variable function creates an undetermined Oz finite set variable.

4.15.2.3 Count function

The count function returns:

```
o(fd: 0
  fs: 1)
```

4.15.2.4 Additional interface features

None.

4.15.3 Domain functor

The domain lattice encodes constants from a finite domain as integers. Before encoding, the list of atoms is sorted lexically using `Value.'<'`.

4.15.3.1 Select function

The select function uses the selection constraint for Oz finite domain variables to select one value from the list of values.

4.15.3.2 Make variable function

The make variable function creates an Oz finite domain variable ranging over the domain of the lattice.

4.15.3.3 Count function

The count function returns:

```
o(fd: 1
  fs: 0)
```

4.15.3.4 Additional interface features

Below, we show the additional interface features of this lattice:

```
constants: As
card: CardI
dSpec: DSpec
a2I: A2I
i2A: I2A
```

The `constants` feature corresponds to the list of atoms which constitute the finite domain, sorted lexically.

The `card` feature corresponds to an integer denoting the cardinality of the finite domain.

The `dSpec` feature corresponds to the finite domain specification of the finite domain.

The `a2I` feature corresponds to a function from Oz atoms in the finite domain to their corresponding Oz integers.

The `i2A` feature corresponds to a function from Oz integers in the finite domain to their corresponding Oz atoms.

4.15.4 Flat lattice functor

The *flat lattice functor* defines basic functionality inherited by some of the other lattice functors.

4.15.4.1 Select function

The select function uses the selection constraint for Oz finite domain variables to select one value from the list of values.

4.15.4.2 Make variable function

The make variable function of this lattice creates an undetermined Oz variable.

4.15.4.3 Count function

The count function returns:

```
o(fd: 0
  fs: 0)
```

4.15.4.4 Additional interface features

None.

4.15.5 Integer functor

4.15.5.1 Select function

The select function uses the selection constraint for Oz finite domain variables to select one integer from the list of integers.

4.15.5.2 Make variable function

The make variable function creates an Oz finite domain variable ranging over all possible integers.

4.15.5.3 Count function

The count function returns:

```
o(fd: 1
  fs: 0)
```

4.15.5.4 Additional interface features

None.

4.15.6 List functor

4.15.6.1 Select function

See flat lattice (Section 4.15.4 [Lat-Flat], page 85).

4.15.6.2 Make variable function

See flat lattice (Section 4.15.4 [Lat-Flat], page 85).

4.15.6.3 Count function

See flat lattice (Section 4.15.4 [Lat-Flat], page 85).

4.15.6.4 Additional interface features

Below, we show the additional interface features of this lattice:

```
domain: Lat
```

The `domain` feature corresponds to the lattice of the list domain.

4.15.7 Record functor

4.15.7.1 Select function

The select function of the record lattice functor recursively calls the select functions of the lattices corresponding to its features.

4.15.7.2 Make variable function

The make variable function of the record lattice functor recursively calls the make variable functions of the lattices corresponding to its features.

4.15.7.3 Count function

The count function returns:

```
o(fd: FDI
  fs: FSI)
```

where FDI is the sum of the finite domain variables counted for the co-domains of the record, and FSI the sum of the finite set variables.

4.15.7.4 Additional interface features

Below, we show the additional interface features of this lattice:

```
record: ALatRec
```

The `record` feature is a record representing mapping the fields in the arity of the record (A) to the lattices corresponding to their types (Lat).

4.15.8 Set functor

The set lattice functor is defined differently depending on its domain, which can be:

1. a finite domain of constants or a tuple of which all projections are finite domains of constants
2. integer

4.15.8.1 Select function

1. The select function uses the selection constraint for Oz finite set variables to select one set of integers from the list of sets of integers.
2. See 1.

4.15.8.2 Make variable function

1. The make variable function creates an Oz finite set variable ranging over all possible sets of integers over the domain of the set.
2. See 1.

4.15.8.3 Count function

1. Returns:

   ```
   o(fd: 0
     fs: 1)
   ```

2. See 1.

4.15.8.4 Additional interface features

1.

   ```
   domain: Lat
   card: I
   setTypeA: A
   ```

   The `domain` feature is the lattice corresponding to the domain of the set.

   The `card` feature is the cardinality of the power set of the set.

   The `setTypeA` feature is either the atom `a` (accumulative set), or `i` (intersective set).

2.
```
domain: Lat
setTypeA: A
```

4.15.9 String functor

4.15.9.1 Select function

See flat lattice (Section 4.15.4 [Lat-Flat], page 85).

4.15.9.2 Make variable function

See flat lattice (Section 4.15.4 [Lat-Flat], page 85).

4.15.9.3 Count function

See flat lattice (Section 4.15.4 [Lat-Flat], page 85).

4.15.9.4 Additional interface features

None.

4.15.10 Tuple functor

For the tuple lattice functor, all projections must be finite domains of constants.

4.15.10.1 Select function

The select function uses the selection constraint for Oz finite domain variables to select one integer (encoding the tuple) from the list of integers (encoding the list of tuples).

4.15.10.2 Make variable function

The make variable function creates an Oz finite domain variable ranging over the domain of the lattice, a finite domain of integers encoding the tuples.

4.15.10.3 Count function

Returns:
```
o(fd: 1
  fs: 0)
```

4.15.10.4 Additional interface features

```
domains: Lats
card: I
dSpec: DSpec
i2As: I2As
```

The `domains` feature corresponds to a list of lattices which correspond to the projections of the tuple.

The `card` feature corresponds to the cardinality of the domain encoding the tuple.

The `dSpec` feature corresponds to the finite domain specification of the finite domain used to encode the tuple.

The `i2As` feature corresponds to a function from integers (encoding tuples) to lists of atoms representing the tuple corresponding to the integer.

5 Grammars

In this chapter, we describe the example grammars coming with the XDK. Note that the XTAG grammar generator Chapter 6 [XTAG], page 97 offers a larger-scale grammar than those describes in this chapter.

All grammars suffixed with `PW` use only principles compiled by Chapter 10 [PrincipleWriter], page 171.

5.1 ANBN

This grammar covers the context-free language of words with n as followed by n bs. It was written by Ralph Debusmann for his dissertation.

5.2 ANBNCN

This grammar covers the non-context-free language of words with n as followed by n bs followed by n cs. It was written by Ralph Debusmann for his dissertation.

5.3 ANBNCNPW

This grammar is the same as Section 5.2 [ANBNCN], page 89 except that it only uses PW principles.

5.4 ANBNPW

This grammar is the same as Section 5.1 [ANBN], page 89 except that it only uses PW principles.

5.5 Acl01

This grammar covers German subordinate clauses and word order variation therein. It uses the two TDG graph dimensions ID and LP, and was written by Denys Duchier and Ralph Debusmann, for their ACL 2001 paper *Topological Dependency Trees: A Constraint-Based Account of Linear Precedence* ([References], page 219), and is described therein.

5.6 Acl01PW

This grammar is the same as Section 5.5 [Acl01], page 89 except that it only uses PW principles.

5.7 Arabic

This is an Arabic grammar developed in a Forschungspraktikum by Marwan Odeh. It is described in *Topologische Dependenzgrammatik fuers Arabische* ([References], page 219).

5.8 Benoit

This is a French toy grammar written by Benoit Crabbe.

5.9 CSD

This grammar covers the language of n ns followed by n vs, where the ith n depends on the ith v, using the special principle `principle.csd` (Section 7.2.8 [CSD1], page 106). It is a demo grammar for cross-serial dependencies. It was written by Ralph Debusmann for his dissertation.

5.10 CSDPW

This grammar is the same as Section 5.9 [CSD], page 90 except that it only uses PW principles.

5.11 Chorus

This is a grammar for English covering the sentences of the old CHORUS demo system and more. It uses five graph dimensions: ID and LP (as in TDG), DS (Deep Syntax), PA (Predicate Argument) and SC (Scope). It was written by Ralph Debusmann and Stefan Thater. Parts of it are described in *A Relational Syntax-Semantics Interface Based on Dependency Grammar* ([References], page 219). It was used for the CHORUS demo for the “Begehung” in May 2004.

5.12 Diplom

This grammar covers many interesting German word order phenomena. It uses the two TDG graph dimensions ID and LP, and was written by Ralph Debusmann, for his Diplomarbeit *A Declarative Grammar Formalism For Dependency Grammar* ([References], page 219), and it described therein. An extended version can be found here (Section 5.50 [softproj], page 94).

5.13 Dutch

This grammar covers many interesting Dutch word order phenomena. It uses only the two TDG graph dimensions ID and LP, and was written by Ralph Debusmann, for the unpublished article *Topological Dependency Analysis of the Dutch Verb Cluster* ([References], page 219), and is described therein.

5.14 EQAB

This grammar covers the language of n as and n bs, in any order. It was written by Ralph Debusmann for his dissertation.

5.15 EQABPW

This grammar is the same as Section 5.14 [EQAB], page 90 except that it only uses PW principles.

5.16 ESSLLI04_id

This is the first part of the grammar developed in the ESSLLI 2004 course, using only the ID graph dimension. It was written by Ralph Debusmann and Denys Duchier. The ESSLLI 2004 course slides can be found here: `http://www.ps.uni-sb.de/~rade/talks.html`.

5.17 ESSLLI04_idPW

This grammar is the same as Section 5.16 [ESSLLI04_id], page 90 except that it only uses PW principles.

5.18 ESSLLI04_idlp

This is the second part of the grammar developed in the ESSLLI 2004 course, using the ID and LP graph dimensions. It was written by Ralph Debusmann and Denys Duchier. The ESSLLI 2004 course slides can be found here: `http://www.ps.uni-sb.de/~rade/talks.html`.

5.19 ESSLLI04_idlpPW

This grammar is the same as Section 5.18 [ESSLLI04_idlp], page 91 except that it only uses PW principles.

5.20 ESSLLI04_idlpds

This is the third part of the grammar developed in the ESSLLI 2004 course, using the ID, LP and DS graph dimensions. It was written by Ralph Debusmann and Denys Duchier. The ESSLLI 2004 course slides can be found here: `http://www.ps.uni-sb.de/~rade/talks.html`.

5.21 ESSLLI04_idlpdsPW

This grammar is the same as Section 5.20 [ESSLLI04_idlpds], page 91 except that it only uses PW principles.

5.22 ESSLLI04_idlpdspa

This is the fourth part of the grammar developed in the ESSLLI 2004 course, using the ID, LP, DS and PA graph dimensions. It was written by Ralph Debusmann and Denys Duchier. The ESSLLI 2004 course slides can be found here: `http://www.ps.uni-sb.de/~rade/talks.html`.

5.23 ESSLLI04_idlpdspaPW

This grammar is the same as Section 5.22 [ESSLLI04_idlpdspa], page 91 except that it only uses PW principles.

5.24 ESSLLI04_idlpdspasc

This is the fifth part of the grammar developed in the ESSLLI 2004 course, using the ID, LP, DS, PA and SC graph dimensions. It was written by Ralph Debusmann and Denys Duchier. The ESSLLI 2004 course slides can be found here: `http://www.ps.uni-sb.de/~rade/talks.html`.

5.25 ESSLLI04_idlpdspascPW

This grammar is the same as Section 5.24 [ESSLLI04_idlpdspasc], page 91 except that it only uses PW principles.

5.26 FG_TAGDC

This grammar combines TAG and Dominance Constraints. It was written by Ralph Debusmann.

5.27 FG_TAGDCgen

This grammar combines TAG and Dominance Constraints. It was written by Ralph Debusmann.

5.28 MTS10

This grammar models the combination of lexicalized context-free grammars from the paper at ESSLLI 2007 (Workshop: Model-Theoretic Syntax at 10). It was written by Ralph Debusmann.

5.29 MWE

This is a grammar for English covering the couple of sentences in the paper *Multiword expressions as dependency subgraphs* ([References], page 219) for the ACL 2004 Workshop on Multiword Expressions. It uses three graph dimensions: ID and LP (as in TDG) and PA (Predicate Argument). The paper and the grammar were written by Ralph Debusmann.

This is the grammar for the parsing direction.

5.30 MWEgen

This is a grammar for English covering the couple of sentences in the paper *Multiword expressions as dependency subgraphs* ([References], page 219) for the ACL 2004 Workshop on Multiword Expressions. It uses three graph dimensions: ID and LP (as in TDG) and PA (Predicate Argument). The paper and the grammar were written by Ralph Debusmann.

This is the grammar for the generation direction.

5.31 Negation

This is a grammar modeling French negation. It was written by Denys Duchier.

5.32 SAT

This is a grammar representing a reduction of the NP-complete SAT problem to XDG parsing, using the special principle `principle.pl` (Section 7.2.59 [PL], page 128). It was written by Ralph Debusmann and Gert Smolka, and is featured in Ralph Debusmann's dissertation and the paper *Multi-dimensional Dependency Grammar as Multigraph Description* ([References], page 219).

5.33 SATPW

This grammar is the same as Section 5.32 [SAT], page 92 except that it only uses PW principles.

5.34 SCR

This grammar covers the language of n ns followed by n vs, where the ith n depends on any of the vs. It is a demo grammar for scrambling. It was written by Ralph Debusmann for his dissertation.

5.35 SCRPW

This grammar is the same as Section 5.34 [SCR], page 93 except that it only uses PW principles.

5.36 TAG

This grammar showcases the new *Tree Adjoining Grammar (TAG)* encoding developed by Denys Duchier and Marco Kuhlmann, and was written by Marco Kuhlmann.

5.37 adjunction

This grammar showcases the old *Tree Adjoining Grammar (TAG)* encoding developed by Marco Kuhlmann, and was also written by him.

5.38 coindex

This is a grammar demoing `principle.coindex` (Section 7.2.11 [Coindex], page 107) for coindexing, which will probably be crucial for encoding FB-TAG in XDG. It was written by Ralph Debusmann.

5.39 diss

This is a superset of `Chorus.ul` (Section 5.11 [Chorus], page 90), including the treatment of information structure from `igk` (Section 5.44 [igk], page 94). It uses six graph dimensions: ID and LP (as in TDG), PA (Predicate Argument), SC (Scope), PS (Prosodic Structure) and IS (Information Structure). It also uses the lexicalized order principle `principle.order2` (Section 7.2.52 [Order2], page 126) instead of a non-lexicalized one. It was written by Ralph Debusmann for his dissertation.

5.40 dissPW.2dorder

This grammar is the same as Section 5.39 [diss], page 93 except that it only uses PW principles, and uses a new two-dimensional order principle. Adapted by Jorge Pelizzoni.

5.41 dissPW

This grammar is the same as Section 5.39 [diss], page 93 except that it only uses PW principles.

5.42 ema

This grammar covers some Czech sentences. It was written by Ondrej Bojar.

5.43 ema_th

This grammar covers some Czech sentences. The difference to the `ema` grammar is that this grammar includes an additional TH graph dimension (THematic) modeling tectogrammatical structure. It was written by Ondrej Bojar.

5.44 igk

This grammar covers a small fragment of English, and includes the two graph dimensions of Prosodic Structure (PS) and Information Structure (IS). It was written by Ralph Debusmann, in a project at the IGK annual meeting 2004 in Edinburgh together with Ciprian Gerstenberger, Oana Postolache, Stefan Thater and Maarika Traat. It is described in *A Modular Account of Information Structure in Extensible Dependency Grammar* ([References], page 219).

5.45 nut

This is an English toy grammar written for the nutshell chapter of the dissertation by Ralph Debusmann. Compared to Section 5.46 [nut1], page 94, it does not use lexical classes but generates the lexicon directly.

5.46 nut

This is an English toy grammar written for the nutshell chapter of the dissertation by Ralph Debusmann. Compared to Section 5.45 [nut], page 94, it uses lexical classes for economic lexicon generation.

5.47 nut1PW

This grammar is the same as Section 5.46 [nut1], page 94 except that it only uses PW principles.

5.48 nutPW

This grammar is the same as Section 5.45 [nut], page 94 except that it only uses PW principles.

5.49 regdgPW

This grammar models a regular dependency grammar as introduced in the ACL 2007 paper by Marco Kuhlmann and Mathias Moehl.

5.50 softproj

This grammar is an extension of the Diplom grammar (Section 5.12 [Diplom], page 90), and was developed in a Softwareprojekt by Regine Bader, Christine Foeldesi, Ulrich Pfeiffer and Jochen Steigner. It is described in *Modellierung grammatischer Phaenomene der deutschen Sprache mit Topologischer Dependenzgrammatik* ([References], page 219).

5.51 ww

This grammar covers the language $L = \{ww|win\{a,b\}\}$, using a *Tree Adjoining Grammar (TAG)* encoding developed by Ralph Debusmann and Marco Kuhlmann (the same encoding as used for the XTAG grammar generator of the XDK). It was written by Ralph Debusmann.

6 XTAG

The XTAG grammar generator generates XDG grammars from the TAG grammar developed in the XTAG project (`http://www.cis.upenn.edu/~xtag/`). For installing the relevant files from the XTAG grammar, see Chapter 3 [Installation], page 13 (optional installation bits, XTAG grammar generator, XTAG additional functionality).

The grammar generator uses the socket functionality of the XDK, which provides the possibility to read in grammars for specific input sentences from a server (over a socket connection).

To make it work, you need to take two steps:

1. Start a grammar generator server by entering the `XTAG` directory and then starting the server by typing:

   ```
   ./XTAGServer.exe -p 4712
   ```

   where the `-p` option determines the port which is taken by the server (default: 4712).
2. Start the XDK and open the grammar "file" `4712.ulsocket` (given the server runs on port 4712). Now, if the connection of the XDK and the server could be established, you can parse English sentences using the XTAG grammar. For each sentence, a new grammar is generated on-the-fly.

The full set of commandline arguments is the following:

- `--help` or `--nohelp` (short version: `-h`): Display an overview of the commandline arguments. Default: `--nohelp`.
- `--prune` or `--noprune` (`-r`): Prune tree lookup, i.e., when looking up the elementary trees for a word in the input, remove those multiply anchored trees where any of the additional anchors is not present in the input sentence. Pruning is used per default.
- `--filter none` or `--filter simple` or `--filter tagger` or `--filter supertagger` (`-f none` or `-f simple` or `-f tagger` or `-f supertagger`): Filter the set of elementary trees selected for the words in the input. `none` does not filter, `simple` uses a reimplementation of `simple_filter.pl`, the default tree filter from the lem parser distribution, `tagger` a reimplementation of `tagger_filter.pl`, and `supertagger` uses the supertagger available on the XTAG webpage. For the `tagger` option, the `mxpost` tagger by Adwait Ratnaparkhi must be installed in the directory denoted by the environment variable `MXPOST`. For the `supertagger` option, the environment variable `COREF` must point to the currently used data directory within the supertagger directory (as stated in the `README` there), e.g. to the `200K.data` directory. Default: `--filter none`.

The XDG grammars generated from the XTAG grammar make use of the principles:

- `principle.xTAG` (Section 7.2.70 [XTAG1], page 132)
- `principle.xTAGRedundant` (Section 7.2.72 [XTAGRedundant], page 134)
- `principle.xTAGRoot` (Section 7.2.73 [XTAGRoot], page 134)

And the output `output.xTAGDerivation` (Section 9.35 [XTAGDerivation], page 160) to display XTAG derivation trees using the tree viewer from the XTAG project lem parser.

7 Solver

This chapter presents the heart of the XDG development kit, viz. the XDK solver, a solver for specialized *constraint satisfaction problems (CSPs).* These CSPs are defined by:

1. a list of tokens that we call *words*
2. the *lexicon* which maps each word to a set of possible *lexical entries.*
3. the set of used *dimensions* and the set of *principles* used on them

The CSP is split up into a set of dimensions, each of which uses a set of principles.

The principles are either taken from a predefined *principle library*, or can be conveniently written in a First-Order Logic using the principle compiler Chapter 10 [PrincipleWriter], page 171.

A principle definition includes:

- the *principle identifier*
- the set of *dimension variables*
- the set of *argument variables* and their types
- *default values* for the argument variables
- the type of the *model record* introduced by the principle
- the set of *constraints* and their *priorities* which implement the principle. The higher the priority, the earlier the constraint is executed by the solver. By convention, node constraints have priority >=100 for constraints doing propagation, and <100 for those doing distribution.

We give an overview of the library in Principle overview (Section 7.1 [Principles overview], page 99). In Principle list (Section 7.2 [Principles list], page 103), we explain the whole range of principles in the current library.

Developers only: The *node record* is the internal representation of a node in a solution. We explicate it in Section 7.3 [Node record], page 134. In Section 7.4 [Writing new principles], page 135 we explain how to write new principles.

7.1 Principles overview

In this section, we give an overview of the *principle library.* Roughly, we divide the principles into families lumped together under explanatory themes.

7.1.1 Graph

This family defines graphs and further restricts them. It contains the following principles:

- Graph (Section 7.2.18 [Graph], page 110)
- GraphConstraints (Section 7.2.19 [GraphConstraints], page 111)
- GraphDist (Section 7.2.20 [GraphDist], page 111)
- Graph1 (Section 7.2.21 [Graph1], page 111)
- Graph1Constraints (Section 7.2.22 [Graph1Constraints], page 112)
- Graph1Dist (Section 7.2.23 [Graph1Dist], page 112)
- Dag (Section 7.2.13 [Dag], page 108)

- Tree (Section 7.2.67 [Tree], page 131)
- Tree1 (Section 7.2.68 [Tree1], page 132)

The Graph1 principle is more efficient than the Graph principle, but it is not compatible with some of the principles, e.g. the Valency principle (Section 7.2.69 [Valency1], page 132).

The Graph1Constraints and GraphConstraints omit non-deterministic *distribution*, which can be added on using resp. Graph1Dist and GraphDist. Omitting distribution can be useful if you do not want to enumerate the models on a particular dimension, e.g. for scope underspecification (cf. the Chorus grammar (Section 5.11 [Chorus], page 90)).

The Dag and Tree principles further restrict the graphs to be *directed acyclic graphs (dags)* or *trees*.

7.1.2 Valency

This family constrains the incoming and outgoing edges of nodes. It contains the following principles:

- In (Section 7.2.24 [In], page 112)
- In1 (Section 7.2.25 [In1], page 113)
- In2 (Section 7.2.26 [In2], page 113)
- Out (Section 7.2.58 [Out], page 127)
- Valency (Section 7.2.69 [Valency1], page 132)

7.1.3 Agreement

This family controls agreement and government. It contains the following principles:

- Agr (Section 7.2.1 [Agr], page 103)
- Agreement (Section 7.2.2 [Agreement], page 103)
- Agreement1 (Section 7.2.3 [Agreement1], page 104)
- AgreementSubset (Section 7.2.4 [AgreementSubset], page 104)
- Government (Section 7.2.16 [Government], page 109)
- Government1 (Section 7.2.17 [Government1], page 110)
- PartialAgreement (Section 7.2.60 [PartialAgreement], page 128)

Both the Agreement and the Government principles depend on the Agr principle.

7.1.4 Order

This family defines order on graphs and further restricts it. It contains the following principles:

- Order (Section 7.2.48 [Order], page 124)
- Order1 (Section 7.2.49 [Order1], page 125)
- Order1Constraints (Section 7.2.50 [Order1Constraints], page 125)
- Order1Dist (Section 7.2.51 [Order1Dist], page 126)
- Order2 (Section 7.2.52 [Order2], page 126)
- Order2Constraints (Section 7.2.53 [Order2Constraints], page 126)
- Order2Dist (Section 7.2.54 [Order2Dist], page 126)

- OrderConstraints (Section 7.2.55 [OrderConstraints], page 126)
- OrderDist (Section 7.2.56 [OrderDist], page 126)
- Projectivity (Section 7.2.61 [Projectivity], page 129)

The Order1 principle is more general than the Order principle: It orders sets of labels instead of just labels. The Order2 principle lexicalized but less efficient.

The Order1Constraints and OrderConstraints omit non-deterministic *distribution*, which can be added on using resp. Order1Dist and OrderDist.

The Parse principle further restricts the Order and Order1 principles to respect the order of the input. If it is not used, the input is regarded as a bag of words.

The SameOrder principle also further restricts the Order and Order1 principles. For all nodes, it equates the positions of the corresponding words.

7.1.5 Climbing

This family specifies and further restricts the climbing relation between two dimensions. It contains the following principles:

- Climbing (Section 7.2.10 [Climbing], page 107)
- Barriers (Section 7.2.5 [Barriers], page 104)
- BarriersAttrib (Section 7.2.6 [BarriersAttrib], page 105)
- BarriersLabels (Section 7.2.7 [BarriersLabels], page 105)

The Barriers, BarriersAttrib and BarriersLabels principles depends on the Climbing principle, and specifies further restrictions on climbing.

7.1.6 Linking

This is the family of *linking principles*, stipulating, for each edge from v to v' labeled l on `D1`, the corresponding path (from v) to v' on `D2`.

It contains the following principles:

- Linking12BelowStartEnd (Section 7.2.27 [Linking12BelowStartEnd], page 114)
- LinkingAbove (Section 7.2.28 [LinkingAbove], page 114)
- LinkingAboveBelow1or2Start (Section 7.2.29 [LinkingAboveBelow1or2Start], page 115)
- LinkingAboveEnd (Section 7.2.30 [LinkingAboveEnd], page 115)
- LinkingAboveStart (Section 7.2.31 [LinkingAboveStart], page 116)
- LinkingAboveStartEnd (Section 7.2.32 [LinkingAboveStartEnd], page 116)
- LinkingBelow (Section 7.2.33 [LinkingBelow], page 117)
- LinkingBelow1or2Start (Section 7.2.34 [LinkingBelow1or2Start], page 117)
- LinkingBelowEnd (Section 7.2.35 [LinkingBelowEnd], page 118)
- LinkingBelowStart (Section 7.2.36 [LinkingBelowStart], page 118)
- LinkingBelowStartEnd (Section 7.2.37 [LinkingBelowStartEnd], page 119)
- LinkingDaughter (Section 7.2.38 [LinkingDaughter], page 119)
- LinkingDaughterEnd (Section 7.2.39 [LinkingDaughterEnd], page 120)
- LinkingEnd (Section 7.2.40 [LinkingEnd], page 120)

- LinkingMother (Section 7.2.41 [LinkingMother], page 121)
- LinkingMotherEnd (Section 7.2.42 [LinkingMotherEnd], page 121)
- LinkingNotDaughter (Section 7.2.43 [LinkingNotDaughter], page 122)
- LinkingNotMother (Section 7.2.44 [LinkingNotMother], page 122)
- LinkingSisters (Section 7.2.45 [LinkingSisters], page 123)

Some principles (Section 7.2.43 [LinkingNotDaughter], page 122, Section 7.2.44 [LinkingNotMother], page 122, Section 7.2.45 [LinkingSisters], page 123 do not stipulate a path, but we dub them *linking principles* nonetheless since they follow a similar concept.

7.1.7 TAG encoding

This family includes constraints developed for the encoding of *Tree Adjoining Grammar (TAG)*. It includes the following constraints:

- OrderWithCuts (Section 7.2.57 [OrderWithCuts], page 126)
- TAG (Section 7.2.65 [TAG], page 130)
- XTAG (Section 7.2.70 [XTAG1], page 132)
- XTAGLinking (Section 7.2.71 [XTAGLinking], page 133)
- XTAGRedundant (Section 7.2.72 [XTAGRedundant], page 134)
- XTAGRoot (Section 7.2.73 [XTAGRoot], page 134)

7.1.8 Test

This principle can be used to test out constraints:

- Test (Section 7.2.66 [Test], page 131)

7.1.9 Miscellaneous

These are the remaining constraints, defying to be lumped together yet:

- Chorus (Section 7.2.9 [Chorus1], page 106)
- Coindex (Section 7.2.11 [Coindex], page 107)
- CSD (Section 7.2.8 [CSD1], page 106)
- Customs (Section 7.2.12 [Customs], page 107)
- Dutch (Section 7.2.14 [Dutch], page 108)
- Entries (Section 7.2.15 [Entries], page 109)
- LockingDaughters (Section 7.2.46 [LockingDaughters], page 123)
- LookRight (Section 7.2.47 [LookRight], page 124)
- PL (Section 7.2.59 [PL], page 128)
- Relative (Section 7.2.62 [Relative], page 129)
- SameEdges (Section 7.2.63 [SameEdges], page 129)
- Subgraphs (Section 7.2.64 [Subgraphs], page 130)

7.2 Principles list

In this section, we explain the whole range of principles in the predefined *principle library*. Note that the library is constantly changing, and that some principles, may still lack a description.

7.2.1 Agr principle

- identifier: `principle.agr`
- dimension variables: `D`
- argument variables:
  `Agr: tv(T)`
  `Agrs: iset(tv(T))`
- default values:
  `Agr: _.D.attrs.agr`
  `Agrs: _.D.entry.agrs`
- model record: empty
- constraints: `Agr` (priority 130)
- edge constraints: none

The Agr principle picks out one agreement value from a set of possible agreement values.

The type variable `tv(T)` is typically a tuple of e.g. person, number, gender etc.

The principle stipulates that:

- for all nodes, `Agr` is an element of `Agrs`

7.2.2 Agreement principle

- identifier: `principle.agreement`
- dimension variables: `D`
- argument variables:
  `Agr1: tv(T)`
  `Agr2: tv(T)`
  `Agree: set(label(D))`
- default values:
  `Agr1: ^.D.attrs.agr`
  `Agr2: _.D.attrs.agr`
  `Agree: ^.D.entry.agree`
- model record: empty
- constraints: `Agreement` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.

The Agreement principle establishes agreement between the nodes connected by an edge.

It stipulates that:

- for all edges labeled l, if l is in `Agree`, then `Agr1=Agr2`

7.2.3 Agreement1 principle

- identifier: `principle.agreement1`
- dimension variables: `D`, `D1`
- argument variables:
  `Agr1: tv(T)`
  `Agr2: tv(T)`
  `Agree: set(label(D))`
- default values:
  `Agr1: ^.D1.attrs.agr`
  `Agr2: _.D1.attrs.agr`
  `Agree: ^.D.entry.agree`
- model record: empty
- constraints: `Agreement` (priority 100)
- edge constraints: none

This is the same principle as Section 7.2.2 [Agreement], page 103 except that it has an additional dimension variable `D1`.

7.2.4 AgreementSubset principle

- identifier: `principle.agreementSubset`
- dimension variables: `D`
- argument variables:
  `Agr1: tv(T)`
  `Agr2: tv(T)`
  `Agree: set(label(D))`
- default values:
  `Agr1: ^.D.attrs.agr`
  `Agr2: _.D.attrs.agr`
  `Agree: ^.D.entry.agree`
- model record: empty
- constraints: `AgreementSubset` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.

The AgreementSubset principle establishes subset agreement between the nodes connected by an edge.

It stipulates that:

- for all edges labeled l, if l is in `Agree`, then `Agr2` is a subset of `Agr1`

7.2.5 Barriers principle

- identifier: `principle.barriers`
- dimension variables: `D1`, `D2`, `D3`

- argument variables:
  `Blocks: set(label(D2))`
- default values:
  `Blocks: _.D3.entry.blocks`
- model record: empty
- constraints: `Barriers` (priority 110)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

The `Blocks` argument variable defines the set of *blocked edge labels*.

The effect of the barriers principle is that nodes become "barriers" for other nodes, and effectively prohibits unbounded climbing. It is therefore most useful in conjunction with the Climbing principle (Section 7.2.10 [Climbing], page 107).

The principle creates for each node v a set of blocked nodes $blocked(v)$ which must stay below v on `D1`. v' is in $blocked(v)$ if it satisfies the conjunction of the following constraints:

- v' is below v on `D2`
- the incoming edge label of v' is one from the set of blocked edge labels of one of the nodes between v and v'

7.2.6 BarriersAttrib principle

- identifier: `principle.barriers.attrib`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Blocks: set(T) Attrib: set(T)`
- default values: none
- model record: empty
- constraints: `BarriersAttrib` (priority 110)
- edge constraints: none

This principle was written by Denys Duchier.

7.2.7 BarriersLabels principle

- identifier: `principle.barriers.labels`
- dimension variables: `D1`, `D2`, `D3`, `D4`
- argument variables:
  `Blocks: set(label(D3))`
- default values:
  `Blocks: _.D4.entry.blocks`
- model record: empty
- constraints: `BarriersLabels` (priority 110)
- edge constraints: none

This principle was written by Denys Duchier.

7.2.8 CSD principle

- identifier: `principle.csd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `NodeLabels: set(label(D2))`
- default values:
  `NodeLabels: {}`
- model record: empty
- constraints: `CSD` (priority 110)
- edge constraints: none

This principle supports the grammar for cross-serial dependencies `Grammars/CSD.ul` (Section 5.9 [CSD], page 90).

It stipulates the constraint that all noun dependents of a node v must follow the noun dependents of the nodes above v.

The `NodeLabels` argument variable determines the set of labels for noun dependents, e.g. `{n}` for `Grammars/CSD.ul` (Section 5.9 [CSD], page 90).

7.2.9 Chorus principle

- identifier: `principle.chorus`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Chorus: set(label(D1))`
- default values:
  `Chorus: _.D3.entry.chorus`
- model record: empty
- constraints: `Chorus` (priority 130)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[1]

It is fairly specialized, and is so far only used in the Chorus grammar (Section 5.11 [Chorus], page 90) for optimization (hence its name).

It creates for each node v the two sets $S1$ and $S2$. $S1$ is the set of nodes below edges labeled by l' in `Chorus` which emanate from v on `D1`. $S2$ is the set of nodes equal or below the mother of v on `D2`.

It then stipulates for all nodes v that $S1$ must be subset of $S2$.

[1] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D1` as it accesses the model record feature `downL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

7.2.10 Climbing principle

- identifier: `principle.climbing`
- dimension variables: `D1`, `D2`
- argument variables:
  `Subgraphs: bool`
  `MotherCards: bool`
- default values:
  `Subgraphs: true`
  `MotherCards: true`
- model record: empty
- constraints: `Climbing` (priority 110)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

The climbing principle stipulates that `D1` is a flatter graph than `D2`. Intuitively, that means that nodes on `D2` dimension can "climb up" and end up higher up on `D1`.

The argument variable `Subgraphs` specifies whether each node is required to take its entire subgraph along when migrating upwards (`true`), or not (`false`). Its default value is `true`.

The argument variable `MotherCards` specifies whether the for each node, the cardinalities of the sets of mothers on `D1` and `D2` must be equal (`true`), or not (`false`). This is an optimization for the case that both `D1` and `D2` are trees. If any of the two is not a tree, `MotherCards` should be set to `false`. The default value of `MotherCards` is `true`.

Climbing can be restricted by the Barriers principle (Section 7.2.5 [Barriers], page 104).

7.2.11 Coindex principle

- identifier: `principle.coindex`
- dimension variables: `D`
- argument variables: none
- default values: none
- model record: empty
- constraints: `Coindex` (priority 120)
  `CoindexEdge` (priority 100)
- edge constraints: none

This principle supports the grammar `Grammars/coindex.ul` (Section 5.38 [coindex], page 93).

7.2.12 Customs principle

- identifier: `principle.customs`
- dimension variables: `D`
- argument variables:
  `Customs: iset(label(D))`

- default values:
  `Customs: _.D.entry.customs`
- model record: empty
- constraints: `Customs` (priority 130)
- edge constraints: none

This principle was written by Ondrej Bojar.

7.2.13 Dag principle

- identifier: `principle.dag`
- dimension variables: `D`
- argument variables:
  `Connected: bool`
  `DisjointDaughters: bool`
- default values:
  `Connected: false`
  `DisjointDaughters: false`
- model record: empty
- constraints: `Dag` (priority 130)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.

The `Connected` argument variable is a boolean. Its default value is `false`. The `DisjointDaughters` argument variable is also a boolean. Its default value is `false`.

The dag principle states that the graph on dimension `D` must be a *directed acyclic graph (dag)*. If `Connected` is true, this dag must be connected, i.e., has only one root. If `DisjointDaughters` is true, then the sets of daughters must be disjoint, i.e., there can be no more than one outgoing edge to the same node.

This principle is less specific than the Tree principle (Section 7.2.67 [Tree], page 131).

7.2.14 Dutch principle

- identifier: `principle.dutch`
- dimension variables: `D1`, `D2`
- argument variables: none
- default values: none
- model record: empty
- constraints: `Dutch` (priority 110)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`. It also assumes that an order principle is used on `D1`, and that the set of edge labels on `D2` contains `subj`, `iobj` and `obj`.

The dutch principle is fairly specialized, and so far only used in the Dutch grammar (Section 7.2.14 [Dutch], page 108). It posits the conjunction of the following two constraints:

- for each node v, the subject (`subj`) daughter of v precedes the indirect object (`iobj`) daughter, which in turn precedes the direct object daughter (`obj`).
- for each node v, the set of noun daughters above v (i.e. those daughters of nodes above which have incoming edge label `subj`, `iobj` or `obj`) precede the set of noun daughters of v

7.2.15 Entries principle

- identifier: `principle.entries`
- dimension variables: none
- argument variables: none
- default values: none
- model record: empty
- constraints: `Entries` (priority 80)
- edge constraints: none

This principle can be applied on any dimension, but it is usually used on the lex dimension only.

The purpose of the Entries principle is to ensure that for each node, precisely one lexical entry is selected.

If you do not use this principle, and there are two lexical lexical entries for a word in the input which do not make a difference for the analysis, the XDK solver does not select one of the two. If you do use it, it does select one, i.e. it enumerates all possible lexical entries for a word in the input.

7.2.16 Government principle

- identifier: `principle.government`
- dimension variables: `D`
- argument variables:
  `Agr2: tv(T)`
  `Govern: vec(label(D) iset(tv(T)))`
- default values:
  `Agr2: _.D.attrs.agr`
  `Govern: ^.D.entry.govern`
- model record: empty
- constraints: `Government` (priority 100)
- edge constraints: none

The government principle establishes government between two nodes connected by an edge.

The type variable `tv(T)` is typically a tuple of e.g. person, number, gender etc.

It stipulates for all edges from v to v' labeled l on `D`:

- if `Govern(l)` is not empty, then `Agr2` must be in `Govern(l)`

7.2.17 Government1 principle

- identifier: `principle.government1`
- dimension variables: `D`, `D1`
- argument variables:
  `Agr2: tv(T)`
  `Govern: vec(label(D) iset(tv(T)))`
- default values:
  `Agr2: _.D1.attrs.agr`
  `Govern: ^.D.entry.govern`
- model record: empty
- constraints: `Government` (priority 100)
- edge constraints: none

This is the same principle as Section 7.2.16 [Government], page 109 except that it has additional dimension variable `D1`.

7.2.18 Graph principle

- identifier: `principle.graph`
- dimension variables: `D`
- argument variables: none
- default values: none
- model record:

```
{ mothers:    ints
  daughters:  ints
  up:         ints
  down:       ints
  index:      int
  eq:         ints
  equp:       ints
  eqdown:     ints
  labels:     set(label(D))
  mothersL:   vec(label(D) ints)
  upL:        vec(label(D) ints)
  daughtersL: vec(label(D) ints)
  downL:      vec(label(D) ints)}
```

- constraints: `GraphMakeNodes` (priority 130), `GraphConditions` (120), `GraphMakeEdges` (100) and `GraphDist` (90)
- edge constraints: none

The Graph principle introduces a model record with the following features for each node v:

- `mothers`: set of mothers of v
- `daughters`: set of daughters of v
- `up`: set of nodes above v

- `down`: set of nodes below v
- `index`: index of v
- `eq`: singleton set containing the index of v
- `equp`: set of nodes equal or above v
- `eqdown`: set of nodes equal or below v
- `labels`: set of incoming edge labels of v
- `mothersL`: set of mothers of v over an edge labeled l
- `upL`: set of nodes equal or above an edge into v labeled l
- `daughtersL`: set of daughters of v over an edge labeled l
- `downL`: set of nodes equal or below the daughters of v labeled l

The integers and sets of integers contain node indices.

The Graph principle states that `D` is a graph.

7.2.19 Graph principle (no distribution)

This principle is like the Graph principle (Section 7.2.18 [Graph], page 110), except that it does not apply non-deterministic *distribution*. That is, it does not use the constraint functor `GraphDist`.

7.2.20 Graph principle (only distribution)

This principle adds non-deterministic *distribution* to the GraphConstraints principle (Section 7.2.19 [GraphConstraints], page 111). The two principles together are equivalent to the Graph principle (Section 7.2.18 [Graph], page 110).

7.2.21 Graph1 principle

- identifier: `principle.graph1`
- dimension variables: `D`
- argument variables: none
- default values: none
- model record:

```
{ mothers:    ints
  daughters:  ints
  up:         ints
  down:       ints
  index:      int
  eq:         ints
  equp:       ints
  eqdown:     ints
  labels:     set(label(D))
  daughtersL: vec(label(D) ints)}
```

- constraints: `GraphMakeNodes1` (priority 130), `GraphConditions1` (120), `GraphMakeEdges1` (100) and `GraphDist` (90)
- edge constraints: none

The Graph1 principle introduces a model record with the following features for each node v:

- `mothers`: set of mothers of v
- `daughters`: set of daughters of v
- `up`: set of nodes above v
- `down`: set of nodes below v
- `index`: index of v
- `eq`: singleton set containing the index of v
- `equp`: set of nodes equal or above v
- `eqdown`: set of nodes equal or below v
- `labels`: set of incoming edge labels of v
- `daughtersL`: set of daughters of v over an edge labeled l

The integers and sets of integers contain node indices.

The Graph1 principle states that `D` is a graph.

The only difference of the Graph1 principle and the Graph principle (Section 7.2.18 [Graph], page 110) is that the Graph1 principle omits the model record features `mothersL`, `upL` and `downL`, and thereby improves solving efficiency. Most principles work in conjunction with both principles, but some (e.g. the Valency principle (Section 7.2.69 [Valency1], page 132)) do require the additional model record features of the Graph principle (Section 7.2.18 [Graph], page 110). Each such principle clearly states this particularity in its description in this document.

7.2.22 Graph1 principle (no distribution)

This principle is like the Graph1 principle (Section 7.2.21 [Graph1], page 111), except that it does not apply non-deterministic *distribution*. That is, it does not use the constraint functor `GraphDist`.

7.2.23 Graph1 principle (only distribution)

This principle adds non-deterministic *distribution* to the Graph1Constraints principle (Section 7.2.22 [Graph1Constraints], page 112). The two principles together are equivalent to the Graph1 principle (Section 7.2.21 [Graph1], page 111).

7.2.24 In principle

- identifier: `principle.in`
- dimension variables: `D`
- argument variables:
  `In: iset(label(D))`
- default values:
  `In: _.D.entry.in`
- model record: empty
- constraints: `In` (priority 130)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.

The `In` argument variable defines the set of *possible incoming edge labels*. Its default value is lexicalized by the lexical entry feature `in` on `D`.

It stipulates for all nodes v that:

- the set of incoming edge labels of v is a subset of the set of possible incoming edge labels

This principle is now mostly superseded by the Valency principle (Section 4.9.13 [Valency], page 47), but is still used for the classic grammars, e.g. the Acl01 grammar (Section 5.5 [Acl01], page 89).

Notice that the In2 principle (Section 7.2.26 [In2], page 113) uses the same constraint functor, but the type of the `In` argument variable is an accumulative set of labels on `D`, instead of an intersective one.

7.2.25 In1 principle

- identifier: `principle.in1`
- dimension variables: `D`
- argument variables:
  `In: valency(label(D))`
- default values:
  `In: _.D.entry.in`
- model record: empty
- constraints: `In1` (priority 130)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.[2]

The `In` argument variable defines the *incoming edge labels cardinality specification*. Its default value is lexicalized by the lexical entry feature `in` on `D`.

It stipulates for all nodes v that:

- the incoming edges of v must satisfy the incoming edge labels cardinality specification

The In1 principle is symmetric to the Out principle (Section 7.2.58 [Out], page 127), and is now mostly superseded by the Valency principle (Section 7.2.69 [Valency1], page 132).

7.2.26 In2 principle

- identifier: `principle.in2`
- dimension variables: `D`
- argument variables:
  `In: set(label(D))`

[2] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) as it accesses the model record feature `mothersL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

- default values:
  `In: _.D.entry.in`
- model record: empty
- constraints: `In` (priority 130)
- edge constraints: none

The only difference between this principle and the In principle (Section 7.2.24 [In], page 112) is that the type of the `In` argument variable is an accumulative set of labels on `D`, instead of an intersective one.

7.2.27 Linking12BelowStartEnd principle

- identifier: `principle.linking12BelowStartEnd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Start: vec(label(D1) set(label(D2)))`
  `End: vec(label(D1) set(label(D2)))`
- default values:
  `Start: ^.D3.entry.start`
  `End: ^.D3.entry.end`
- model record: empty
- constraints: `Linking12BelowStartEnd` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[3]

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraints:

- if `Start`(l) is not empty, then on `D2`, v' must be the daughter, or the daughter of the daughter of v, and the outgoing edge of v must be labeled by a label in `Start`(l)
- if `End`(l) is not empty, then on `D2`, v' must be the daughter, or the daughter of the daughter of v, and the incoming edge of v' must be in `End`(l)

That is, `Start`(l) and `End`(l) specify the direction, distance, startpoint and endpoint of the path from v to v' on `D2`.

7.2.28 LinkingAbove principle

- identifier: `principle.linkingAbove`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Which: set(label(D1))`
- default values:
  `Which: ^.D3.entry.which`

[3] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D2` as it accesses the model record feature `mothersL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

- model record: empty
- constraints: `LinkingAbove` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

The constraint for all edges from v to v' labeled l on `D1` is:

- if l in `Which`, then v' must be above v on `D2`

That is, `Which` specifies that the daughter v' of v on `D1` can be found above v on `D2`. In other words, `Which` specifies the direction of the path from v to v' on `D2`.

7.2.29 LinkingAboveBelow1or2Start principle

- identifier: `principle.linkingAboveBelow1or2Start`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Start: vec(label(D1) set(label(D2)))`
- default values:
  `Start: ^.D3.entry.start`
- model record: empty
- constraints: `LinkingBelow1or2Start` (priority 100)
- edge constraints: none

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraint:

- if `Start`(l) is not empty, then on `D2`, v' must be either the daughter or the daughter of the daughter of a node v'' above v, and the first edge on the path from v to v' must be labeled by a label in `Start`(l)

7.2.30 LinkingAboveEnd principle

- identifier: `principle.linkingAboveEnd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `End: vec(label(D1) set(label(D2)))`
- default values:
  `End: ^.D3.entry.end`
- model record: empty
- constraints: `LinkingAboveEnd` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[4]

[4] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D2` as it accesses the model record feature `downL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraint:

- if `End`(l) is not empty, then on `D2`, v' must be above v, and the outgoing edge of v' must be in `End`(l)

That is, `End`(l) specifies the direction and endpoint of the path from v to v' on `D2`.

7.2.31 LinkingAboveStart principle

- identifier: `principle.linkingAboveStart`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Start: vec(label(D1) set(label(D2)))`
- default values:
  `Start: ^.D3.entry.start`
- model record: empty
- constraints: `LinkingAboveStart` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[5]

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraint:

- if `Start`(l) is not empty, then on `D2`, v' must be above an edge into v labeled by a label in `Start`(l)

That is, `Start`(l) specifies the direction and startpoint of the path from v to v' on `D2`.

7.2.32 LinkingAboveStartEnd principle

- identifier: `principle.linkingAboveStartEnd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Start: vec(label(D1) set(label(D2)))`
  `End: vec(label(D1) set(label(D2)))`
- default values:
  `Start: ^.D3.entry.start`
  `End: ^.D3.entry.end`
- model record: empty
- constraints: `LinkingAboveStartEnd` (priority 100)
- edge constraints: none

[5] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D2` as it accesses the model record feature `upL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[6]

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraints:

- if `Start`(l) is not empty, then on `D2`, v' must be above an edge into v labeled by a label in `Start`(l)
- if `End`(l) is not empty, then on `D2`, v' must be above v, and the outgoing edge of v' must be in `End`(l)

That is, `Start`(l) and `End`(l) specify the direction, startpoint and endpoint of the path from v to v' on `D2`.

7.2.33 LinkingBelow principle

- identifier: `principle.linkingBelow`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Which: set(label(D1))`
- default values:
  `Which: ^.D3.entry.which`
- model record: empty
- constraints: `LinkingBelow` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

The constraint for all edges from v to v' labeled l on `D1` is:

- if l in `Which`, then v' must be below v on `D2`

That is, `Which` specifies that the daughter v' of v on `D1` can be found below v on `D2`. In other words, `Which` specifies the direction of the path from v to v' on `D2`.

7.2.34 LinkingBelow1or2Start principle

- identifier: `principle.linkingBelow1or2Start`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Start: vec(label(D1) set(label(D2)))`
- default values:
  `Start: ^.D3.entry.start`
- model record: empty
- constraints: `LinkingBelow1or2Start` (priority 100)

[6] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D2` as it accesses the model record features `downL` and `upL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

- edge constraints: none

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraint:

- if `Start`(l) is not empty, then on `D2`, v' must be either the daughter or the daughter of the daughter of v, and the first edge on the path from v to v' must be labeled by a label in `Start`(l)

That is, `Start`(l) specifies the direction and startpoint of the path from v to v' on `D2`.

7.2.35 LinkingBelowEnd principle

- identifier: `principle.linkingBelowEnd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `End: vec(label(D1) set(label(D2)))`
- default values:
  `End: ^.D3.entry.end`
- model record: empty
- constraints: `LinkingBelowEnd` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[7]

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraint:

- if `End`(l) is not empty, then on `D2`, v' must be below v, and the incoming edge of v' must be in `End`(l)

That is, `End`(l) specifies the direction and endpoint of the path from v to v' on `D2`.

7.2.36 LinkingBelowStart principle

- identifier: `principle.linkingBelowStart`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Start: vec(label(D1) set(label(D2)))`
- default values:
  `Start: ^.D3.entry.start`
- model record: empty
- constraints: `LinkingBelowStart` (priority 100)
- edge constraints: none

[7] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D2` as it accesses the model record feature `upL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[8]

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraint:

- if `Start`(l) is not empty, then on `D2`, v' must be below an edge emanating from v and labeled by a label in `Start`(l)

That is, `Start`(l) specifies the direction and startpoint of the path from v to v' on `D2`.

7.2.37 LinkingBelowStartEnd principle

- identifier: `principle.linkingBelowStartEnd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Start: vec(label(D1) set(label(D2)))`
  `End: vec(label(D1) set(label(D2)))`
- default values:
  `Start: ^.D3.entry.start`
  `End: ^.D3.entry.end`
- model record: empty
- constraints: `LinkingBelowStartEnd` (`D1`)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[9]

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraints:

- if `Start`(l) is not empty, then on `D2`, v' must be below an edge emanating from v and labeled by a label in `Start`(l)
- if `End`(l) is not empty, then on `D2`, v' must be below v, and the incoming edge of v' must be in `End`(l)

That is, `Start`(l) and `End`(l) specify the direction, startpoint and endpoint of the path from v to v' on `D2`.

7.2.38 LinkingDaughter principle

- identifier: `principle.linkingDaughter`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Which: set(label(D1))`

---

[8] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D2` as it accesses the model record feature `downL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

[9] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D2` as it accesses the model record features `downL` and `upL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

- default values:
  `Which: ^.D3.entry.which`
- model record: empty
- constraints: `LinkingDaughter` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

The constraint for all edges from v to v' labeled l on `D1` is:

- if l in `Which`, then v' must be a daughter of v on `D2`

That is, `Which` specifies that the daughter v' of v on `D1` can be found again as a daughter of v on `D2`. In other words, `Which` specifies the direction and distance of the path from v to v' on `D2`.

7.2.39 LinkingDaughterEnd principle

- identifier: `principle.linkingDaughterEnd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `End: vec(label(D1) set(label(D2)))`
- default values:
  `End: ^.D3.entry.end`
- model record: empty
- constraints: `LinkingDaughterEnd` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraint:

- if `End`(l) is not empty, then on `D2`, v' is a daughter of v whose incoming label is in `End`(l)

That is, `End`(l) specifies the direction, distance, and endpoint of the path from v to v' on `D2`.

7.2.40 LinkingEnd principle

- identifier: `principle.linkingEnd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `End: vec(label(D1) iset(label(D2)))`
- default values:
  `End: ^.D3.entry.end`
- model record: empty

- constraints: `LinkingEnd` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

The constraint for all edges from v to v' labeled l on `D1` is:

- if `End`(l) is not empty, then on `D2`, the incoming edge label of v' must be in `End`(l).

In other words, `End` specifies the *endpoint* of the path to v' on `D2`.

7.2.41 LinkingMother principle

- identifier: `principle.linkingMother`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Which: set(label(D1))`
- default values:
  `Which: ^.D3.entry.which`
- model record: empty
- constraints: `LinkingMother` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

The constraint for all edges from v to v' labeled l on `D1` is:

- if l in `Which`, then v' must be a mother of v on `D2`

That is, `Which` specifies that the daughter v' of v on `D1` can be found again as a mother of v on `D2`. In other words, `Which` specifies the direction and distance of the path from v to v' on `D2`.

7.2.42 LinkingMotherEnd principle

- identifier: `principle.linkingMotherEnd`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `End: vec(label(D1) set(label(D2)))`
- default values:
  `End: ^.D3.entry.end`
- model record: empty
- constraints: `LinkingMotherEnd` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

For all edges from v to v' labeled l on `D1`, it stipulates the constraint:

- if `End`(l) is not empty, then on `D2`, v' is a mother of v, and the incoming label of v is in `End`(l)

That is, `End`(l) specifies the direction, distance, and endpoint of the path from v to v' on `D2`.

7.2.43 LinkingNotDaughter principle

- identifier: `principle.linkingNotDaughter`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Which: set(label(D1))`
- default values:
  `Which: ^.D3.entry.which`
- model record: empty
- constraints: `LinkingNotDaughter` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

The constraint for all edges from v to v' labeled l on `D1` is:

- if l in `Which`, then v' must *not* be a daughter of v on `D2`

That is, `Which` specifies that the daughter v' of v on `D1` cannot be again a daughter of v on `D2`.

7.2.44 LinkingNotMother principle

- identifier: `principle.linkingNotMother`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Which: set(label(D1))`
- default values:
  `Which: ^.D3.entry.which`
- model record: empty
- constraints: `LinkingNotMother` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

The constraint for all edges from v to v' labeled l on `D1` is:

- if l in `Which`, then v' must *not* be a mother of v on `D2`

That is, `Which` specifies that the daughter v' of v on `D1` cannot be a mother of v on `D2`.

7.2.45 LinkingSisters principle

- identifier: `principle.linkingSisters`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Which: set(label(D1))`
- default values:
  `Which: ^.D3.entry.which`
- model record: empty
- constraints: `LinkingSisters` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

This principle is from the family of *linking principles*.

The constraint for all edges from v to v' labeled l on `D1` is:

- if l in `Which`, then there must be another node v'', being the mother of both v and v' on `D2`, i.e. v and v' must be sisters on `D2`

7.2.46 LockingDaughters principle

- identifier: `principle.lockingDaughters`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `LockDaughters: set(label(D1))`
  `ExceptAbove: set(label(D1))`
  `Key: set(label(D2))`
- default values:
  `LockDaughters: {}`
  `ExceptAbove: {}`
  `Key: {}`
- model record: empty
- constraints: `LockingDaughters` (priority 110)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D1`: it does not work with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D1`.

It states the constraint that for all nodes v, the dependents v' reachable on `D1` via an edge label l in the lexically specified set `LockDaughters` are "locked", i.e., on `D2`, they cannot be a dependent of any node except:

- v
- those nodes above v on `D1` reachable via an edge labeled l' in `ExceptAbove`
- those mothers of v' which enter v via an edge label labeled l' in `Key` on `D1`

7.2.47 LookRight principle

- identifier: `principle.lookright`
- dimension variables: `D`
- argument variables:
  `LookRight: vec("id.agrreq" iset(label(D)))`
- default values:
  `LookRight: _.D.entry.lookright`
- model record: empty
- constraints: `LookRight` (priority 130)
- edge constraints: none

This principle was written by Ondrej Bojar.

7.2.48 Order principle

- identifier: `principle.order`
- dimension variables: `D`
- argument variables:
  `On: iset(label(D))`
  `Order: list(label(D))`
  `Yields: bool`
- default values:
  `On: _.D.entry.on`
  `Order: []`
  `Yields: false`
- model record:

```
{ selfSet:  vec(label(D) ints)
  self:     label(D)
  pos:      ints
  yield:    ints
  yieldS:   ints
  yieldL:   vec(label(D) ints) }
```

- constraints: `OrderMakeNodes` (priority 130), `OrderConditions` (120), and `OrderDist` (90)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.

The argument variable `On` specifies the set of *node labels* for each node to position it with respect to its daughters. The default value is lexicalized by the lexical entry feature `on` on `D`.

The argument variable `Order` specifies a total order on a subset of the edge labels on `D`. Its default value is the empty list (i.e. nothing is ordered).

The argument variable `Yields` specifies whether the yields (`true`) or the daughters (`false`) of each node on `D` shall be ordered. Its default value is `false` (i.e. the yields are not ordered).

The order principle constrains the linear order of the nodes. In particular, it orders the daughters of each node according to their edge label, and positions the head with respect to the daughters using their node label. The `On` argument specifies the set of possible node labels for a node. The `Order` argument variable specifies a total order on a subset of the set of labels.

Notice that there is also the more general Order1 principle (Section 7.2.49 [Order1], page 125) where the `Order` argument variable specifies a total order on sets of labels.

The order principle is most efficient if the following is satisfied:

1. `Order` is an exhaustive total order on *all* edge labels of dimension `D`
2. `Yields` is `true`

Otherwise, the order principle uses a weaker constraint (`FS.int.seq` of the Mozart FS library instead of Denys Duchier's `Select.seqUnion` of his selection constraint package).

7.2.49 Order1 principle

- identifier: `principle.order1`
- dimension variables: `D`
- argument variables:
  `On: iset(label(D))`
  `Order: list(set(label(D)))`
  `Yields: bool`
- default values:
  `On: _.D.entry.on`
  `Order: []`
  `Yields: false`
- model record:

```
{ selfSet:  vec(label(D) ints)
  self:     label(D)
  pos:      ints
  yield:    ints
  yieldS:   ints
  yieldL:   vec(label(D) ints) }
```

- constraints: `OrderMakeNodes` (priority 130), `Order1Conditions` (120), and `OrderDist` (90)
- edge constraints: none

This principle is a generalization of the original Order principle (Section 7.2.48 [Order], page 124). Contrary to that, it the `Order` argument variable specifies an order on sets of labels, instead of just labels.

7.2.50 Order1 principle (no distribution)

This principle is like the Order1 principle (Section 7.2.49 [Order1], page 125), except that it does not apply non-deterministic *distribution*.

7.2.51 Order1 principle (no distribution)

This principle adds non-deterministic *distribution* to the Order1Constraints principle (Section 7.2.50 [Order1Constraints], page 125). The two principles together are equivalent to the Order1 principle (Section 7.2.49 [Order1], page 125).

7.2.52 Order2 principle

- identifier: `principle.order2`
- dimension variables: `D`
- argument variables:
  `Order: set(tuple(label(D)|{"^"} label(D)|{"^"}))`
  `Yields: bool`
- default values:
  `Order: _.D.entry.order`
  `Yields: false`
- model record:

```
{ pos:     ints
  yield:   ints
  yieldS:  ints
  yieldL:  vec(label(D) ints) }
```

- constraints: `Order2MakeNodes` (priority 130), `Order2Conditions` (120), and `Order2Dist` (90)
- edge constraints: none

This is the new lexicalized order principle used also in the thesis.

7.2.53 Order2 principle (no distribution)

This principle is like the Order2 principle (Section 7.2.52 [Order2], page 126), except that it does not apply non-deterministic *distribution*.

7.2.54 Order2 principle (no distribution)

This principle adds non-deterministic *distribution* to the Order2Constraints principle (Section 7.2.53 [Order2Constraints], page 126). The two principles together are equivalent to the Order2 principle (Section 7.2.52 [Order2], page 126).

7.2.55 Order principle (no distribution)

This principle is like the Order principle (Section 7.2.48 [Order], page 124), except that it does not apply non-deterministic *distribution*.

7.2.56 Order principle (no distribution)

This principle adds non-deterministic *distribution* to the OrderConstraints principle (Section 7.2.55 [OrderConstraints], page 126). The two principles together are equivalent to the Order principle (Section 7.2.48 [Order], page 124).

7.2.57 OrderWithCuts principle

- identifier: `principle.orderWithCuts`

- dimension variables: D
- argument variables:
  On: iset(label(D))
  Order: list(label(D))
  Cut: vec(label(D) set(label(D)))
  Paste: set(label(D))
- default values:
  On: _.D.entry.on
  Order: []
  Cut: _.D.entry.out
  Paste: _.D.entry.paste
- model record:

```
{ selfSet:   vec(label(D) ints)
  self:      label(D)
  pos:       ints
  yield:     ints
  yieldS:    ints
  yieldL:    vec(label(D) ints)
  pasteByL:  vec(label(D) ints)
  paste:     ints
  pasteL:    vec(label(D) ints)
  takeProjL: vec(label(D) ints)
  giveProjL: vec(label(D) ints)
  keepProjL: vec(label(D) ints)
}
```

- constraints: OrderWithCutsMakeNodes (priority 130), OrderWithCutsConditions (120), and OrderWithCutsDist (90)
- edge constraints: none

This principle was written by Marco Kuhlmann.

7.2.58 Out principle

- identifier: principle.out
- dimension variables: D
- argument variables:
  Out: valency(label(D))
- default values:
  Out: _.D.entry.out
- model record: empty
- constraints: Out (priority 130)

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension D.

The Out argument variable defines the set of *possible outgoing edge labels*. Its default value is lexicalized by the lexical entry feature out on D.

It stipulates for all nodes v that:

- the outgoing edges of v must satisfy the outgoing edge labels cardinality specification

The Out principle is symmetric to the In1 principle (Section 7.2.25 [In1], page 113), and is now mostly superseded by the Valency principle (Section 7.2.69 [Valency1], page 132).

7.2.59 PL principle

- identifier: `principle.pl`
- dimension variables: `D`
- argument variables: none
- default values: none
- model record: empty
- constraints: `PL` (priority 120)
- edge constraints: none

This principle contains constraint necessary for the reduction of the NP-complete SAT problem to XDG parsing in grammar `Grammars/SAT.ul` (Section 5.32 [SAT], page 92).

7.2.60 PartialAgreement principle

- identifier: `principle.partialAgreement`
- dimension variables: `D`
- argument variables:
  `Agr1: tv(T)`
  `Agr2: tv(T)`
  `Agree: set(label(D))`
  `Projs: ints`
- default values:
  `Agr1: ^.D.attrs.agr`
  `Agr2: _.D.attrs.agr`
  `Agree: ^.D.entry.agree`
  `Projs: {}`
- model record: empty
- constraints: `PartialAgreement` (priority 100)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.

The PartialAgreement principle establishes partial agreement between the nodes connected by an edge, similar to Agreement principle `principle.agreement` (Section 7.2.2 [Agreement], page 103), which establishes complete agreement. The argument variable `Projs` represents a set of integers which are the projections of the agreement tuple which must agree.

It stipulates that:

- for all edges labeled l, if l is in `Agree`, then for all projections i in `Projs`, `Agr1.i=Agr2.i`

7.2.61 Projectivity principle

- identifier: `principle.projectivity`
- dimension variables: `D`
- argument variables: none
- default values: none
- model record: empty
- constraints: `Projectivity` (priority 130)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.

The operation of this principle depends on whether on `D`, a principle is used that introduces the `yield` feature, i.e., typically an order principle. If the `yield` feature is present, then the principle states that for each node on `D`, its `yield` set must be convex (i.e. without holes), otherwise, the `eqdown` set must be convex.

7.2.62 Relative principle

- identifier: `principle.relative`
- dimension variables: `D1`, `D2`
- argument variables: none
- default values: none
- model record: empty
- constraints: `Relative` (priority 110)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`. It also assumes that `D1` defines a node attribute `agr` whose type is a tuple of at least three domains. It also assumes that `D1` defines a node attribute `cat` whose type is a domain including at least the constant `prels`. It also assumes that `D1` includes the constant `rel` in its set of edge labels, and that `D2` includes `rvf`.

The dutch principle is fairly specialized, and so far only used in the Diplom grammar (Section 5.12 [Diplom], page 90). It stipulates the following constraints:

- a node has category `prels` iff it is a relative pronoun
- a node has incoming edge label `rel` iff it has a relative pronoun below itself
- on `D2`, the set of nodes equal or below the relative pronoun is equal to the set of nodes equal or below the `rvf`-daughter
- the relative pronoun of the node agrees partially (person, gender, number only, not def, case) with its ID mother

7.2.63 SameEdges principle

- identifier: `principle.relative`
- dimension variables: `D1`, `D2`
- argument variables: none

- default values: none
- model record: empty
- constraints: `SameEdges` (priority 130)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.

It states that the graph on dimension `D1` must have the same edges as the graph on dimension `D2`.

7.2.64 Subgraphs principle

- identifier: `principle.subgraphs`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Start: vec(label(D1) set(label(D2)))`
- default values:
  `Start: ^.D3.entry.start`
- model record: empty
- constraints: `Subgraphs` (priority 110)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimensions `D1` and `D2`.[10]

The principle creates two sets $S1$ and $S2$ for each node v, and each edge label l on `D1`: $S1$ is the set of nodes (i.e. the subgraph) below an edge labeled l emanating from v on `D1`. $S2$ is the set of nodes below all edges labeled by a label in `Start(l)` emanating from v on `D2`.

The principle then stipulates for each node v, and each edge label l on `D1`:

- if `Start(l)` is not empty, then $S1$ must be a subset of $S2$

7.2.65 TAG principle

- identifier: `principle.tag`
- dimension variables: `D`
- argument variables:
  `Anchor: iset(label(D))`
  `Dominates: vec(label(D) set(label(D)))`
  `Foot: set(label(D))`
  `Leaves: set(label(D))`
  `Order: list(label(D))`

[10] This principle does not work in conjunction with the Graph1 principle (Section 7.2.21 [Graph1], page 111) on `D1` and `D2` as it accesses the model record feature `downL` only introduced by the Graph principle (Section 7.2.18 [Graph], page 110).

- default values:
  `Anchor: _.D.entry.anchor`
  `Dominates: _.D.entry.dominates`
  `Foot: _.D.entry.foot`
  `Leaves: _.D.entry.leaves`
  `Order: []`
- model record:

```
{ anchorsL:    vec(label(D) ints)
  belowL:      vec(label(D) ints)
  pasteL:      vec(label(D) ints)
  pastedL:     vec(label(D) ints)
  yieldL:      vec(label(D) ints)
  leaveYieldL: vec(label(D) ints)
  yield:       ints
}
```

- constraints: `TAGMakeNodes` (priority 130), `TAGConditions` (120), and `TAGDist` (90)
- edge constraints: none

This principle was written by Marco Kuhlmann.

7.2.66 Test principle

- identifier: `principle.test`
- dimension variables: none
- argument variables: none
- default values: none
- model record: empty
- constraints: `Test` (priority 120)
- edge constraints: none

This principle can be used to try out new constraints, without having to worry yet about the principle definition, makefiles etc. After trying out, principles should however be properly integrated into the XDK, as described in Writing new principles (Section 7.4 [Writing new principles], page 135).

7.2.67 Tree principle

- identifier: `principle.tree`
- dimension variables: `D`
- argument variables: none
- default values: none
- model record: empty
- constraints: `TreeMakeNodes` (priority 130) and `TreeConditions` (120)
- edge constraints: none

This principle assumes that the Graph principle (Section 7.2.18 [Graph], page 110) is used on dimension `D`.

The principle stipulates states that D must be a *tree*.

This principle is more specific than the Dag principle (Section 7.2.13 [Dag], page 108).

7.2.68 Tree1 principle

- identifier: `principle.tree1`
- dimension variables: `D`
- argument variables: none
- default values: none
- model record: empty
- constraints: `TreeMakeNodes` (priority 130) and `TreeConditions1` (120)
- edge constraints: none

This principle is equivalent to the Tree principle (Section 7.2.67 [Tree], page 131). The only difference is that it uses the constraint functor `TreeConditions1` instead of `TreeConditions`, which includes a native treeness constraint that can lead to better propagation.

7.2.69 Valency principle

- identifier: `principle.valency`
- dimension variables: `D`
- argument variables:
  `In: valency(label(D))`
  `Out: valency(label(D))`
- default values:
  `In: _.D.entry.in`
  `Out: _.D.entry.out`
- model record: empty
- constraints: `In1` (priority 130), `Out` (priority 130)

This principle combines the In1 principle (Section 7.2.25 [In1], page 113) and the Out principle (Section 7.2.58 [Out], page 127) in one, and is thus easier to use.

7.2.70 XTAG principle

- identifier: `principle.xTAG`
- dimension variables: `D`
- argument variables:
  `Anchor: label(D)`
  `Foot: set(label(D))`
- default values:
  `Anchor: _.D.entry.anchor`
  `Foot: _.D.entry.foot`
- model record:

```
{ cover:  ints
  coverL:   vec(label(D) ints)
  foot:     ints }
```

- constraints: `XTAG` (priority 120)
- edge constraints: none

This principle is the core of the TAG encoding developed by Ralph Debusmann and Marco Kuhlmann for the XTAG grammar generator of the XDK. It contains the non-redundant constraints required for the encoding, whereas Section 7.2.72 [XTAGRedundant], page 134 contains the redundant ones.

The edge labels on dimension `D` must be Gorn addresses.

The argument variable `Anchor` encodes the Gorn address of the anchor in the tree corresponding to the node, `Foot` the singleton set containing the foot node of the tree and the empty set if there is none.

The model record feature `cover` models for each node v the set of nodes that are "covered" by v, i.e., the nodes below it on `D` plus, if it has been adjoined, those which it has "cut out" from the tree into which it has been adjoined, and which are then "pasted" into itself at the foot node (according to the dominance relation on Gorn addresses). For each node v `cover`(v) is convex.

The model record feature `coverL` models the partition of `cover` sorted by edge labels/Gorn addresses.

The model record feature `foot` models the set of nodes "pasted" into the tree at its foot node (if any).

The principle orders the partition `coverL` according to the precedence relation on Gorn addresses.

7.2.71 XTAGLinking principle

- identifier: `principle.xTAGLinking`
- dimension variables: `D1`, `D2`, `D3`
- argument variables:
  `Link: vec(label(D2) iset(label(D1)))`
- default values:
  `Link: ^.D3.entry.end`
- model record: empty
- constraints: `XTAGLinking` (priority 100)
- edge constraints: none

This principle states two constraints:

1. The graph on dimension `D1` must have the same edges as the graph on dimension `D2` (same as Section 7.2.63 [SameEdges], page 129)
2. For all edges from v to v' labeled l on `D2` it must be the case that on `D1`, the incoming edge label of v' is in `Link`(l). That is, contrary to the less strict Section 7.2.40 [LinkingEnd], page 120 principles, the constraint must hold regardless of whether `Link`(l) is non-empty or not. Another difference to Section 7.2.40 [LinkingEnd], page 120 is that the order of the dimensions `D1` and `D2` is reversed.

7.2.72 XTAGRedundant principle

- identifier: `principle.xTAGRedundant`
- dimension variables: `D1`, `D2`
- argument variables:
  `Anchor: label(D2)`
  `Foot: set(label(D2))`
- default values:
  `Anchor: _.D.entry.anchor`
  `Foot: _.D.entry.foot`
- model record: empty
- constraints: `XTAGRedundant` (priority 120)
- edge constraints: none

This principle extends the principle Section 7.2.70 [XTAG1], page 132 with redundant constraints to improve propagation.

7.2.73 XTAGRoot principle

- identifier: `principle.xTAGRoot`
- dimension variables: `D`
- argument variables: none
- default values: none
- model record: empty
- constraints: `XTAGRoot` (priority 120)
- edge constraints: none

This principle states that the tree at the root of each derivation must be labeled by category `S`. I.e., in the current encoding, the lexical `in` value of the corresponding lexical entry must include the label `S_s` (category `S`, initial tree).

7.3 Node record

The *node record* is the internal representation of a node in the XDK solver. Each node corresponds to a word in the input.

The node record is defined as follows:

```
o(word: A
  index: I
  nodeSet: M
  entryIndex: I
  <dimension identifier 1>: o(attrs: SL
                              entry: SL
                              model: SL)
  ...
  <dimension identifier n>: o(attrs: SL
                              entry: SL
                              model: SL))
```

The value of the `word` feature is an Oz atom representing the word.

The value of the `index` feature is an Oz integer representing the unique *node index* of the node.

The value of the `nodeSet` feature is an Oz finite set of integers representing the set of node indices of all nodes.

The value of the `entryIndex` feature is the an Oz integer representing the *entry index* of the node. Notice that the entry index is different from the node index: Each word corresponds to a set of lexical entries, and each of these has an entry index. The entry index of the node is the index of the selected lexical entry for the node.

The next features introduce sub records for the used dimensions `<dimension identifier i>` (`1<=i<=n`). Each of these sub records has the features `attrs`, `entry` and `model`. The value of the `attrs` feature is the attributes record of the node. The value of the `entry` feature is the entry record of the node. The value of the `model` feature is the model record of the node.

7.4 Writing new principles

In this section, we explain how you can write new principles. Principles must be written in Mozart/Oz. In order to write an principle, you need to provide two things:

1. the *principle definition functor*
2. the *constraint functors*

7.4.1 Principle definition functor

You write the principle definition in the IL, and embed it into a functor. The functor has to export the principle definition as `Principle`, and has to reside in `Solver/Principles`.

7.4.1.1 Example (graph principle)

We display an example principle definition functor of the graph principle (`Solver/Principles/Graph.oz`) below:

```
%% Copyright 2001-2008
%% by Ralph Debusmann <rade@ps.uni-sb.de> (Saarland University) and
%%    Denys Duchier <duchier@ps.uni-sb.de> (LIFO, Orleans) and
%%    Jorge Marques Pelizzoni <jpeliz@icmc.usp.br> (ICMC, Sao Paulo) and
%%    Jochen Setz <info@jochensetz.de> (Saarland University)
%%
functor
export
   Principle
define
   Principle =
   elem(tag: principledef
id: elem(tag: constant
 data: 'principle.graph')
dimensions: [elem(tag: variable
  data: 'D')]
```

```
model:
   elem(tag: 'type.record'
args:
   [elem(tag: constant
 data: 'mothers')#
    elem(tag: 'type.ints')
    %%
    elem(tag: constant
 data: 'daughters')#
    elem(tag: 'type.ints')
    %%
    elem(tag: constant
 data: 'up')#
    elem(tag: 'type.ints')
    %%
    elem(tag: constant
 data: 'down')#
    elem(tag: 'type.ints')
    %%
    elem(tag: constant
 data: 'eq')#
    elem(tag: 'type.ints')
    %%
    elem(tag: constant
 data: 'equp')#
    elem(tag: 'type.ints')
    %%
    elem(tag: constant
 data: 'eqdown')#
    elem(tag: 'type.ints')
    %%
    elem(tag: constant
 data: 'labels')#
    elem(tag: 'type.set'
 arg: elem(tag: 'type.labelref'
   arg: elem(tag: variable
     data: 'D')))
    %%
    elem(tag: constant
 data: 'mothersL')#
    elem(tag: 'type.vec'
 arg1: elem(tag: 'type.labelref'
    arg: elem(tag: variable
      data: 'D'))
 arg2: elem(tag: 'type.ints'))
    %%
    elem(tag: constant
```

```
 data: 'daughtersL')#
    elem(tag: 'type.vec'
 arg1: elem(tag: 'type.labelref'
    arg: elem(tag: variable
      data: 'D'))
 arg2: elem(tag: 'type.ints'))
    %%
    elem(tag: constant
 data: 'upL')#
    elem(tag: 'type.vec'
 arg1: elem(tag: 'type.labelref'
    arg: elem(tag: variable
      data: 'D'))
 arg2: elem(tag: 'type.ints'))
    %%
    elem(tag: constant
 data: 'downL')#
    elem(tag: 'type.vec'
 arg1: elem(tag: 'type.labelref'
    arg: elem(tag: variable
      data: 'D'))
 arg2: elem(tag: 'type.ints'))
   ])
constraints: [elem(tag: constant
   data: 'GraphMakeNodes')#
      elem(tag: integer
   data: 130)
      %%
      elem(tag: constant
   data: 'GraphConditions')#
      elem(tag: integer
   data: 120)
      %%
      elem(tag: constant
   data: 'GraphMakeEdges')#
      elem(tag: integer
   data: 100)
      %%
      elem(tag: constant
   data: 'GraphDist')#
      elem(tag: integer
   data: 90)
     ]
       )
end
```

The value of the `id` feature is an IL constant representing the unique principle identifier (here: `'principle.graph'`).

The value of the `dimensions` feature is a list of IL variables representing the dimension variables introduced by the principle (here: `'D'`).

The value of the `model` feature is a list of IL pairs representing the type of the model record introduced by the principle. Here, the graph principle introduces the following features:

- `mothers` (a set of integers)
- `daughters` (a set of integers)
- `up` (a set of integers)
- `down` (a set of integers)
- `eq` (a set of integers)
- `equp` (a set of integers)
- `eqdown` (a set of integers)
- `labels` (a set of edge labels on dimension `D`)
- `mothersL` (a mapping from edge labels on dimension `D` to sets of integers)
- `daughtersL` (a mapping from edge labels on dimension `D` to sets of integers)
- `upL` (a mapping from edge labels on dimension `D` to sets of integers)
- `downL` (a mapping from edge labels on dimension `D` to sets of integers)

The value of the `constraints` feature is a list of pairs of an IL constant and an IL integer, representing a mapping from constraint functor file names (modulo the suffix `.ozf`) to their priorities. Here, the constraint functor `GraphMakeNodes` has priority 130, `GraphConditions` 120, `GraphMakeEdges` 100, and `GraphDist` 90. The XDK solver invokes constraint functors with higher priority first. Changing the priority of the constraint functors can lead to less/more efficient solving.

7.4.1.2 Example (out principle)

The graph principle does not have any arguments. Therefore, we display another example principle definition functor of the out principle (`Solver/Principles/Out.oz`), since it does have an argument:

```
%% Copyright 2001-2008
%% by Ralph Debusmann <rade@ps.uni-sb.de> (Saarland University) and
%%    Denys Duchier <duchier@ps.uni-sb.de> (LIFO, Orleans) and
%%    Jorge Marques Pelizzoni <jpeliz@icmc.usp.br> (ICMC, Sao Paulo) and
%%    Jochen Setz <info@jochensetz.de> (Saarland University)
%%
functor
export
   Principle
define
   Principle =
   elem(tag: principledef
id: elem(tag: constant
```

```
     data: 'principle.out')
    dimensions: [elem(tag: variable
      data: 'D')]
    args: [elem(tag: variable
        data: 'Out')#
           elem(tag: 'type.valency'
        arg: elem(tag: 'type.labelref'
          arg: elem(tag: variable
    data: 'D')))]
    defaults: [elem(tag: variable
    data: 'Out')#
       elem(tag: featurepath
    root: '_'
    dimension: elem(tag: variable
    data: 'D')
    aspect: 'entry'
    fields: [elem(tag: constant
          data: 'out')])]
    constraints: [elem(tag: constant
       data: 'Out')#
          elem(tag: integer
       data: 130)])
    end
```

Here, the principle identifier is `'principle.out'`, the dimension variable is `D`, and the principle makes use of the constraint functor `Out` with priority `130`.

The value of the `args` feature is a list of IL pairs representing the argument variables introduced by the principle and their types. Here, the out principle introduces the argument `Out` (a valency of edge labels on dimension `D`).

The value of the `default` feature is a list of IL pairs representing the default values of a subset of the argument variables introduced by the principle. Here, the default value of the `Out` argument is a feature path.

7.4.1.3 Integrate the principle definition

To integrate the principle definition into the XDK, you can either trust the perl-script `scripts/addprinciple`, or, by hand do the following:

- add the principle definition functor to the `ozmake` makefile `Solver/Principles/makefile.oz`
- add the principle definition functor to the imported functors of the functor `Solver/Principles/Principles.oz`, and also to the list `Principles` on top of `Solver/Principles/Principles.oz`.
- add the identifier of the new principle to the XML file `Solver/Principles/principles.xml`. Here, for each new principle, you add a line like the following for the graph principle:

```
<principleDef id="principle.graph"/>
```

This step is necessary because XML language grammar files contain only principle uses, but not principle definitions. Therefore, the principle identifiers of the used principles are only referred to but not defined in XML language grammar files, which leads to errors running an XML validator on them.

- optionally (but highly encouraged): add the principle definition to the manual, files `solver-principles_overview.texi` and `solver-principles_list.texi`

7.4.2 Constraint functors

You write a constraint functor as an Oz functor exporting the procedure `Constraint: Nodes G Principle FD FS Select -> U`.

`Constraint` has six arguments:

- `Nodes` is a list of node records representing the solution
- `G` is the current grammar
- `Principle` is the current principle instantiation
- `FD` is the FD functor (finite domain constraints)
- `FS` is the FS functor (finite set constraints)
- `Select` is the Select functor (selection constraints)

where `Principle` is a record of the following type:

```
principle(pIDA: PIDA
          modelLat: ModelLat
          dVA2DIDA: DVA2DIDA
          dIDAs: DIDAs
          argRecProc: ArgRecProc
          argsLat: ArgsLat
          dIDA: DIDA
          ...)
```

where we omit the features `modelTn`, `constraints`, `edgeconstraints`, `dVA2DIDARec`, `argRec` and `argsTn`, which are irrelevant for writing constraint functors.

where the value of `pIDA` is the principle identifier (e.g. `principle.valency`.

The value of `modelLat` is the lattice for the model record.

The value of `dVA2DIDA` is a function mapping dimension variables to dimension identifiers.

The value of `argRecProc` is a function of type `ArgRecProc: A AXRec -> SL` from a principle argument variable (`A`) and a record (`AXRec`) to the argument value bound to the argument variable.

The value of `argsLat` is the lattice for the argument record of the principle.

The value of `DIDA` is the dimension identifier of the dimension on which the principle is instantiated.

By convention, you should only access the argument values of a principle use through the `ArgRecProc` function. The record `AXRec` is a mapping from either Oz atom '_', '^', or 'ql' to either a node record ('_' and '^') or an Oz atom representing an edge label ('ql'). I.e., the record `AXRec` includes two mappings in one record:

- By mapping the root variables '_' and '^' to a node record, we bind bind the root variables of feature paths.

- By mapping 'ql' to an Oz atom representing an edge label, we bind the edge label variable to an edge label.

In a constraint functor, you usually posit constraints over all nodes. In order to get a principle argument, you usually call `ArgRecProc` providing only the mapping of root variable `'_'` to the current node.

7.4.2.1 Example (binding root variables)

Here is an example of using `ArgRecProc` to get the value of the `Out` argument of the out principle, taken from the constraint functor implementing the out principle (`Solver/Principles/Lib/Out.oz`):

```
for Node in Nodes do
   LAOutMRec = {ArgRecProc 'Out' o('_': Node)}
in
   ...
end
```

Here, in a `for` loop, the constraint functor posits a constraint over each node `Node` (in `Nodes`). The root variable '_' is bound to `Node` in the call of the `ArgRecProc` function to get the value of the argument variable `Out` for node `Node`.

7.4.2.2 Check model helper procedure

The functor `Solver/Principles/Lib/Helpers.oz` exports the procedure `CheckModel: V Nodes DIDAATups -> B`. By convention, `CheckModel` should be used to check whether all model record features accessed in the constraint functor are actually present (in order to prevent crashes of the XDK solver).

The `CheckModel` procedure has three arguments:

- `V`: an Oz virtual string representing the file name of the constraint functor invoking the `CheckModel` procedure. This way the procedure knows in which constraint functor an error has occurred if there is one.
- `Nodes`: a list of node records representing the solution.
- `DIDAATups`: a list of pairs `DIDA#A` of a dimension identifier (`DIDA`) and an Oz atom (`A`).

The procedure returns `true` if for a node `Node` (in `Nodes`), it is the case that for all pairs `DIDA#A` in `DIDAATups`, the model record of the node (`Node.DIDA.model`) has field `A`. If the model record lacks a feature, it returns `false` and the procedure prints out a warning to stderr.

By convention, you should call `CheckModel` before the constraint functor actually posits constraints, and the constraint functor should not posit any constraints if `CheckModel` returns `false`.

Here is an example of a use of the `CheckModel` procedure in the constraint functor implementing the barriers principle (`Solver/Principles/Lib/Barriers.oz`):

```
proc {Constraint Nodes G Principle}
   DVA2DIDA = Principle.dVA2DIDA
   ArgRecProc = Principle.argRecProc
   %%
   D1DIDA = {DVA2DIDA 'D1'}
```

```
      D2DIDA = {DVA2DIDA 'D2'}
   in
      %% check features
      if {Helpers.checkModel 'Barriers.oz' Nodes
          [D2DIDA#down
           D1DIDA#mothers
           D2DIDA#up
           D2DIDA#labels]} then
         D2DownMs = {Map Nodes
                     fun {$ Node} Node.D2DIDA.model.down end}
         %%
         BlocksMs = {Map Nodes
                     fun {$ Node}
                        BlocksM = {ArgRecProc 'Blocks' o('_':Node)}
                     in
                        BlocksM
                     end}
      in
         for Node in Nodes do
            %% get all nodes below my D1 mother on D2
            %% down_D2_mothers_D1(v) =
            %% union { down_D2(v') | v' in mothers_D1(v) }
            D2DownD1MothersM =
            {Select.union D2DownMs Node.D1DIDA.model.mothers}
            %% from this set, keep only those nodes which are above me
            %% these are then between my D1 mother and myself on D2
            %% between(v) = down_D2_mothers_D1(v) intersect up_D2(v)
            BetweenM =
            {FS.intersect D2DownD1MothersM Node.D2DIDA.model.up}
            %% get all edge labels which are blocked by the nodes in
            %% between(v)
            %% blocked(v) = union { blocks(v') | v' in between(v) }
            BlockedLM = {Select.union BlocksMs BetweenM}
         in
            %% my incoming edge labels set must be disjoint from the
            %% set of blocked labels.
            %% labels_D2(v) disjoint blocked(v)
            {FS.disjoint Node.D2DIDA.model.labels BlockedLM}
         end
      end
   end
```

This constraint functor accesses the model record fields `mothers` on dimension `D1`, and `down`, `up` and `labels` on dimension `D2`. It makes use of the `CheckModel` procedure to check this.

7.4.2.3 Integrate the constraint functor

To integrate the constraint functor into the XDK, you can again either trust the perl-script `scripts/addprinciple`, or by hand do the following:

- add the constraint functor to the `ozmake` makefile `Solver/Principles/Lib/makefile.oz`
- add the constraint functor to the top level `ozmake` makefile `makefile.oz` in order to include it in the `ozmake` package created for the XDK.

8 Oracles

Oracles can be used to guide the search for solutions of the XDK solver. They are stand-alone programs acting as a server to be asked by a client (= the XDK solver) to choose one among a number of possibilities for further search.

Oracles and the XDK solver communicate over a socket. The socket port to be used can be set upon start of the oracle, and must be the same as set for the XDK solver.

This is work in progress. At the moment, we have only one oracle viz. the *manual oracle* which allows the user to determine the path through the search tree.

8.1 Manual Oracle

The manual oracle co-operates with the IOzSeF search visualization tool. Upon solving an input sentence, the IOzSeF Tree Viewer pops up, but does not yet enumerate any solution. Instead, it shows only a grey triangle. With `Search->Next Solution` (or by simply pressing `n`), the next solution is explored. If there is a choice point, then the manual oracle displays a window showing the partially solved multigraph up until the choicepoint. By clicking the buttons `choice 1` and `choice 2` with the right mouse button, the manual oracle displays the edges which would be added if the left or the right alternative would be chosen. The choices can then be taken by clicking the corresponding buttons with the left mouse button.

The manual oracle can be started using the executable `ManualOracleServer.exe` in `Oracles/ManualOracle`. The `--port`-option makes the oracle use a port other than the default port 4711. E.g.:

```
ManualOracleServer.exe --port=42
```

makes the oracle use port 42. This can be also be shortened to:

```
ManualOracleServer.exe -p 42
```

9 Outputs

This chapter is about the *output functors* or just *outputs* of the XDK. The outputs visualize individual solutions.

Outputs can also be used to post-process XDG analyses using external programs written in another language than Mozart/Oz (e.g. perl-scripts). This is the purpose of the XML output functors, which output analyses in XML.

Each dimension specifies its own set of *chosen outputs* and *used outputs*. We call the dimension on which the output is used the *output dimension* Chosen outputs appear in the `Outputs` pull-down menu of the graphical user interface of the XDK. Used outputs are actually used to visualize the individual solutions.

An output definition includes:

- the *output identifier*
- the *output open method*
- the *output close method*

The output functors are taken from a predefined *output library*. Below, we present the definitions of all the output functors in the current output library. Note that we leave out output functors which are currently in development and might vanish again in later releases.

Most output functors *print* information. This printing can be redirected (e.g. to the Oz Inspector, the Oz Browser, stdout, or into a file).

We explain the syntax of the *Output Language (OL)* used in the Pretty output functor (Section 9.36 [OL syntax], page 161).

Developers only: We describe the *output record* in Section 9.37 [Output record], page 166. The output record is the representation of a solution prepared to be used by the output functors. In Section 9.38 [Writing new outputs], page 168, we explain how to write new output functors.

9.1 AllDags

This section explains the AllDags output functor.

- identifier: `output.allDags`
- open method: opens a window displaying the dags corresponding to all dimensions (including e.g. the `lex` dimension), not only the dimensions using the graph principle (see Section 9.15 [Dags], page 152). The individual dags are drawn as by the Dag output functor (Section 9.11 [Dag1], page 149)
- close method: closes all of these windows.

9.2 AllDags1

This section explains the AllDags1 output functor.

- identifier: `output.allDags1`
- open method: opens a window displaying the dags corresponding to all dimensions (including e.g. the `lex` dimension), not only the dimensions using the graph principle (see Section 9.16 [Dags1], page 152). The individual dags are drawn as by the Dag1 output functor (Section 9.12 [Dag11], page 150)

- close method: closes all of these windows.

9.3 AllDags2

This section explains the AllDags2 output functor.

- identifier: `output.allDags2`
- open method: opens a window displaying the dags corresponding to all dimensions (including e.g. the `lex` dimension), not only the dimensions using the graph principle (see Section 9.17 [Dags2], page 152). The individual dags are drawn as by the Dag1 output functor (Section 9.13 [Dag12], page 151)
- close method: closes all of these windows.

9.4 AllDags3

This section explains the AllDags3 output functor.

- identifier: `output.allDags3`
- open method: opens a window displaying the dags corresponding to all dimensions (including e.g. the `lex` dimension), not only the dimensions using the graph principle (see Section 9.18 [Dags3], page 153). The individual dags are drawn as by the Dag1 output functor (Section 9.14 [Dag13], page 151)
- close method: closes all of these windows.

9.5 AllLatexs

This section explains the AllLatexs output functor.

- identifier: `output.allLatexs`
- open method: prints the dags corresponding to all dimensions (including e.g. the `lex` dimension), not only the dimensions using the graph principle (see Section 9.24 [Latexs], page 157). The individual dags are printed as by the Latex output functor (Section 9.20 [Latex], page 153).
- close method: does nothing

9.6 AllLatexs1

This section explains the AllLatexs1 output functor.

- identifier: `output.allLatexs1`
- open method: prints the dags corresponding to all dimensions (including e.g. the `lex` dimension), not only the dimensions using the graph principle (see Section 9.25 [Latexs1], page 157). The individual dags are printed as by the Latex output functor (Section 9.21 [Latex1], page 156).
- close method: does nothing

9.7 AllLatexs2

This section explains the AllLatexs2 output functor.

- identifier: `output.allLatexs2`

- open method: prints the dags corresponding to all dimensions (including e.g. the `lex` dimension), not only the dimensions using the graph principle (see Section 9.26 [Latexs2], page 158). The individual dags are printed as by the Latex output functor (Section 9.22 [Latex2], page 156).
- close method: does nothing

9.8 AllLatexs3

This section explains the AllLatexs3 output functor.

- identifier: `output.allLatexs3`
- open method: prints the dags corresponding to all dimensions (including e.g. the `lex` dimension), not only the dimensions using the graph principle (see Section 9.27 [Latexs3], page 158). The individual dags are printed as by the Latex output functor (Section 9.23 [Latex3], page 157).
- close method: does nothing

9.9 CLLS

This section explains the CLLS output functor.

- identifier: `output.clls`
- open method: opens a daVinci window showing the CLLS constraint read off from dimensions pa and sc.
- close method: closes the daVinci window

9.10 CLLS

This section explains the CLLS1 output functor.

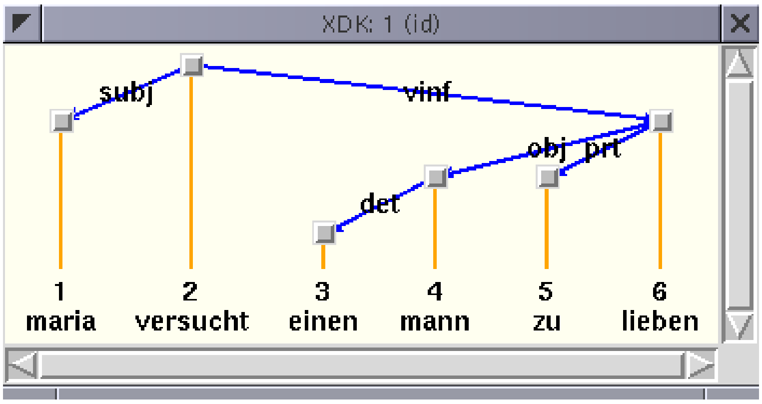
- identifier: `output.clls1`
- open method: prints out the CLLS constraint read off from dimensions pa and sc.
- close method: none

9.11 Dag

This section explains the Dag output functor. The Dag output functor assumes that the graph principle is used on the output dimension.

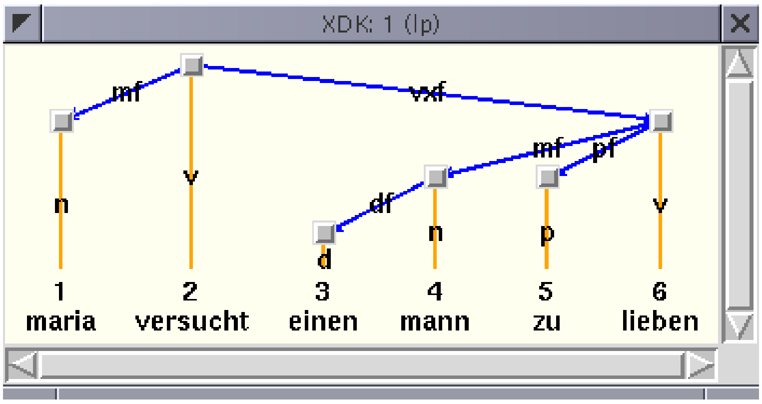
- identifier: `output.dag`
- open method: opens a window displaying all known edges (and all known edge labels and all known node labels) of the directed acyclic graph on the output dimension
- close method: closes all of these windows opened on the output dimension

Below, we display an example Dag output:



The blue edges correspond to edges in the dag, and the orange edges called *projection edges* connect each node in the dag with the corresponding index and word.

Below, we display an example Dag output including node labels. Node labels appear on the projection edges:



If you click on one of the gray nodes with the left mouse button, the XDK prints the abbreviated OL representation of the node on the output dimension. The middle mouse button prints the abbreviated OL representation of the node on all dimensions. The right mouse button prints the abbreviated OL representation of all nodes in the analysis, and on all dimensions.

9.12 Dag1

This section explains the Dag1 output functor. The Dag1 output functor assumes that the graph principle is used on the output dimension.

- identifier: `output.dag1`
- open method: opens a window displaying all known edges (and all known edge labels and all known node labels) of the directed acyclic graph on the output dimension, and all known dominance edges. Ghosts a node (in gray) if either a) it corresponds to word `.` or b) its set of incoming edge labels is non-empty and a subset of `{dummy, del}`. Ghosts edges and dominance edges labeled with a label in `{root, root1, root2, dummy, del}`. Compares the previous analysis with the current one: edges which are

new from the previous Dag1 output on the output dimension are red. New dominance edges are pink.

- close method: closes all of these windows opened on the output dimension and resets the comparison (i.e. the next output does not compare).

Notice that the Dag1 output functor is able to display labeled dominance edges only if the graph principle is used on the output dimension (showing labeled dominance edges requires the `downL` feature). Otherwise, it can only to display unlabeled dominance edges.

Dominance edges are light blue.

Compared with the Dag output functor (Section 9.11 [Dag1], page 149), Dag1 adds the following features:

- it includes dominance edges
- it ghosts nodes
- it ghosts edges
- it compares analyses

9.13 Dag2

This section explains the Dag2 output functor. The Dag2 output functor assumes that the graph principle is used on the output dimension.

- identifier: `output.dag2`
- open method: opens a window displaying all known edges (and all known edge labels and all known node labels) of the directed acyclic graph on the output dimension, and all known dominance edges. Compares the previous analysis with the current one.
- close method: closes all of these windows opened on the output dimension and resets the comparison (i.e. the next output does not compare).

Compared with the Dag output functor (Section 9.11 [Dag1], page 149), the Dag2 functor adds the following features:

- it includes dominance edges
- it compares analyses

Compared with the Dag1 output functor (Section 9.12 [Dag11], page 150), the Dag2 functor

- does not ghost nodes, and
- does not ghost edges

9.14 Dag3

This section explains the Dag3 output functor. The Dag3 output functor assumes that the graph principle is used on the output dimension.

- identifier: `output.dag3`
- open method: opens a window displaying all known edges (and all known edge labels and all known node labels) of the directed acyclic graph on the output dimension. Ghosts a node (in gray) if either a) it corresponds to word `.` or b) its set of incoming edge labels is non-empty and a subset of `{dummy, del}`. Ghosts edges and dominance edges

labeled with a label in `{root, root1, root2, dummy, del}`. Compares the previous analysis with the current one.

- close method: closes all of these windows opened on the output dimension and resets the comparison (i.e. the next output does not compare).

Compared with the Dag output functor (Section 9.11 [Dag1], page 149), the Dag3 functor adds the following features:

- it ghosts nodes
- it ghosts edges
- it compares analyses

Compared with the Dag1 and Dag2 output functors (Section 9.12 [Dag11], page 150, Section 9.13 [Dag12], page 151), the Dag3 functor

- does not include dominance edges

9.15 Dags

This section explains the Dags output functor.

- identifier: `output.dags`
- open method: opens a window displaying the dags corresponding to all dimensions using a graph principle (excluding e.g. the `lex` dimension). The individual dags are drawn as by the Dag output functor (Section 9.11 [Dag1], page 149)
- close method: closes all of these windows.

9.16 Dags1

This section explains the Dags1 output functor.

- identifier: `output.dags1`
- open method: opens a window displaying the dags corresponding to all dimensions using a graph principle (excluding e.g.\ the `lex` dimension). The individual dags are drawn as by the Dag1 output functor (Section 9.12 [Dag11], page 150).
- close method: closes all of these windows and resets the comparisons (i.e. the next output does not compare).

9.17 Dags2

This section explains the Dags2 output functor.

- identifier: `output.dags2`
- open method: opens a window displaying the dags corresponding to all dimensions using a graph principle (excluding e.g. the `lex` dimension). The individual dags are drawn as by the Dag2 output functor (Section 9.13 [Dag12], page 151).
- close method: closes all of these windows and resets the comparisons (i.e. the next output does not compare).

9.18 Dags3

This section explains the Dags3 output functor.

- identifier: `output.dags3`
- open method: opens a window displaying the dags corresponding to all dimensions using the graph principle (excluding e.g. the `lex` dimension). The individual dags are drawn as by the Dag3 output functor (Section 9.14 [Dag13], page 151).
- close method: closes all of these windows and resets the comparisons (i.e. the next output does not compare).

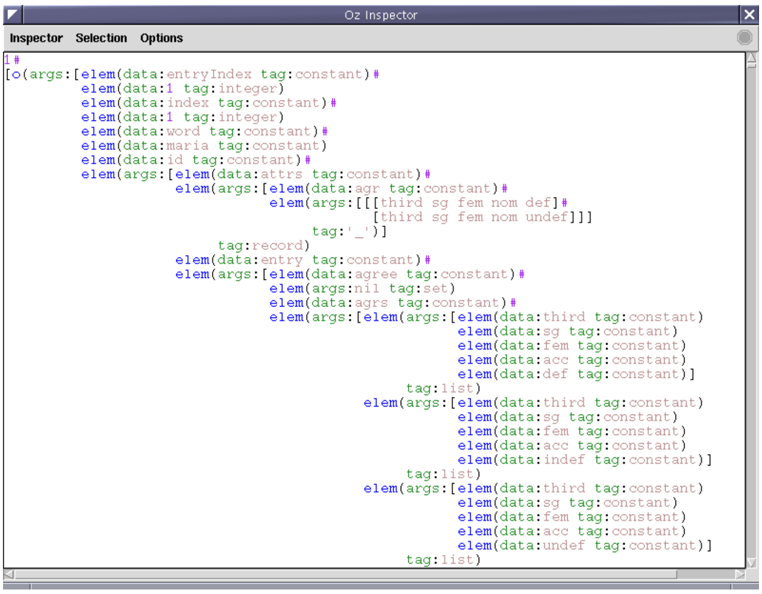
9.19 Decode

This section explains the Decode output functor.

- identifier: `output.decode`
- open method: prints the solution on the output dimension in the intermediate language (IL)
- close method: does nothing

This output functor prints the solution on the output dimension in a very detailed way, including the model record, using the intermediate language (IL). An equally detailed but more readable output can be obtained using the Pretty output functor (Section 9.28 [Pretty], page 158).

Below, we display an example Decode output printed in the Inspector:



9.20 Latex

This section explains the Latex output functor.

- identifier: `output.latex`

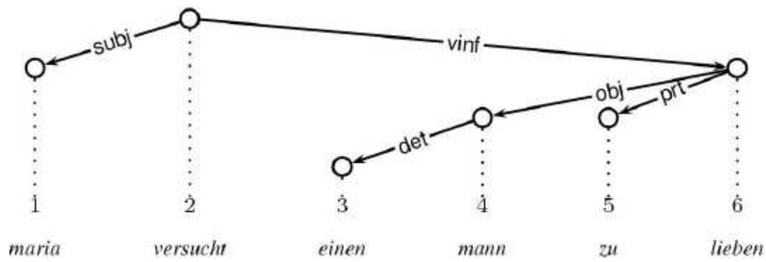
- open method: prints a LaTeX file containing all known edges (and known edge labels and node labels) of the directed acyclic graph on the output dimension
- close method: does nothing

The Latex output functor assumes that the graph principle is used on the output dimension.

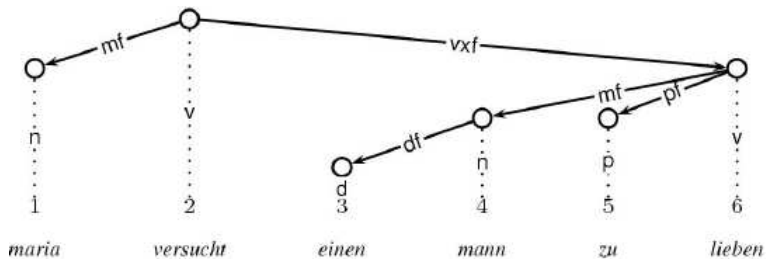
The resulting LaTeX output makes use of the style file `xdag.sty`, which is based on the original style file `dtree.sty` by Denys Duchier.

Unix users can use the shell script `xdag2eps` to convert the latex output into encapsulated postscript (EPS), `xdag2pdf` into PDF, or `xdag2jpg` to JPG. `xdag2eps`, `xdag2pdf` and `xdag2jpg` require the style file `xdag.sty` to be in the current directory. The latex file to convert is also required to be in the current directory.

Below, we display an example Latex output after having been compiled into pdf:



And below, we display an example Dag output including node labels (written on the vertical *projection edges*):



To get the LaTeX output into a file, just tick `file` in the `Extras` pull-down menu instead of the default `inspect`. This effects that all output normally printed using the Oz Inspector is redirected into a file. Whenever this happens, you are asked where to create this file. Having the file in your hands, you can then convert it into EPS, PDF or JPG using the scripts `xdag2eps`, `xdag2pdf` and `xdag2jpg`, respectively.

Here is the LaTeX code for the latter Dag:

```
\begin{xdag}
\node{1}{2}{$\begin{array}{c}1\\\\\textrm{maria}\end{array}$}{n}
\node{2}{1}{$\begin{array}{c}2\\\\\textrm{versucht}\end{array}$}{v}
\node{3}{4}{$\begin{array}{c}3\\\\\textrm{einen}\end{array}$}{d}
```

```
\node{4}{3}{$\begin{array}{c}4\\\\\textrm{mann}\end{array}$}{n}
\node{5}{3}{$\begin{array}{c}5\\\\\textrm{zu}\end{array}$}{p}
\node{6}{2}{$\begin{array}{c}6\\\\\textrm{lieben}\end{array}$}{v}
\edge{6}{5}{pf}
\edge{6}{4}{mf}
\edge{4}{3}{df}
\edge{2}{6}{vxf}
\edge{2}{1}{mf}
\end{xdag}
```

The Latex output functor paints dags using the `xdag` environment from the `xdag.sty` style file. `xdag` provides two basic commands: `\node` and `\edge`.

`\node` has four arguments:

1. a unique index of the node
2. the depth of the node (1 corresponds to the top of the dag, and `>1` to a lower position)
3. the word corresponding to the node (here: augmented with the index of the node using an array)
4. the node label of the node (usually empty for all dimensions other than lp)

`\edge` has three arguments:

1. a node index denoting the startpoint of the edge
2. a node index denoting the endpoint of the edge
3. the edge label

Notice that you can increase the horizontal distance between nodes using the `\xdagExtraColSep` command:

```
\begin{xdag}
\xdagExtraColsep{1}{20pt}
\node{1}{2}{$\begin{array}{c}1\\\\\textrm{maria}\end{array}$}{n}
\node{2}{1}{$\begin{array}{c}2\\\\\textrm{versucht}\end{array}$}{v}
\node{3}{4}{$\begin{array}{c}3\\\\\textrm{einen}\end{array}$}{d}
\node{4}{3}{$\begin{array}{c}4\\\\\textrm{mann}\end{array}$}{n}
\node{5}{3}{$\begin{array}{c}5\\\\\textrm{zu}\end{array}$}{p}
\node{6}{2}{$\begin{array}{c}6\\\\\textrm{lieben}\end{array}$}{v}
\edge{6}{5}{pf}
\edge{6}{4}{mf}
\edge{4}{3}{df}
\edge{2}{6}{vxf}
\edge{2}{1}{mf}
\end{xdag}
```

Here, the horizontal distance between the first and the second node is increased by `20pt`. You can also set this distance all nodes using `\xdagColsep`, as in the following example:

```
\begin{xdag}
\xdagColsep=20pt
\node{1}{2}{$\begin{array}{c}1\\\\\textrm{maria}\end{array}$}{n}
\node{2}{1}{$\begin{array}{c}2\\\\\textrm{versucht}\end{array}$}{v}
```

```
\node{3}{4}{$\begin{array}{c}3\\\\\textrm{einen}\end{array}$}{d}
\node{4}{3}{$\begin{array}{c}4\\\\\textrm{mann}\end{array}$}{n}
\node{5}{3}{$\begin{array}{c}5\\\\\textrm{zu}\end{array}$}{p}
\node{6}{2}{$\begin{array}{c}6\\\\\textrm{lieben}\end{array}$}{v}
\edge{6}{5}{pf}
\edge{6}{4}{mf}
\edge{4}{3}{df}
\edge{2}{6}{vxf}
\edge{2}{1}{mf}
\end{xdag}
```

9.21 Latex1

This section explains the Latex1 output functor. The Latex1 output functor assumes that the graph principle is used on the output dimension.

- identifier: `output.latex1`
- open method: prints a LaTeX file containing all known edges (and known edge labels and node labels) of the directed acyclic graph on the output dimension, and all known dominance edges. Ghosts a node if either a) it corresponds to word `.` or b) its set of incoming edge labels is non-empty and a subset of `{dummy, del}`. Ghosts edges and dominance edges labeled with a label in `{root, root1, root2, dummy, del}`.
- close method: does nothing

Notice that the Latex1 output functor is able to print labeled dominance edges only if the graph principle is used on the output dimension (accessing labeled dominance edges requires the `downL` feature). Otherwise, it can only to print unlabeled dominance edges.

Compared with the Latex output functor (Section 9.20 [Latex], page 153), Latex1 adds the following features:

- it includes dominance edges
- it ghosts nodes
- it ghosts edges

9.22 Latex2

This section explains the Latex2 output functor. The Latex2 output functor assumes that the graph principle is used on the output dimension.

- identifier: `output.latex2`
- open method: prints a LaTeX file containing all known edges (and all known edge labels and all known node labels) of the directed acyclic graph on the output dimension, and all known dominance edges.
- close method: does nothing

Compared with the Latex output functor (Section 9.20 [Latex], page 153), the Latex2 functor adds the following feature:

- it includes dominance edges

Compared with the Latex1 output functor (Section 9.21 [Latex1], page 156), the Latex2 functor

- does not ghost nodes, and
- does not ghost edges

9.23 Latex3

This section explains the Latex3 output functor. The Latex3 output functor assumes that the graph principle is used on the output dimension.

- identifier: `output.latex3`
- open method: prints a LaTeX file containing all known edges (and all known edge labels and all known node labels) of the directed acyclic graph on the output dimension. Ghosts a node if either a) it corresponds to word `.` or b) its set of incoming edge labels is non-empty and a subset of `{dummy, del}`. Ghosts edges and dominance edges labeled with a label in `{root, root1, root2, dummy, del}`.
- close method: does nothing

Compared with the Latex output functor (Section 9.20 [Latex], page 153), the Latex3 functor adds the following features:

- it ghosts nodes
- it ghosts edges

Compared with the Latex1 and Latex2 output functors (Section 9.21 [Latex1], page 156, Section 9.22 [Latex2], page 156), the Latex3 functor

- does not include dominance edges

9.24 Latexs

This section explains the Latexs output functor.

- identifier: `output.latexs`
- open method: prints the dags corresponding to all dimensions using a graph principle (excluding e.g. the `lex` dimension). The individual dags are printed as by the Latex output functor (Section 9.20 [Latex], page 153).
- close method: does nothing

9.25 Latexs1

This section explains the Latexs1 output functor.

- identifier: `output.latexs1`
- open method: prints the dags corresponding to all dimensions using a graph principle (excluding e.g. the `lex` dimension). The individual dags are printed as by the Latex1 output functor (Section 9.21 [Latex1], page 156).
- close method: does nothing

9.26 Latexs2

This section explains the Latexs2 output functor.

- identifier: `output.latexs2`
- open method: prints the dags corresponding to all dimensions using a graph principle (excluding e.g. the `lex` dimension). The individual dags are printed as by the Latex2 output functor (Section 9.22 [Latex2], page 156).
- close method: does nothing

9.27 Latexs3

This section explains the Latexs3 output functor.

- identifier: `output.latexs3`
- open method: prints the dags corresponding to all dimensions using a graph principle (excluding e.g. the `lex` dimension). The individual dags are printed as by the Latex3 output functor (Section 9.23 [Latex3], page 157).
- close method: does nothing

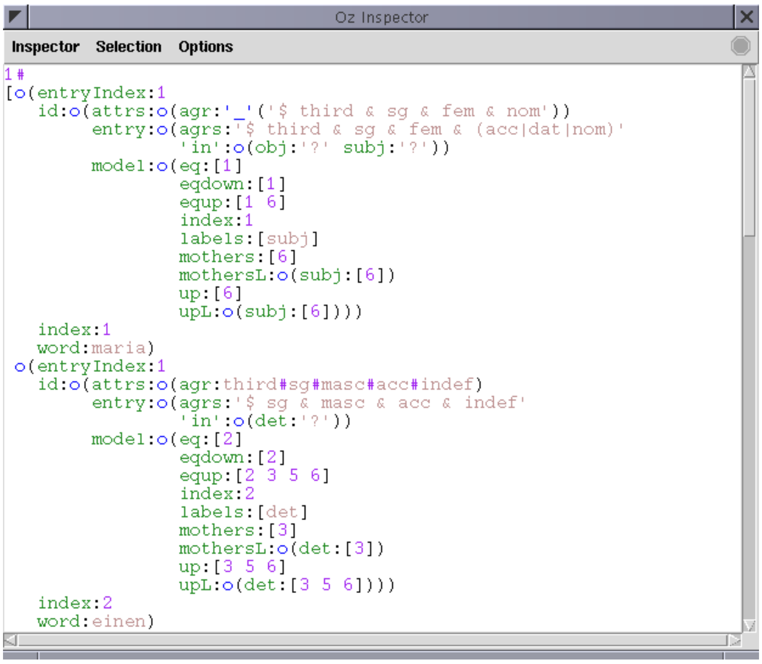
9.28 Pretty

This section explains the Pretty output functor.

- identifier: `output.pretty`
- open method: prints the solution on the output dimension in a detailed but "pretty" fashion (language: abbreviated output language (OL))
- close method: does nothing

This output functor prints the solution on the output dimension in a very detailed way, including the model record, using the abbreviated output language (OL).

Below, we display an example Pretty output printed in the Inspector:



9.29 Pretty1

This section explains the Pretty1 output functor.

- identifier: `output.pretty1`
- open method: prints the solution on the output dimension in a detailed but "pretty" fashion (language: abbreviated output language (OL)). Difference to Section 9.28 [Pretty], page 158: if a solution has already been printed, the next solution is compared with it and only the new/differing parts are printed. This is useful to find out "leaks" in the propagation.
- close method: "Resets" the functor, i.e., it behaves as if no solution has previously been printed.

9.30 XML

This section explains the XML output functor.

- identifier: `output.xml`
- open method: prints an XML file containing all known edges (and known edge labels and node labels) of the directed acyclic graph on the output dimension to stdout. The corresponding DTD can be found in `Extras/statistics.dtd` (starting from the tag `graph`). Does not print the lexical entries corresponding to the nodes (`output.xml1` does this).
- close method: does nothing

9.31 XML1

This section explains the XML1 output functor.

- identifier: `output.xml1`
- open method: prints an XML file containing all known edges (and known edge labels and node labels) of the directed acyclic graph on the output dimension to stdout.

The corresponding DTD can be found in `Extras/statistics.dtd` (starting from the tag `graph`). (difference to the ordinary XML output functor (Section 9.30 [XML], page 159): also prints the attribute records corresponding to the nodes.)

- close method: does nothing

9.32 XML2

This section explains the XML2 output functor.

- identifier: `output.xml2`
- open method: prints an XML file containing all known edges (and known edge labels and node labels) of the directed acyclic graph on the output dimension to stdout. The corresponding DTD can be found in `Extras/statistics.dtd` (starting from the tag `graph`). (difference to the ordinary XML output functor (Section 9.30 [XML], page 159): also prints the lexical entries corresponding to the nodes.)
- close method: does nothing

9.33 XML3

This section explains the XML3 output functor.

- identifier: `output.xml3`
- open method: prints an XML file containing all known edges (and known edge labels and node labels) of the directed acyclic graph on the output dimension to stdout. The corresponding DTD can be found in `Extras/statistics.dtd` (starting from the tag `graph`). (difference to the ordinary XML output functor (Section 9.30 [XML], page 159): also prints the model records corresponding to the nodes.)
- close method: does nothing

9.34 XML4

This section explains the XML4 output functor.

- identifier: `output.xml4`
- open method: prints an XML file containing all known edges (and known edge labels and node labels) of the directed acyclic graph on the output dimension to stdout. The corresponding DTD can be found in `Extras/statistics.dtd` (starting from the tag `graph`). (difference to the ordinary XML output functor (Section 9.30 [XML], page 159): also prints the attributes records, lexical entries and model records corresponding to the nodes.)
- close method: does nothing

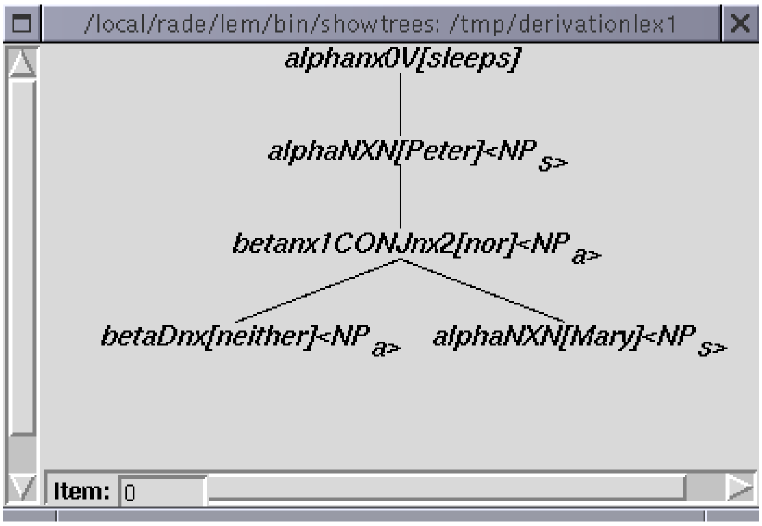
9.35 XTAGDerivation

This section explains the XTAGDerivation output functor.

- identifier: `output.xTAGDerivation`
- open method: runs the tree viewer from the XTAG lem parser package to show the derivation tree corresponding to the solution. Currently used only for grammars generated by the XTAG module of the XDK. Arbitrary many tree viewers can be

run simultaneously. For this function to work, the executables of the lem parser (`ftp://ftp.cis.upenn.edu/pub/xtag/lem/lem-0.14.0.tgz`) must be installed in the search path. The executable `showtrees` is called.

- close method: Closes all run tree viewers.



9.36 Output Language

In this section, we describe how to transform *Solver Language (SL)* expressions into *Output Language (OL)* expressions.

Notice that the XDK also provides output in abbreviated form: the abbreviated OL syntax is like the OL syntax, but with the following abbreviations to make the output a little less cluttered:

- top values are abbreviated with `top`
- bottom values are abbreviated with `bot`
- sets of tuples of which all projections are finite domains of constants are abbreviated using set generator expressions
- undetermined sets of tuples of which all projections are finite domains of constants are abbreviated using set generator expressions
- undetermined tuples of which all projections are finite domains of constants are abbreviated using set generator expressions

The abbreviated OL syntax is used by the Pretty and output functors to visualize all the information contained in a solution.

9.36.1 Feature path

Here is the syntax of a SL feature path:

```
featurepath(root: RootA
            dimension: DVA
            dimension_idref: IDA
```

```
                  aspect: AspectA
                  fields: FieldAs)
```

The corresponding OL expression is an Oz atom made up of the following parts:

```
'<root>.<dim var>(<dim id>).<aspect>.<field_1>.....<field_n>'
```

`<root>` corresponds to (`RootA`), `<dim var>` to `DVA`, `<dim id>` to `IDA`, `<aspect>` to `AspectA`, and `<field_i>` (`1<=i<=n`) to `FieldAs`.

9.36.2 Cardinality set

Here is the syntax of SL cardinality sets:

```
M
```

where `M` is an Oz finite set encoding the cardinality set `IL`.

And here is the corresponding OL expression:

```
OL
```

where `OL` is the OL expression encoding the cardinality set `M`.

9.36.3 Constraint

Undefined.

9.36.4 Domain

Here is the syntax of SL constants:

```
I
```

`I` is an Oz integer encoding the constant.

And here is the corresponding OL expression:

```
A
```

`A` is the constant encoded by `I`.

9.36.5 Integer

Here is the syntax of SL integers:

```
I
```

And here is the corresponding OL expression:

```
I
```

`I` stays the same.

9.36.6 List

Here is the syntax of SL lists:

```
SLs
```

`SLs` is an Oz list of SL expressions.

And here is the corresponding OL expression:

```
OLs
```

`OLs` is an Oz list of OL expressions encoding `SLs`.

9.36.7 Map

See Record.

9.36.8 Record

Here is the syntax of SL records:

```
o(A1:SL1
  ...
  An:SLn)
```

`Ai:SLi` (`1<=i<=n`) is a feature of an Oz atom `Ai` (the field) and a SL expression `SLi` (the value).

And here is the corresponding OL expression:

```
o(A1:OL1
  ...
  An:OLn)
```

`Ai` (`1<=i<=n`) stays the same, and `OLi` is the OL expression encoding `SLi`.

9.36.9 Set

The syntax of SL sets differs depending on the type of the domain of the set:

1. a finite domain of constants
2. a tuple of which all projections are finite domains of constants
3. any other type

Here is the syntax of SL sets for the different cases:

1.
   ```
   M
   ```
   `M` is an Oz finite set of integers encoding the constants in the set.
2. (see 1.)
3.
   ```
   SLs
   ```
   `SLs` is an Oz list of SL expressions in the set.

And here is the syntax of the corresponding OL expressions:

1.
   ```
   As
   ```
   `As` is an Oz list of Oz atoms encoding the set `M`.
2.
   ```
   Tups
   ```
   `Tups` is an Oz list of Oz tuples encoded in set `M`.[1]
3.
   ```
   OLs
   ```
   `OLs` is an Oz list of OL expressions encoding `SLs`.

[1] The abbreviated OL syntax is `A`, and `A` is a set generator expression encoding the set of tuples `M`.

9.36.10 String

Here is the syntax of SL strings:

```
A
```

`A` is an Oz atom encoding the string.

And here is the OL syntax:

```
A
```

`A` stays the same.

9.36.11 Tuple

The syntax of SL tuples differs depending on the type of the projections of the tuple:

1. all projections are finite domains of constants
2. at least one projection is not a finite domain of constants

Here is the syntax of SL tuples for the different cases:

1.
   ```
   I
   ```
   `I` is an Oz integer encoding the tuple.
2.
   ```
   [SL1 ... SLn]
   ```
   `SLi` is the SL expression on projection `i` (`1<=i<=n`) of the tuple.

And here is the syntax of the corresponding OL expressions:

1.
   ```
   OL1#...#OLn
   ```
   `OLi` is the OL expression encoding `SLi` (`1<=i<=n`).
2. (see 1.)

9.36.12 Undetermined values

The XDK solver can also yield *partial solutions* in which not all values in the node record are determined; instead some of the values are still variables. In the following, we show how these variables are represented in the OL.

9.36.12.1 Undetermined constants

This is the OL syntax for undetermined constants (i.e. constant variables):

```
'_'(DSpec)
```

`DSpec` is a *domain specification* representing the set of constants which can still be bound to the constant variable.

9.36.12.2 Undetermined integers

This is the OL syntax for undetermined integers (i.e. integer variables):

```
'_'(DSpec)
```

`DSpec` is a domain specification representing the set of integers which can still be bound to the integer variable.

9.36.12.3 Undetermined lists

This is the OL syntax for undetermined lists (i.e. list variables):

```
'_'
```

9.36.12.4 Undetermined sets

The OL syntax for undetermined sets (i.e. set variables) differs depending on the domain of the set:

1. a finite domain of constants or a tuple of which all projections are finite domains of constants
2. any other type

Here is the OL syntax of undetermined sets for the different cases:

1.
   ```
   '_'(MSpec1 MSpec2 DSpec)
   ```
   `MSpec1` is a set specification, representing the set of constants which are already known to be in the set variable.[2]

   `MSpec2` is a set specification representing the set of constants which could still end up in the set variable.[3]

   `DSpec` is a domain specification representing the set of integers which can still be bound to the integer variable representing the cardinality of the set variable.
2.
   ```
   '_'
   ```

9.36.12.5 Undetermined strings

This is the OL syntax for undetermined strings (i.e. string variables):

```
'_'
```

9.36.12.6 Undetermined tuples

The OL syntax for undetermined tuples (i.e. tuple variables) differs depending on the projections of the tuple:

1. all projections are finite domains of constants
2. at least one of the projections is no finite domain of constants

Here is the OL syntax of undetermined tuples for the different cases:

1.
   ```
   '_'(Tups)
   ```
   `Tups` is an Oz list of tuples representing the the set of tuples which can still be bound to the tuple variable.[4]

---

[2] The abbreviated OL syntax is `A`, and `A` is a set generator expression representing the set of constants which are already known to be in the set variable (if the set is over a tuple of which all projections are finite sets of constants).

[3] The abbreviated OL syntax is `A`, and `A` is a set generator expression representing the set of constants which are already known to be in the set variable (if the set is over a tuple of which all projections are finite sets of constants).

[4] The abbreviated OL syntax is `'_'(A)`, and `A` is a set generator expression representing the set of tuples which can still be bound to the tuple variable.

2.
```
'_'
```

9.36.12.7 Undetermined cardinality sets in valencies

This is the OL syntax for undetermined cardinality sets (i.e. cardinality set variables) in valencies:

```
'_'(OL1 OL2)
```

`OL1` is the cardinality set representing the set of integers which are already known to be in the cardinality set variable. `OL2` is the cardinality set representing the set of integers which can still be bound to the cardinality set variable.

9.37 Output record

The *output record* is the result of preparing the solution for the individual output functors. A solution is a list of node records. The output record is defined as follows:

```
o(usedDIDAs: DIDAs
  graphUsedDIDAs: DIDAs
  nodes: SLs
  nodeILs: ILs
  nodeOLs: OLs
  nodeOLAbbrs: OLAbbrs
  index2Pos: I2I
  printProc: PrintProc
  edges: EdgesRec)
```

The value of the `usedDIDAs` feature is a list of dimension identifiers (`DIDAs`) which are the used dimensions.

The value of the `usedGraphDIDAs` feature is a list of dimension identifiers (`DIDAs`) which are the used *graph dimensions*. A graph dimension is a dimension on which the either `principle.graph` or `principle.graph1` is used. This is useful to distinguish graph dimensions (which can e.g. be visualized using `output.dag`) from special dimensions like `lex` (purpose: assign a word form to a lexical entry) and `multi` (purpose: use multi-dimensional principles) which need not be visualized like this.

The value of the `nodes` feature is the Solver Language (SL) version of the solution: a list of node records.

The value of the `nodeILs` feature is the Intermediate Language (IL) version of the solution.

The value of the `nodeOLs` feature is the Output Language (OL) version of the solution.

The value of the `nodeOLAbbrs` feature is the abbreviated Output Language (OL) version of the solution (here: top values are abbreviated with `top` and bottom values with `bot`, and features denoting `top` are left out from records and valencies).

The value of the `index2Pos` feature[5] is a function from node indices (`I`) to the corresponding node positions (`I`).

[5] This feature only makes sense if you use the order principle.

The value of the `printProc` feature is a function from anything to nothing (`X -> U`), used for printing.

indices (`I`) to the corresponding node positions (`I`).

The value of the `edges` feature[6] is the *edges record* `EdgesRec`:

```
o(edges: DIDAEdgesRec
  ledges: DIDALEdgesRec
  lusedges: DIDALUSEdgesRec
  dedges: DIDADEdgesRec
  ldedges: DIDALDEdgesRec
  lusdedges: DIDALUSDEdgesRec)
```

The values of the features of the edges record are defined as follows:

- `DIDAEdgesRec`: Maps dimension identifier `DIDA` to the set of determined edges (`Edges`) on `DIDA`. An edge is an Oz record `edge(I1 I2)` representing an edge from the node with index `I1` to the node with index `I2`. The edge label of the edge need not be determined.
- `DIDALEdgesRec`: Maps dimension identifier `DIDA` to the set of determined labeled edges (`LEdges`) on `DIDA`. A labeled edge is an Oz record `edge(I1 I2 LA)` representing an edge from the node with index `I1` to the node with index `I2` labeled `LA`. The edge label of the edge must be determined.
- `DIDALUSEdgesRec`: Maps dimension identifier `DIDA` to the set of determined edges (`LUSEdges`) on `DIDA`. Here, the edge label of the edge must still be undetermined.
- `DIDADEdgesRec`: Maps dimension identifier `DIDA` to the set of *unlabeled dominance edges* (`DEdges`) on `DIDA`. An unlabeled dominance edge is an Oz record `dom(I1 I2)` representing a unlabeled dominance edge from the node with index `I1` to the node with index `I2`. An unlabeled dominance edge holds between nodes v and v' if:
  1. v' is in the set of nodes below v
  2. the set of mothers of v' is not yet determined
  3. v' is not in the set of nodes below any node below v
- `DIDALDEdgesRec`: Maps dimension identifier `DIDA` to the set of *labeled dominance edges* (`LDEdges`) on `DIDA`. A labeled dominance edge is an Oz record `dom(I1 I2 LA)` representing a labeled dominance edge from the node with index `I1` to the node with index `I2` labeled `LA`. A labeled dominance edge exists between `I1` and `I2` if:
  1. v' is in the set of nodes equal or below the daughters of v labeled with l
  2. the set of mothers of v' is not yet determined
  3. v' is not in the set of nodes below any node below v
- `DIDALUSDEdgesRec`: Maps dimension identifier `DIDA` to the set of *unlabeled dominance edges* (`LUSDEdges`) on `DIDA`. Here, the edge label of the dominance edge must still be undetermined.

[6] This feature only makes sense for dimensions using the graph principle (or the graph1 principle).

9.38 Writing new outputs

In this section, we explain how you can write new outputs in Mozart/Oz. You may also choose to do post-processing using another programming language, building on one of the XML output functors.

In order to write an output in Mozart/Oz, you need to provide two things:

1. the *output definition*
2. the *output functor*

9.38.1 Output definition

You write the output definition in the IL, and add it to the list of output definitions bound to the Oz variable `Outputs` at the top of the functor `Outputs/Outputs.oz`.

9.38.1.1 Example (dag output)

Here is an example output definition of the output `output.dag`:

```
elem(tag: outputdef
     id: elem(tag: constant
              data: 'output.dag')
     'functor': elem(tag: constant
                     data: 'Dag'))
```

The value of the `id` feature is an IL constant denoting the unique output identifier. The value of the `'functor'` feature is an IL constant denoting the filename of the Oz functor implementing the output (modulo the suffix `.ozf`).

9.38.1.2 Integrate the output definition

In order to integrate the output definition into the XDK, you need to add the identifier of the new output to the XML file `Outputs/outputs.xml`. Here, for each new output, you add a line like the following for the Dag output:

```
<outputDef id="output.Dag"/>
```

This step is necessary because XML language grammar files contain only output chooses and uses, but not output definitions. Therefore, the output identifiers of the chosen/used outputs are only referred to but not defined in XML language grammar files, which leads to errors running an XML validator on them.

9.38.2 Output functor

You write the output functor as an Oz functor exporting two procedures:

- `Open: DIDA I OutputRec -> U` (open the output for dimension `DIDA`)
- `Close: DIDA -> U` (close all windows opened by this output for dimension `DIDA`)

`Open` has three arguments:

1. `DIDA` is a dimension identifier denoting the output dimension
2. `I` is an Oz integer denoting the number of the solution
3. `OutputRec` is the output record providing the decoded information contained in the solution

`Close` has one argument:

1. `DIDA` is a dimension identifier denoting the output dimension

The output functor has to reside in `Outputs/Lib`. Its file name must match the value of the `'functor'` feature in the output definition, i.e. for `Dag`, it must be `Dag.oz`.

9.38.2.1 Integrate the output functor

In order to integrate the output definition into the XDK, you need to add the output functor to the `ozmake` makefile in `Outputs/Lib` (`Outputs/Lib/makefile.oz`).

9.38.2.2 Check cycles helper procedure

The functor `Outputs/Lib/Helpers.oz` exports the procedure `CheckCycles: DIDA NodeOLs -> U`. `CheckCycles` checks whether a graph has a cycle. It has two arguments:

- `DIDA`: a dimension identifier of the dimension which shall be checked
- `NodeOLs`: a list of node records in the OL, denoting a solution.

The procedure assumes that the graph principle (or the graph1 principle) is used on dimension `DIDA`.

10 PrincipleWriter

The principle compiler *PrincipleWriter* is thoroughly described in Jochen Setz' BSc. Thesis ([References], page 219). Also be sure to check out the example principles in `PrincipleWriter/Examples`, and the grammars suffixed `PW` (Chapter 5 [Grammars], page 89) which use them.

Basically, PW makes writing principles much easier. You can write down your principles in a simple first-order logic, and PW compiles them into efficient principle implementations for the XDK.

This is how it works, in a nutshell. An example is the climbing principle:

```
defprinciple "principle.climbingPW" {
  dims {D1 D2}
  constraints {

forall V: forall V1:
  dom(V V1 D1) => dom(V V1 D2)
  }
}
```

What you do is to define the name of the principle (`principle.climbingPW`), the dimensions over which it should abstract (`D1` and `D2`), and then a set of constraints.

Then you go into the directory `PrincipleWriter`, and call the compiler as follows:[1]

```
pw.exe -p Examples/climbingPW.ul
```

This compiles the principle, and puts the principle definition functor `ClimbingPW.oz` into `Solver/Principles` and the constraint functor also called `ClimbingPW.oz` into `Solver/Principles/Lib`.

To use the principle, it has to be integrated into the XDK. You can do this using the perl script `addprinciple`:

```
addprinciple ClimbingPW ClimbingPW
```

or by adding it manually as follows:

1. add the principle definition functor to the `ozmake` makefile `Solver/Principles/makefile.oz`
2. add the principle definition functor to the imported functors of the functor `Solver/Principles/Principles.oz`, and also to the list `Principles` on top of `Solver/Principles/Principles.oz`.
3. add the identifier of the new principle to the XML file `Solver/Principles/principles.xml`. Here, for each new principle, you add a line like the following for the graph principle:
4. add the constraint functor to the `ozmake` makefile `Solver/Principles/Lib/makefile.oz`
5. add the constraint functor to the top level `ozmake` makefile `makefile.oz` in order to include it in the `ozmake` package created for the XDK.

[1] Use `pw.exe --help` to get a full summary of the commandline options.

To finalize the integration of the new principle, call `ozmake` from the XDK main directory (where e.g. `xdk.exe` resides in).

...and off you go :)

11 Programs

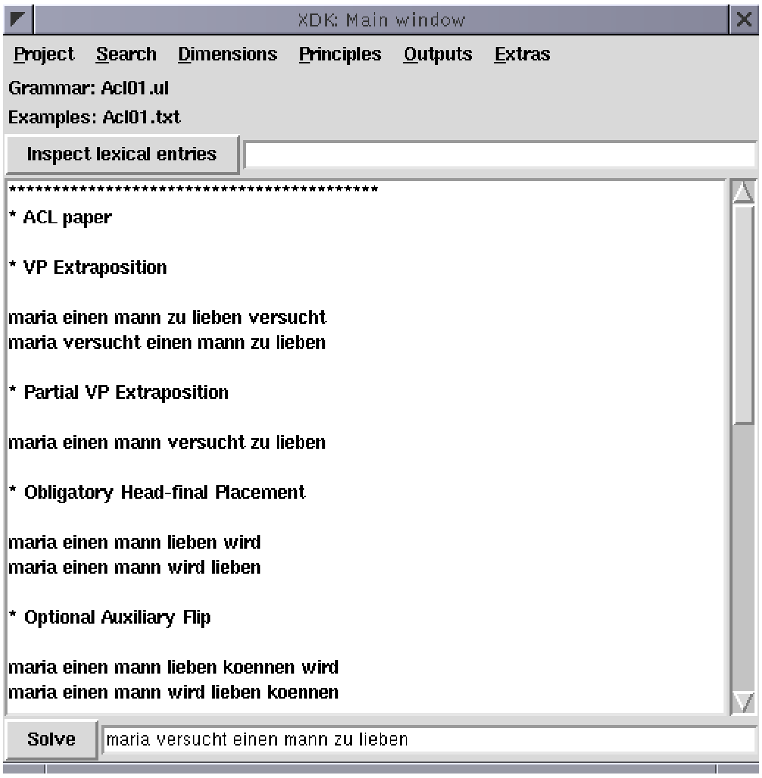
In this chapter, we describe the executable programs which expose the functionality of the XDK: the graphical user interface (`xdk`), the standalone grammar file compiler (`xdkc`), the standalone grammar file converter, and the standalone solver (`xdks`).

11.1 xdk

This section is about the *graphical user interface (GUI)* of the XDK.

11.1.1 Main window

We show a screenshot of the main window of the GUI below:



The main window consists of five parts, from top to bottom:

1. the menu bar comprising the pull-down menus
2. the status display comprising information about the currently loaded grammar and example files
3. the inspect lexical entries button and text field
4. the examples list view and scroll bar
5. the solve button and text field

11.1.1.1 Menu bar

The menu bar consists of the following pull-down menus, from left to right:

- Project
- Search
- Dimensions
- Principles
- Outputs
- Extras

The `Project` menu consists of the following menu entries:

- `About...`: Opens a dialog to display some information about the XDK.
- `Open grammar file...`: Opens a file dialog in which you can select a grammar file which is then compiled. The GUI also tries to find the corresponding example file with suffix `txt`.
- `Open grammar file/socket...`: Opens a string dialog in which you can select a grammar file or grammar socket (e.g. `4712.xmlsocket`). The GUI also tries to find the corresponding example file with suffix `txt`.
- `Open multiple grammar files...`: Opens a sequence of file dialogs in which you can select multiple grammar files which are then compiled and merged. Click on the `Cancel` button of the file dialog after the last grammar file to break the sequence. The GUI also tries to find the corresponding example file with suffix `txt` (for the first grammar file in the sequence).
- `Reload grammar files`: Recompiles the currently opened grammar file(s) and reloads the corresponding examples file.
- `Save compiled grammar file...`: Opens a file dialog in which you can select a path and filename and then saves the compiled grammar there.
- `Convert grammar file...`: Opens two file dialogs in which you can select the paths and filenames of the source and destination files for grammar conversion. The grammar file language into which the destination grammar file is converted depends on its suffix. The source grammar file can be one of the following:
  - UL file (suffix: `ul`)
  - XML file (suffix: `xml`)
  - IL pickle (suffix: `ilp`)
  - IL functor[1] (suffix: `ozf`)

  And the destination grammar file can be one of the following:
  - UL file (suffix: `ul`)
  - XML file (suffix: `xml`)
  - IL pickle (suffix: `ilp`)
- `Open examples file...`: Opens an examples file which is then displayed in the examples list view.

[1] The functor must export the IL grammar under the key `grammar`.

- `Reload examples file`: Reloads the currently opened examples file.
- `Close output windows`: Calls the close method of all currently used outputs
- `Quit`: Quits the GUI.

The `Search` menu consists of the following menu entries:

- `First solution/All solutions/Print CSP/Print FlatZinc`: When solving, either search for the first solution only, enumerate all solutions, print the CSP into a file, or print the CSP into a file using FlatZinc syntax.
- `Explorer/IOzSeF/Oracle`: Choose the search exploration tool, either the Oz Explorer, IOzSeF or Oracle (also using IOzSeF).
- `Set oracle port...`: Set oracle socket port for communication with the oracle server. Must be the same as set for the oracle server.

The `Dimensions`, `Principles` and `Outputs` menus are different depending on the currently opened grammar. Here, you can decide whether you wish to use or to switch off individual dimensions[2], principles and outputs.

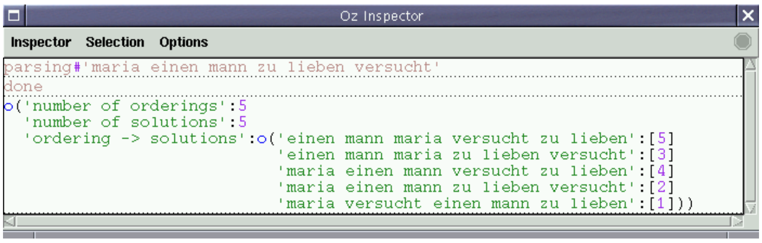
The `Extras` menu consists of the following menu entries:

- `Print`: Here, you can specify how the output functors *print* information:
  - `inspect` uses the Oz Inspector.
  - `browse` uses the Oz Browser (useful for copying text to the clipboard, which the Inspector cannot do).
  - `stdout` prints to stdout.
  - `file` prints into a file.
- `Compare lem solutions` Compares the derivation trees obtained from the generated XDG grammar (by the XTAG grammar generator) with the derivation trees obtained from the original XTAG lem parser. Counts the number of solutions and then prints out those solutions unique to the lem parser and the XDK. For this function to work, the executables of the lem parser (`ftp://ftp.cis.upenn.edu/pub/xtag/lem/lem-0.14.0.tgz`) must be installed in the search path. The executables `runparser` and `print_deriv` are called. There are a couple of variants:
  - `No filter` does not use any tree filtering.
  - `Simple filter` uses a reimplementation of `simple_filter.pl` from the lem parser from the lem parser distribution.
  - `Tagger filter` uses a reimplementation of `tagger_filter.pl` to do tree filtering. For this, the `mxpost` tagger by Adwait Ratnaparkhi must be installed in the directory denoted by the environment variable `MXPOST`. In addition, for `Tagger filter`, the `LEM` environment variable must point to the location of the lem parser.
- `Generate orderings...`: Uses the solver to search for the possible orderings of a sentence.
- `Solve examples`: Going downwards from the highlighted example, solves each sentence from the list of examples. Can come in handy for debugging grammars (answering the question whether all the examples still work).

[2] Toggling the lex dimension has no effect on the XDK solver - the lexicon is always used.

- `Solving Statistics`:
  - `Save solving statistics...`: Opens a file dialog in which you can select a path and a filename. Then obtains the solving statistics for the current grammar and examples. The statistics are in XML format (DTD: `Extras/statistics.dtd`). Unix users can use the shell script `diffnotime` to compare two solving statistics without taking the solving time into account (e.g. to spot the differences with respect to solutions, choices and failures).
  - `Set number of solutions...`: Opens a dialog in which you can set the maximum number of solutions for solving statistics. 0 means no solving (can be used e.g. to do profiling only)
  - `Set number of failures...`: Opens a dialog in which you can set the maximum number of failures for solving statistics.
  - `Set recomputation distance...`: Opens a dialog in which you can set the maximum recomputation distance for solving statistics.
  - `Open outputs`: If checked, the used outputs are opened for each solution in solving statistics.
- `Debug mode`: If checked, the XDK is in debug mode, giving out more information to ease (system, not grammar) debugging.

Below, we display an example output of the `Generate all orderings...` function:



`'number of orderings'` states the number of possible orderings. `'number of solutions'` states the number of solutions (this can be higher than the number of possible orderings for e.g. different analyses on a dimension other than lp). `'ordering -> solutions'` states a mapping from sentences to lists of indices of their corresponding solutions. By clicking on this list with the right mouse button, and then selecting `Actions` and then `Outputs`, you can invoke all used output functors for the solutions contained in the list.

The `Save solving statistics file...` function omits empty examples, examples starting with `/`, `*` or `%`.

11.1.1.2 Status display

The status display shows two things:

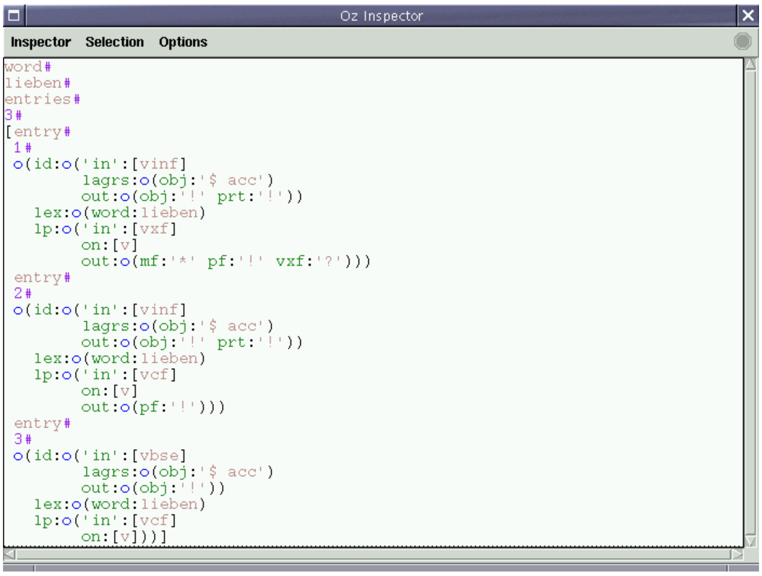
- `Grammar`: the name of the currently opened grammar file(s)
- `Examples`: the name of the currently opened examples file.

If the selected grammar file(s) could not be successfully compiled, the `Grammar` status includes the note `(not successfully compiled)`.

11.1.1.3 Inspect lexical entries button and text field

In the `inspect lexical entries` text field you can enter a list of words. After pressing the return key or after clicking on the `Inspect lexical entries` button the GUI opens the Oz Inspector to display all possible lexical entries for that word for the used dimensions. The lexical entries are displayed in the `Output Language (OL)`. If a lexical entry in the list equals another, e.g. the third is equal to the first, then this is signified as follows: `entry#3#'=1'`.

Below, we show an example output of this function:



Here, the first number (3) corresponds to the number of lexical entries for the word (`lieben`), and the numbers before each of these lexical entries is the index of that entry.

11.1.1.4 Examples list view and scroll bar

The examples list view displays the list of currently opened examples. Use the scroll bar on the right hand side of the list view to scroll through it. If you click on one of the examples, the GUI copies this example to the solve text field. If you double click on one of the examples, the GUI first copies the example to the solve text field and then solves the example.

11.1.1.5 Solve button and text field

In the `solve` text field you can enter a list of words. After pressing the return key or after clicking on the `solve` button the GUI opens the Oz Explorer to display all possible solutions for that list of words (under the currently opened grammar and selections in the `Dimensions` and `Principles` menus).

11.1.1.6 Tips and tricks

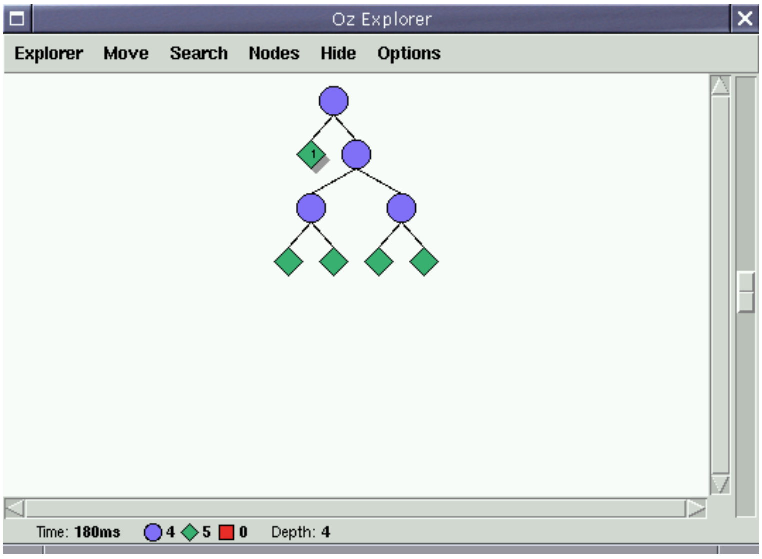
When looking for example files which correspond to grammar files, the GUI simply removes the suffix from the grammar file and adds the new suffix `txt`. E.g. for the grammar file `Grammars/Acl01.ul`, it looks for the examples file `Grammars/Acl01.txt`.

You can "tear-off" the pull-down menus such that they become independent windows (depends on the version of Tcl/Tk and your operating system, e.g. seems to work under Unix and Windows, but not MacOS X).

Some of the menu entries have keyboard shortcuts, displayed to their right. E.g. by pressing the keys `control` and `o` simultaneously, you can invoke the menu entry `Open grammar file...` of the `Project` menu.

11.1.2 Explorer window

We employ the Oz Explorer to display the search space traversed during the search for solutions. The Oz Explorer is described in more detail in `http://www.mozart-oz.org/documentation/explorer/index.html`. Essentially, blue circles denote choice points (XDG analyses which are not yet fully specified) in the search space, green diamonds solutions (fully specified XDG analyses) and red boxes failures. If the search space contains only red boxes, the solver could not find a solution for the input sentence. We show an example Explorer search tree below:



Non-failed nodes in the search tree (blue circles and green diamonds) can be double-clicked to invoke the used outputs.

Note that you can get a raw display of the underlying representation of the solver by selecting the menu `Nodes`, sub menu `Information` action, and then `Inspect`. This uses the Oz Inspector. If you choose `Show`, the display is printed to stdout. Choose `Outputs` to get back to the used outputs.

11.1.3 Commandline

You can supply the GUI with various parameters upon startup using commandline arguments. Each argument has a long version (starting with a double dash) and, of its positive occurrence, a short version (starting with one dash):

- `--help` or `--nohelp` (short version: `-h`): Display an overview of the commandline arguments. Default: `--nohelp`.
- `--grammars` (`-g`): Select the list of grammar files which shall be compiled (and then merged). Default: no files.
- `--examples` (`-e`): Select the examples file. Default: first grammar file with suffix `txt`
- `--input` (`-i`): Supply the GUI with a sentence which is then copied to the solve text field and solved. Default: `""`.
- `--search first`, `--search all`, `--search print` or `--search flatzinc` (`-s first`, `-s all`, `-s print` or `-s flatzinc`): Specify whether the solver shall search only for the first solution, enumerate all solutions or print the propagators to a file. Default: `--search all`.
- `--explorer explorer` or `--explorer iozsef` or `--explorer oracle` (`-x explorer` or `-x iozsef` or `-x oracle`): Use either the Oz Explorer, IOzSeF or the oracle for search visualization.
- `--port <Int>` (`-t`): Set oracle port to `<Int>`. Default: `4711`.
- `--print inspect`, `--print browse`, `--print stdout` or `--print file`: Specify the print procedure of the outputs. Default: `--print inspect`.
- `--solutions <Int>` (`-u`): Set maximum number of solutions to `<Int>` (for `Save solving statistics...`). Default: `1000`.
- `--failures <Int>` (`-u`): Set maximum number of failures to `<Int>` (for `Save solving statistics...`). Default: `1000`.
- `--reco <Int>` (`-c`): Set maximum recomputation distance to `<Int>` (for `Save solving statistics...`). Default: `5`.
- `--profile` or `--noprofile` (`-p`): Toggle profiling (for `Save solving statistics...`). Default: `noprofile`.
- `--outputs` or `--nooutputs` (`-o`): Open all used outputs (for `Save solving statistics...`). Default: `nooutputs`.
- `--debug` or `--nodebug` (`-d`): Switch on the debug mode. Default: `nodebug`.
- `--local` or `--nolocal` (`-l`): Specify where to look for constraints and outputs. Default: `--local` if `xdk.oz`, `xdk.ozf` and `xdk.exe` are in the current directory, else `--nolocal`.

11.2 xdkc

This section is about the standalone grammar file compiler of the XDK (`xdkc`).

11.2.1 Invocation

These are the commandline arguments of `xdkc`:

- `--help` or `--nohelp` (short version: `-h`): Display an overview of the commandline arguments. Default: `--nohelp`.

- `--grammars` (`-g`): Select the list of grammar files which shall be compiled (and then merged). Default: no files.
- `--input` (`-i`): Supply a sentence (for dynamic grammar generation). Default: `""`.
- `--output` (`-o`): Specify the filename for the compiled grammar. Default: the name of the first grammar file with its suffix changed to `slp`/`slp_db`.
- `--debug` or `--nodebug` (`-d`): Switch on the debug mode. Default: `nodebug`.
- `--local` or `--nolocal` (`-l`): Specify where to look for constraints and outputs. Default: `--local` if `xdk.oz`, `xdk.ozf` and `xdk.exe` are in the current directory, else `--nolocal`.

11.3 xdkconv

This section is about the standalone grammar file converter of the XDK (`xdkconv`). The grammar file converter takes a source grammar file and converts it into a destination grammar file. The grammar file language into which the destination grammar file is converted depends on its suffix.

The source grammar file can be one of the following:

- UL file (suffix: `ul`)
- XML file (suffix: `xml`)
- IL pickle (suffix: `ilp`)
- IL functor[3] (suffix: `ozf`)

And the destination grammar file can be one of the following:

- UL file (suffix: `ul`)
- XML file (suffix: `xml`)
- IL pickle (suffix: `ilp`)

11.3.1 Invocation

These are the commandline arguments of `xdkconv`:

- `--help` or `--nohelp` (short version: `-h`): Display an overview of the commandline arguments. Default: `--nohelp`.
- `--grammar` (`-g`): Select the source grammar file. Default: `""`.
- `--output` (`-o`): Select the destination grammar file. Default: `""`.
- `--debug` or `--nodebug` (`-d`): Switch on the debug mode. Default: `nodebug`.

11.3.2 Example

Here is an example. Assume you would like to convert the source grammar file `Grammars/Diplom.ul`, which is written in the *User Language (UL)* into the destination grammar file `Grammars/Diplom.xml` in the *XML language*. Here is how you do this with `xdkconv`:

```
xdkconv.exe -g Grammars/Diplom.ul -o Grammars/Diplom.xml
```

`xdkconv` reads the UL grammar `Grammars/Diplom.ul`, and saves the corresponding XML language grammar `Grammars/Diplom.xml`.

[3] The functor must export the IL grammar under the key `grammar`.

11.4 xdks

This section is about the standalone solver of the XDK (xdks). It outputs solving statistics in XML format (DTD: Extras/statistics.dtd). Unix users can use the shell script diffnotime to compare two solving statistics without taking the solving time into account (e.g. to spot the differences with respect to solutions, choices and failures).

11.4.1 Invocation

These are the commandline arguments of xdks:

- --help or --nohelp (short version: -h): Display an overview of the commandline arguments. Default: --nohelp.
- --grammars (-g): Select the list of grammar files which shall be compiled (and then merged). Default: no files.
- --examples (-e): Select the examples file. Default: "".
- --input (-i): Supply a sentence (which is then appended to the examples). Default: "".
- --search first, --search all, --search print or --search flatzinc (-s first, -s all, -s print or -s flatzinc): Specify whether the solver shall search only for the first solution, enumerate all solutions or print the propagators to a file. Default: --search all.
- --solutions <Int> (-u): Set maximum number of solutions to <Int>. Default: 1000.
- --failures <Int> (-u): Set maximum number of failures to <Int>. Default: 1000.
- --reco <Int> (-c): Set maximum recomputation distance to <Int>. Default: 5.
- --profile or --noprofile (-p): Toggle profiling Default: noprofile.
- --outputs or --nooutputs (-o): Open all used outputs. Default: nooutputs.
- --debug or --nodebug (-d): Switch on the debug mode. Default: nodebug.
- --local or --nolocal (-l): Specify where to look for constraints and outputs. Default: --local if xdk.oz, xdk.ozf and xdk.exe are in the current directory, else --nolocal.

12 Debug

In this chapter, we explain how grammars can be debugged. Due to the concurrent constraint-based implementation of the XDK solver, it cannot tell you spot-on what went wrong e.g. if you do not get the desired analysis. Rather, debugging XDG grammars proceeds in an indirect fashion, by individually turning off dimensions and principles.

12.1 Too few solutions

Debugging is easiest with the GUI (`xdk.exe`). Here, the menu `Dimensions` allows you to individually turn off the dimensions, and the menu `Principles` to individually turn off the principles of the grammar. Of course, this can also be done without the GUI by changing the grammar itself (turn off dimensions or principles by not using them).

Here is the basic recipe for debugging, e.g. in the case when a sentence which should yield a solution does not. For instance, assuming that your grammar has two dimensions, first try the two dimensions individually. If you do not get a failure then, but if you do get a failure when using both of the dimensions, then something about the interaction of the two dimensions must be wrong. That means typically either that:

- a word is lexically ambiguous, and one of the entries works only for one dimension, and the other only for the other dimension, or
- you are using a multi-dimensional principle which causes the failure

If you also get a failure when only using one dimension at a time, you can trace the failure by selectively turning off the principles on the respective dimension. E.g. you can switch off the in and out principles to see whether your valency specifications are causing the failure etc..[1]

12.2 Too many (structurally equivalent) solutions

If you get too many solutions, and each of the solutions is structurally equivalent, a common cause is the combination of two things: 1) that your grammar generates too many lexical entries, and 2) you use the principle `principle.entries` (Section 7.2.15 [Entries], page 109). This principle enumerates all different lexical entries, even if they make no structural difference.

The solution to this problem is twofold: 1) deactivate "principle.entries", 2) reduce the number of disjunctions. The reason for 2) is that even though your grammar does not show it and the solver can cope quite well with lexical ambiguity, it is always wise to keep the number of lexical entries as low as possible for efficiency. The key to doing that is to encode the disjunction into sets. For example,

```
...
dim idlp {end: {iobj: {ff | iosf | piosf} } } }
...
```

generates three lexical entries, whereas:

[1] If you switch off principles, the number of solutions can increase wildly. In the Oz Explorer, you can press `Ctrl-C` to immediately stop the search.

```
...
dim idlp {end: {iobj: {ff iosf piosf} } } }
...
```

only generates one and has the same effect: the indirect object (`iobj`) can either go into the `ff`, the `iosf`, or the `piosf`.

This encoding of disjunctions into sets also works for in valencies: you could encode:

```
...
dim id {obj? | iobj?}
...
```

into

```
...
dim id {obj? iobj?}
...
```

given that the models on the id dimension are always trees, which means each node can have at most one incoming edge, which in turn means that the incoming edge in the example can only be obj or iobj, but not both.

13 Directories

The directory structure of the XDK is as follows:

- CSPOutput: programs dealing with CSP output
  - Parsers: Parsers needed for the CSP output programs
- Compiler: Grammar file compiler
  - Lattices: Lattice functors
  - UL: User Language (UL) front end
  - XML: XML language (XML) front end
- Extras: Extra functors
- Grammars: Grammar files
- Oracles: Oracles
  - ManualOracle: Manual oracle
- Outputs: Outputs
  - Lib: Output functors
    - CLLS: Code for the CLLS output
      - DaVinci: daVinci support code (for the CLLS output)
    - Dag: Code for drawing dags using Tk
      - NewTkDAG: Rewrite of Denys Duchier's TkDAG functor
    - Latex: Code for outputting LaTeX code using `xdag.sty`
- PrincipleWriter: Principle compiler
  - Examples: Example principles
  - QuadOptimizer: Principle optimizer
- SXDG: Marco Kuhlmann's SXDG code
- Solver: The solver
  - Principles: Principle definitions
    - Lib: Principle functors
      - Select: Denys Duchier's selection constraint
      - FlatZinc: Support code for FlatZinc output
- XTAG: XTAG grammar generator
  - Grammar: XTAG grammar (not included in the XDK package, contained in `ftp://ftp.cis.upenn.edu/pub/xtag/lem/lem-0.14.0.tgz`)

14 Exceptions

This section describes the standard format for exceptions. We strongly encourage all developers to adhere to this format, for two main reasons:

- all executables pertaining to the XDK expect this format (e.g. the GUI expects it to correctly create the appropriate exception windows)
- consistency

A typical exception from the XDK looks is a record with label `error1`, and features `'functor'`, `'proc'`, `info`, `msg`, `coord` and `file`. The latter three features are obligatory, the others optional. The obligatory features are always printed out by the error handling procedures of the executables pertaining to the XDK. The other features are typically only to be seen when the debug mode is on.

Here is an example exception:

```
error1('functor':'Compiler/TypeCollector.ozf'
       'proc':'CollectTypes1'
       info:o(AttrsType 'dimension')
       msg:'Attributes type must be a record.'
       coord:AttrsCoord
       file:AttrsFile)
```

The value of `'functor'` must be an atom representing the path to the functor in which the exception was raised.

The value of `'proc'` must be an atom representing the name of the procedure in which the exception was raised.

The value of `info` is a tuple with label `o` with arbitrary projections providing extra information (in addition to the message, see below) for debugging purposes.

The value of `msg` is a virtual string representing the message to be printed by the error handling procedures.

The value of `coord` is either the atom `noCoord`, a pair `I1#I2` of two integers `I1` and `I2`, or just an integer `I`, denoting the range of lines or just the line of an error which occurred when compiling a grammar file.

The value of `file` is either the atom `noFile`, or an atom `A` denoting the filename of the grammar file where the error occurred.

15 Variable names

Variables names have the format `<description><type>` where `<description>` is an arbitrary description of the variable, and `<type>` its type. This is similar to the *Hungarian notation* used by Microsoft. Many people do not like it, but I think it is a good way to save time debugging type errors in Mozart (not being statically typed).

Types can be combined to yield new types. This is especially useful for records, tuples and functions:

- records: A record mapping elements from type X to elements of type Y has type XY.
- tuples: A tuple X#Y where the first projection is of type X and the second of type Y has type XYTup.
- functions: A function from values of type X to values of type Y has type X2Y.

In addition, we write XY for a "type disjunction", i.e., a variable which can either be of type X or of type Y.

We display the Oz type hierarchy taken from
`http://www.mozart-oz.org/documentation/base/node2.html#chapter.typestruct`
below:

```
X Y Z - Value
  FI - Number (Float or Int)
    I - Int
      D - FDInt
      Ch - Char
    F - Float

  Rec - Record (R)
    Tup - Tuple (T)
    L - Literal
      A - Atom
      N - Name
        B - Bool
        U - Unit

  Proc - Procedure (P)
  PO - unary Procedure or Object

  Ce - Cell

  C - Chunk
    Arr - Array
    Dict - Dictionary
    BitArr - BitArray
    K - Class
    O - Object
    Lock - Lock
    Port - Port
```

The types used in the XDK are based on the type hierarchy above. Here they are:

```
Attribute - XML attribute
A - atom
Arr - array
B - bool
ByteS - byte string
BitS - bit string
BitArr - bit array
Bot - (SL) lattice bottom value
C - chunk
Ce - cell
Ch - char
CIDA - class ID (atom)
CIDCIL - class ID (IL constant)
CIL - IL constant
CA - (A) principle constraint name
Co - lattice fd/fs variable count
Coord - coordinate
D - finite domain
Desc - expression description
Dict - dictionary
DIDA - dimension ID (atom)
DIDCIL - dimension ID (IL constant)
Dimension - (SL) a dimension
DVA - (A) dimension variable
DSpec - specification of a finite domain
Element - XML element
Entry - (SL) an entry
Entries - (SLs) a list of entries
E - exception
F - float
FI - number (float or integer)
File - file
Functor - functor
Handler - resolve handler
I - integer
IDA - any ID (atom)
IDCIL - any ID (IL constant)
IL - intermediate language (IL) expression
ILDist - IL expression after Distributor.oz
ILEnc - IL expression after Encoder.oz
ILCh - IL expression after TypeChecker.oz
ILTCo - IL expression after TypeCollector.oz
IIL - IL integer
K - class
Lock - lock
```

```
L - literal
Lat - Lattice abstract data type
LI - feature (literal or integer)
M - finite set
MSpec - specification of a finite set
N - name
O - object
OIDA - output ID
OL - output language (OL) expression
On - (N) output name
PIDA - principle ID (atom)
PIDCIL - principle ID (IL constant)
SL - solver language (SL) expression
SLC - SL after Compiler.oz
SLE - SL after Encoder.oz (stateless)
Pn - (N) principle name
Port - port
Principle - (SL) principle
Proc - procedure
Prof - profile
PO - unary procedure or object
Rec - record
S - string
Sem - sem feature value
Spc - space
Str - Stream
Sym - sym feature value
Term - expression
TIDA - type ID (atom)
TIDCIL - type ID (IL constant)
TkDEdges - edges
TkDOptions - options
TkDNodes - nodes
Tkvar - Tk-variable
Tn - (N) type name
Token - token
Top - (SL) lattice top value
Tup - tuple
U - unit
UL - user language expression
Url - URL record
UrlV - virtual string representing a URL
V - virtual string
W - Tk-widget
Win - Tk-toplevel widget (= window)
X - any type
Xs - list of X elements
```

```
Y - any type
Z - any type
```

16 Changes

Version 1.7.0 (December 28, 2007)

Updated manual with some information about how to use the Principle Writer.

Version 1.6.53 (December 27, 2007)

Updated manual with the latest grammars.

Version 1.6.52 (December 21, 2007)

More fixes, most importantly, fixed the CSP and FlatZinc output again. Until I have time for more documentation, for Flatzinc to work, you need to install Gecode, and then the FlatZinc-reader (it's a separate program), and then call it like this on the FlatZinc output file `test.fzn`

```
fz -mode stat -solutions 0 test.fzn
```

Here, `-solutions 0` means "find all solutions". Call `fz --help` for the options of `fz`.

This is it for now, my last day at university has ended. End of this year (after Xmas), I will make an official last uni-release (1.7.0 or whatever). After that, I hope Jorge and Denys will step in again to continue development, with me helping in my spare time (or in the time I have when traveling to/from my new workplace SAP).

With the PW even in this shape, the XDK should actually become very useful at least for teaching dependency grammars, but also for developing/prototyping new kinds of dependency grammars.

Also, it is a challenging topic more in the realms of constraint programming to continue working on the optimizer of the principle compiler. Jorge has already done a lot of work, which he should really bring in a form to publish, but there is ample space for more optimizations.

Version 1.6.51 (December 21, 2007)

Lots of small fixes everywhere :)

Version 1.6.50 (December 20, 2007)

First version with 1) lots of unnecessary stuff removed (support for deep guards, support for Gdbm, node/edge constraints, constraints type/lattice etc.) and 2) lots of new stuff, mostly done by Jorge: Principle Writer optimizations, memoization, lazy variables, dynamic profiling, type inference etc. To be documented soon.

Version 1.6.43 (September 20, 2007)

Removed support for let expressions again, kept only the old PW code not yet touched by Jorge. Jorge has implemented a new, much cooler version of PW which I'll integrate as soon as I can... stay tuned. It's really great stuff.

You can now use the usual valency notation using wild cards also for tuples (PW valency principle). That is, no need to write `[a "?"]` anymore, just write `a?` as before.

New grammar (`regdgPW.ul`) and principles to model regular dependency grammars (Kuhlmann/Moehl ACL 2007). Being able to model regular dependency grammars means that we can prove that XDG is at least as expressive as LCFRS!

Added fix to `GraphPWConstraints`, was incomplete if confronted with cyclic graphs.

Version 1.6.42 (July 23, 2007)

Added support for let expressions done by Jorge. Thanks again!

Added new node record feature `pos`, and removed the `pos` feature from all order principle model record definitions. That means that now all dimensions share the same positions for their nodes.

Great simplification: the solver can now be explicitly instructed to "parse", i.e., to equate the position with the index, or "generate", i.e., leave the `pos` feature untouched and distribute over it. The switch is in the `Search` pulldown menu (or the `--mode/-m` commandline switch).

This change bereaves the XDK somewhat of its flexibility, but makes it much simpler as well. And, most important, it fits better with the latest formalizations of XDG.

As a consequence of this simplification, I could remove the `Parse` and `SameOrder` principles.

Added hand-optimized versions of some of the PW principles suffixed `PW1.oz` in `Solver/Principles/Lib`. Now we need a compositional way of obtaining these optimizations...

Version 1.6.41 (July 20, 2007)

Added a transparent fix of `FS.reified.include` to `Share.oz`. `FS.reified.include` is broken in Mozart 1.3.2 (see Mozart users list), as Jorge has found out by deep private investigation. Thanks a lot! The fix however makes the XDK slower, I hope there will be a fix to Mozart someday.

First version of `dissPW.ul` works with only PW principles, same number of solutions (except for one sentence), and almost the same propagation, but much slower as the principles are not yet optimized.

Version 1.6.40 (July 19, 2007)

Continued writing PW principles for `dissPW.ul`. Now finished, almost works properly :-)

Version 1.6.39 (July 18, 2007)

Important bugfixes to `xdk.oz` (Inspect Lexical Entries) and `Compiler/Lattices/Set.oz` (`EncodeProc1` only worked properly for intersective sets).

New principles for new grammar `dissPW.ul` (dissertation grammar with PW principles).

Added set generator expressions and unequality to PW language, added set equality to `Solver/Principles/Lib/PW.oz`.

Version 1.6.38 (July 17, 2007)

Integrated IsIn optimization code into the `Solver/Principles/Lib/PW.oz`, thereby simplified the optimizer a lot. Same efficiency!

Version 1.6.37 (July 16, 2007)

Added new optimizations for negations/conjunctions/disjunctions/equivalences.

Added `PrincipleWriter` directory containg the work-in-progress principle compiler.

Version 1.6.36 (July 13, 2007)

Slight addition to the grammar record specification: now includes feature `as`, a list of all words in the lexicon of the grammar.

Added slight optimization to PW: now, all lattices used for encoding are created only once at the beginning of the constraint functor. As a result, all automatically generated functors with suffix `PW1` are now as fast (even a bit faster) as their quasi-automatically generated functors with suffix `PW`.

Removed all grammars and principles with suffix `PW1`, and replaced the principles with automatically generated ones using the latest PW.

Version 1.6.35 (July 12, 2007)

Modularized PW. Now we have an executable `pw.exe` in the `PrincipleWriter` subdirectory, which is the first live running version of PW! It still misses type inference and type checking, but it does already do some optimizations. To compile a principle and add it to the principle library, type:

```
pw.exe -p Examples/valencyPW.ul
```

To omit the optimizations, add `--nooptimize`.

Version 1.6.34 (July 11, 2007)

Added further optimizations to PW and indentation a la the emacs mode for Oz. Ready to be modularized tomorrow.

Fixed a bug in `pwtest1.oz` found by Jorge Marques Pelizzoni.

Version 1.6.33 (July 9, 2007)

Now the PW works for the first time and generates non-optimized principles. Used it to generate all `PW1`-suffixed principles, and, voila, seems to work with the example grammars.

Lots of stuff still to do:

- clean up, split into several functors
- error detection and sensible error messages (type checking)
- type inference
- optimizations (as seen in the manually implemented `PW`-suffixed principles)
- new Mozart code for adding new principles to the system (replacing the old perl-based scripts which do not have error checking)

Version 1.6.32 (July 5, 2007)

New PW grammars `nut1PW.ul` and `nut1PW1.ul`, the latter again unoptimized, imaginary PW output.

New PW principles `principle.agrPW`, `principle.agreementPW`, `principle.dagPW`, `principle.linkingEndPW` and `principle.linkingMotherPW`, and unoptimized versions with suffix `1`.

Version 1.6.31 (July 3, 2007)

Changed lexicalized order principle such that `^` is an edge label again, but there cannot be any edges labeled `^`. Also changed all grammars using the principle accordingly.

Removed type union/disjunction (not needed anymore).

Worked a lot on the new PW principles to make them as efficient as the original ones.

Duplicated the PW principles: now, there is always one version which is optimized as far as possible (e.g. `principle.orderPW`), and one which is not optimized, marked by a suffixed `1` (e.g. `principle.orderPW1`). Accordingly, the grammars `ANBNPW.ul` and `EQABPW.ul` use the optimized, and `ANBNPW1.ul` and `EQABPW1.ul` the unoptimized principles.

Version 1.6.30 (June 8, 2007)

Improvements to and optimizations of the new PW principles.

Version 1.6.31 (June 5, 2007)

New principles `principle.orderPW` and `principle.projectivityPW`, and new grammar `ANBNPW.ul`, again to support the ongoing "PrincipleWriter" work.

Version 1.6.29 (June 4, 2007)

New principles `principle.graphPW`, `principle.treePW` and `principle.valencyPW` and new grammar `EQABPW.ul` to support the ongoing work on the principle compiler "PrincipleWriter". The principles are basically what should be obtained from the compiler when it works.

Version 1.6.28 (May 4, 2007)

Solving statistics now supports a timeout, i.e., solving is interrupted after a fixed amount of time. The default is 3600 (1 hour). Notice that the timeout is checked only at choice points/solutions/failures in the search tree, i.e., it may take longer than the timeout to reach the first fixpoint and thus it may take longer to solve in overall despite the timeout.

Changed defaults for solving statistics in `xdk.exe` and `xdks.exe` again: the maximum number of solutions is 9999 again, as is the maximum number of failures.

Version 1.6.27 (April 27, 2007)

New tool for extracting example sentence files from Penn Treebank `*.pos` files (tagged). The tool is called `ptbextraxt.exe` and lets you also specify the minimum and maximum length of the sentences to be extracted. The vocabulary of the sentences is adapted to the XTAG grammar found in the current lem distribution (0.14.0).

Made fixed recomputation with recomputation distance 5 the default for `xdk.exe`, for Explorer, IOzSeF and Oracle search. This has the effect that exploring even large search spaces will not immediately lead to a rather unresponsible system.

Added new option for solving statistics (`xdk.exe` and `xdks.exe`) to specify the maximum number of failures. Using this and the possibility to specify the maximum number of solutions helps to terminate the search for solutions before it goes on forever.

Consistently made executables lower case (except for the included SXDG directory). Hence, `ManualOracleServer.exe` is now called `manualoracleserver.exe`, and `XTAGServer.exe` `xtagserver.exe`.

Many other small improvements.

Version 1.6.26 (April 23, 2007)

Now CSPs can be printed into files whose filename can be conveniently chosen. No more printing out to stdout.

Added experimental functionality to print out CSPs using FlatZinc syntax. This is a new standard syntax for constraint solvers, developed by the G12 project at several universities in Australia (see `http://www.g12.cs.mu.oz.au/Summary.html`). In particular, the Gecode constraint library will soon support FlatZinc officially, which means that this gets the XDK one step closer to integration with Gecode.

Printing out CSPs now also works with the standalone solver `xdks.exe`.

New file suffixes: `csp` for printed out CSPs, `fzn` for printed out CSPs using FlatZinc syntax, and `sol` for solutions obtained from the program `explore.exe` in `CSPOutput`.

Many small bugfixes and enhancements.

Version 1.6.25 (April 11, 2007)

Slight fix to XTAG grammar generation to adapt to the new output functors (which display the words of `word` features of all dimensions, not just `lex`): the `word` feature on the `lp` dimension is now called `word1`.

Version 1.6.24 (April 10, 2007)

Made it much easier for users from the outside to use the oracle functionality of the XDK:

- Added the two IOzSeF packages `tack-iozsef-1.1.pkg` and `tack-TkTreeWidget-0.7.pkg` by Guido Tack to the XDK distribution (in case MOGUL is not available or so).
- Integrated the SXDG package by Marco Kuhlmann.

Now, the example oracle can be used already after simply installing the two IOzSeF packages.

Included a license at last - CeCILL v2 (GNU-GPL-compatible).

Cleaned up a bit:

- Removed script `doublefdfs`.
- Removed example files `pri_EQAB.oz` and `pri_EQAB_lex.oz` from the XDK distribution.

Version 1.6.23 (February 23, 2007)

Removed the code supporting the second node-word mapping (`lex.entry.word1`) since it was too hacky IMHO.

Now, following a discussion with Vineet Chaitanya, the output functors (Tcl/Tk and Latex), and the `GenerateOrderings` code support the possibility of additional node-word mappings on any other dimension. An example is given in the grammar `Grammars/MWE.ul`, where the syntactic dimensions (`id` and `lp`) have a word feature different from that on the semantic dimension (`pa`). Now if the grammar shall be used for parsing, the input word feature (`lex.entry.word`) is unified with the syntactic word and the parsing principle is switched on. If the grammar shall be used for generation, the input word feature is unified with the semantic word, and the parse principle switched off. This is implemented in the example grammar `Grammars/MWEgen.ul`.

Added three work-in-progress example grammars: `Grammars/MTS10.ul`, a grammar where two context-free grammars are combined to model scrambling, and `Grammars/FG_TAGDC.ul` and `Grammars/FG_TAGDCgen.ul`, where a TAG grammar and a grammar for dominance constraints (modeled as in the dissertation) are combined.

Version 1.6.22 (February 19, 2007)

Wrote a new principle Section 7.2.71 [XTAGLinking], page 133 which combines the now removed `principle.linkingEnd1` and Section 7.2.63 [SameEdges], page 129.

Added code to `Solver/Principles/Lib/Helpers.oz` that generates the records encoding the dominance/precedence relations on Gorn addresses on-the-fly for the Section 7.2.70 [XTAG1], page 132 and Section 7.2.72 [XTAGRedundant], page 134 principles.

These two steps greatly simplify the writing of TAG grammars, as can be observed e.g. in the grammar Section 5.51 [ww], page 95.

Further improved the GenerateOrderings dialog.

The GenerateOrderings function and all DAG output functors now recognize whether a grammar as the feature `lex.entry.word1`, which is a feature additional to `lex.entry.word`, modeling a second "node-word mapping". This is convenient for generation, as e.g. in the grammar Section 5.30 [MWEgen], page 92, where the `lex.entry.word1` feature denotes the generated word for the node.

Version 1.6.21 (February 18, 2007)

Improved the `GenerateOrderings` functionality of the GUI. Now, several dimensions can be selected where the parse principle shall be switched off. The selection is now much more comfortable using checkbuttons instead of a text field.

Version 1.6.20 (February 17, 2007)

Updated the Section 7.2.8 [CSD1], page 106 principle to more closely reflect the diss, and to be more configurable.

Removed the `principle.projectivity1` principle and put its functionality into Section 7.2.61 [Projectivity], page 129. Now, if a principle which introduces a `yield` feature is used, then the `yield` set must be convex, otherwise, the principle constrains the `eqdown` set to be convex.

Removed the `Projective` argument variable from the Section 7.2.48 [Order], page 124 and Section 7.2.49 [Order1], page 125 principles (and their variant Section 7.2.55 [OrderConstraints], page 126 and Section 7.2.50 [Order1Constraints], page 125). Please note: if you made use of this argument variable to make your analyses projective, please change your grammars to use the additional Section 7.2.61 [Projectivity], page 129 principle instead! I already changed this for the example grammars of course.

Version 1.6.19 (February 16, 2007)

Optimized XTAG principles.

Added `XTAGDist` constraint functor for distributing the positions when the solver is used as a realizer for TAG grammars.

Updated the main page of the manual, highlighting the compiler and solver chapters.

Fixed severe bug in `principle.linkingAboveStart`.

Fixed bug in the UL grammar file parser — could not handle missing brackets at the end of the file.

Version 1.6.18 (February 14, 2007)

Made the XTAG principles reversible.

Version 1.6.17 (February 13, 2007)

Finished updating the functor profiles for all constraint functors. Now the profiler gives precisely the right number of propagators and variables, which can be tested by using the `solve.exe` executable from the `CSPOutput` directory.

Also added a new chapter to the documentation which explains the profiling functionality.

Version 1.6.16 (February 9, 2007)

Removed memory consumption measurements from solving statistics again (too imprecise).

Removed copies of the NewTkDAG and Select directories from CSPOutput. That is, CSPOutput is not stand-alone anymore, but must be distributed with the entire XDK. Which isn't so much of a problem, compared with the overhead of having to deal with copies of especially the selection constraint.

Changed all hard-wired references to the /tmp/ directory. Would not work on Windows, and this was the reason why the XTAG code would not work on Windows. Now it does — temporary files are now either written to /tmp/, or, if that is not available, to /C/Windows/Temp, or to /C/Temp.

Added new Windows binaries of the slighly optimized Select.fs and Select.union constraints.

Updated constraint functor profiles using the propagator output. Not all done yet though.

Version 1.6.15 (February 7, 2007)

Added slight optimizations to Select.fs and Select.union. Not included in the Windows binary yet.

Version 1.6.14 (February 2, 2007)

Improved LOP grammar (Grammars/LOP.ul) and principle (Solver/Principles/Lib/LOP.oz).

Version 1.6.13 (February 1, 2007)

Plugged in a reimplementation of the tagger_filter.pl tree filter from the lem parser distribution. Uses the mxpost POS tagger by Adwait Ratnaparkhi, which does not seem to be publicly available anymore though. Works perfectly, same number of solutions as the original lem parser. To use it, given that mxpost is properly installed in the directory pointed to by the MXPOST environment variable, call the XTAGServer as shown below:

```
XTAGServer.exe -f tagger
```

Plugged in the supertagger available on the XTAG webpage. Can be selected as a tree filter as shown below:

```
XTAGServer.exe -f supertagger
```

Where the environment variable COREF must point to the currently used data directory within the supertagger directory (as stated in the README there), e.g. to the 200K.data directory.

Added a function in xdk.exe to compare the solutions from the lem parser, using the tagger filter, and the XDK, using the reimplementation of the filter.

Fixed bug in LemComparer.oz (would not properly handle the distinguishing suffixes introduced in the previous version of the XDK).

Fixed XTAGRoot constraint.

Cleaned up Extras menu of the GUI.

Version 1.6.12 (January 30, 2007)

Fixed small small bugs in xdk.exe.

Improved filter reimplementation. Now gets precisely the same solutions as the lem parser when using pruning and filtering. Plugging in a tagger/supertagger comes next.

Prepared plugging in of a tagger/supertagger to the XTAG grammar generator. Most notably, introduced a way to distinguish multiple occurrences of the same word by adding suffixes separated from the word by a hash, e.g. "a man who sleeps sleeps#1". Not thoroughly tested though.

Version 1.6.11 (January 25, 2007)

Added implementation of syn lookup pruning from the lem parser, obtainable via the `--prune` option of the the XTAGServer. Put on by default.

Version 1.6.10 (January 23, 2007)

Added reimplementation of the `simple_filter.pl` tree filter from the lem parser distribution to the XTAG grammar server code. Can be invoked by launching the XTAGServer using the `-f simple` option:

```
XTAGServer.exe -f simple
```

Adapted the GUI to include two different versions of `Compare lem solutions...` (`Extras` menu): one without filtering (useful if you called the XTAGServer with `-f none`), and one with filtering (if you called it with `-f simple`).

Added file `4712.ulsocket` for convenience (for easier access to the XTAG grammar generator, given that it is started on port 4712).

Added debug output to solving statistics, showing the number of failures and solutions (succeeded).

Version 1.6.9 (January 19, 2007)

Renamed example file `pri.oz` to `pri_EQAB.oz`.

Added example file `pri_EQAB_lex.oz`, the valency-lexicalized version of `pri_EQAB.oz`.

Version 1.6.8 (January 18, 2007)

Added missing nodeset files for `diss` and `LOP` grammars.

Changed Denys' location to Orleans at last :)

Version 1.6.7 (January 16, 2007)

stderr printouts of `xdkc.exe` and `xdks.exe` now use `System.showError` instead of `System.printError` for better readability.

Changed memory consumption measurement to use the `gc.active` property.

Added scripts `optiOff` and `optiOn` to toggle the native `IsIn` optimization.

Version 1.6.6 (January 15, 2007)

Small improvement to `principle.lop`.

Changed some defaults for solving statistics in `xdk.exe` and `xdks.exe` profiling is now turned off per default, the maximum number of solutions 1000 (instead of 9999), and the recomputation distance 5 (1).

Added memory consumption to solving statistics, based on the `'memory.heap'` property of Mozart/Oz. Also adapted the DTD `Extras/statistics.dtd`.

Version 1.6.5 (January 12, 2007)

CSPOutput: renamed `solve.exe` to `explore.exe` and added a new program `solve.exe` which does not call the explorer but uses a handmade search engine to obtain solving statistics.

Improved `principle.lop` for the new LOP grammar (`Grammars/LOP.ul`).

Version 1.6.4 (January 10, 2007)

LOP grammar and principle added.

Version 1.6.3 (January 8, 2007)

Added information about the principles and constraint functors belonging to the propagators in the propagators output.

Added `Helpers.isIn` optimization to `Order2Conditions`, big speed-up for grammars using the lexicalized order principle (e.g. `Grammars/diss.ul`).

Version 1.6.2 (January 5, 2007)

Removed constraints that partitioned the sets of daughters and mothers sorted by label (`daughtersL` and `mothersL`) from the `GraphMakeNodes` and `GraphMakeNodes1` constraint functors. This wrongly disallowed that nodes had multiple outgoing edges to the same daughter node (of course with different edge labels).

Added these partitioning constraints to the `TreeConditions` constraint functor, and, optionally (`DisjointDaughters` argument variable) to the `Dag` constraint functor.

Changed default for the `Connected` argument of the `Dag` constraint functor to `false`.

Adapted all grammars affected by these changes (i.e., those using the Dag principle).

Fixed bug in `LinkingAboveBelow1or2StartDG` constraint functor.

Added constraint to `GraphConditions` and `GraphConditions1` constraint functors for better propagation on cyclic graphs (the `down` and `eqdown` sets were not properly constrained whenever a node only had itself as its daughter).

Added example file `pri.oz` which is a first example of principle compilation.

Version 1.6.1 (January 2, 2007)

Added `AddHandlers` procedure for the executable in the `CSPOutput` directory - they were not able to find the selection constraint when compiled from the toplevel XDK directory.

Bumped copyrights to 2007.

Version 1.6.0 (December 22, 2006)

Much much much faster CSP output thanks to an idea by Stefan Thater.

Introduced new directory `CSPOutput` for functors and programs working with CSP output. Includes the programs `solve.exe` for solving CSP outputs, and `view.exe` to view the solutions of CSP outputs (saved from the Explorer in `solve.exe`).

The directory is stand-alone, e.g. it can be tar-gzipped and then be used on a machine without the XDK installed. This is another landmark in our transition to Gecode.

Merry Christmas BTW!

Version 1.5.8 (December 18, 2006)

Fixed bug just introduced into the IsIn optimization — it would suspend when printing out propagators.

Changed `Solver/Principles/Lib/Share.oz` such that it now filters out non-fd/fs variables (e.g. `lex.entry.word`).

Version 1.5.7 (December 14, 2006)

Found out why the parser had become slower - it was the IsIn optimization in `Solver/Principles/Lib/Helpers.oz` which I had changed to the worse. Fixed.

Removed native treeness constraint from `principle.tree` (functor `TreeConditions.oz`. Added a new principle `principle.tree1` (with new functor `TreeConditions1.oz`) which still includes the native treeness constraint in case a grammar still needs it.

Version 1.5.6 (December 13, 2006)

Removed remaining deep guards from `principle.coindex` and `principle.pl`.

Made printing propagators more efficient using a trick involving the counter for finite domain variables available in the Properties module of Mozart/Oz.

New script `doublefdfs` to check whether print propagators output is buggy, introducing the same variable numbers for both finite set and finite domain variables.

New Oz test code `testpropagatoroutput.oz` to test the print propagators output against Mozart/Oz itself.

Continued updating the manual, grammars and principles to reflect the state-of-the-art of the system, many many small fixes and improvements.

Version 1.5.5 (December 5, 2006)

Added logo to GUI (Project -> About).

Updated the grammars to only use principles not using deep guards, except `DiplomDG.ul` and `dissDG.ul`. This also meant that the grammars using edge constraints had to be reformulated without. As a result, no grammar uses edge constraints any more. Since I could also remove all node constraints from the grammars, further versions of the XDK could drop the expression of node and edge constraints altogether, since their benefit is not high enough (I think) to warrant the complications they introduce to the system.

Removed grammar `seg.ul`.

Added diss to list of references in this manual.

Version 1.5.4 (December 1, 2006)

Suffixed all constraint functors, principle definition functors and grammars using edge constraints with DG for "deep guards", and removed the NDG suffix from those not using deep guards. As a result, if the user e.g. uses the `principle.graph` principle, (s)he gets the more efficient implementation without deep guards. Not quite done yet - the grammars and the manual still need to be updated.

Thanks to my wife Simone, the XDK has a logo now:

The logo expresses the extensibility of XDG, and the building of grammars like with lego blocks, by stacking the three acronym letters on each other. This is also a hint to the multiple levels/dimensions that XDG grammars can have. Well, that's what I interpret :-)

Version 1.5.3 (November 28, 2006)

Added hooks for the FD, FS and Select functors. Now, by choosing the print option in the GUI, all the propagators which are normally posted can now be printed out to stdout (and not posted). This is the first step in the transition to the new Gecode constraint engine, which Guido Tack and me are now undertaking. The very idea to simply flatly print out the propagators is due to Gert Smolka.

Transformed principle constraints such that all infixed FD constraints, e.g. `=:` or `=<:` are now prefixed, using the appropriate prefixed equivalents.

Fixed a bug in `Extras/SolvingStatistics.oz` which would raise an unhandled error whenever a word in the input was not in the lexicon of the grammar.

Removed grammar `Grammars/TAG-wwRwwR.ul`.

Fixed small bugs in some of the grammars/examples files.

Version 1.5.2 (October 25, 2006)

Changed defaults for the order principles: now, the argument variables `Projective` and `Yields` have default `false` instead of `true`. Makes the principles more intuitive to use (I had wondered why the hell switching off the projectivity principle would not give me the overgeneration I expected in `Grammars/ANBN.ul`).

Version 1.5.1 (September 14, 2006)

Updated the manual to reflect all the new stuff in the XDK, especially the new principles.

Lots of fixes around the socket functionality of `xdk.exe` and `xdks.exe`, and in particular solving statistics. Solving statistics would not work at all with grammars coming from a socket etc.

Most important news: now, the XDK supports the English TAG grammar developed in the XTAG project (`http://www.cis.upenn.edu/~xtag/`). This is the first real large-scale grammar for the XDK, which will allow us to improve the system, see where its bottlenecks are etc. The XTAG grammar generator module of the XDK is described in Chapter 6 [XTAG], page 97.

The generated grammars make use of three new principles:

- `principle.xTAG` (Section 7.2.70 [XTAG1], page 132)
- `principle.xTAGRedundant` (Section 7.2.72 [XTAGRedundant], page 134)
- `principle.xTAGRoot` (Section 7.2.73 [XTAGRoot], page 134)

And the new output `output.xTAGDerivation` (Section 9.35 [XTAGDerivation], page 160) to display XTAG derivation trees using the tree viewer from the XTAG project lem parser.

The function “Compare lem solutions” in `xdk.exe` compares the derivation trees obtained from the encoded XDG grammar with the derivation trees obtained from the lem parser.

Version 1.5.0 (September 1, 2006)

Added support for the XTAG grammar. Soon to be documented :)

Version 1.4.11 (August 18, 2006)

Added Selection constraint to the XDK package since the Select package would not individually compile under Mozart 1.3.2 anymore :(Makes the installation of the XDK simpler anyway :)

Version 1.4.10 (July 27, 2006)

Fixed `principle.linkingEndNDG` and changed the constraint in `principle.linkingEnd` accordingly.

Version 1.4.9 (July 26, 2006)

Added principle `principle.sameEdges`.

Renamed `principle.sameorder` to `principle.sameOrder` (used only in `Grammars/diss.ul`, `Grammars/dissNDG.ul` and `Grammars/igk.ul`).

Added new output functor `output.pretty1` for better comparison of solutions.

Version 1.4.8 (July 18, 2006)

Recompiled binaries using the new Mozart 1.3.2.

Changed Diplom grammar (`Grammars/Diplom.ul`) such that it does not use the deprecated `principle.nodeconstraints` and `principle.edgeconstraints` principles anymore but instead `principle.agr`, `principle.agreement`, `principle.government` and the new `principle.agreementSubset` principles.

Added new versions of all principles using edge constraints with the crucial difference that they do not use deep guards. The new versions are suffixed with `NDG` ("no deep guards"). Here are the principles:

- `principle.agreementNDG`

- principle.agreementSubsetNDG
- principle.coindexNDG
- principle.copynpasteNDG
- principle.governmentNDG
- principle.graph1ConstraintsNDG
- principle.graph1NDG
- principle.graphConstraintsNDG
- principle.graphNDG
- principle.linking12BelowStartEndNDG
- principle.linkingAboveNDG
- principle.linkingAboveBelow1or2StartNDG
- principle.linkingAboveEndNDG
- principle.linkingAboveStartNDG
- principle.linkingAboveStartEndNDG
- principle.linkingBelowNDG
- principle.linkingBelow1or2StartNDG
- principle.linkingBelowEndNDG
- principle.linkingBelowStartNDG
- principle.linkingBelowStartEndNDG
- principle.linkingDaughterNDG
- principle.linkingDaughterEndNDG
- principle.linkingEndNDG
- principle.linkingMotherNDG
- principle.linkingMotherEndNDG
- principle.linkingNotDaughterNDG
- principle.linkingNotMotherNDG
- principle.linkingSistersNDG
- principle.partialAgreementNDG

Added new grammars using the new principle versions:

- DiplomNDG.ul
- coindexNDG.ul
- dissNDG.ul
- wwNDG.ul
- wwRNDG.ul
- wwRwwRNDG.ul

Version 1.4.7 (May 9, 2006)

Fixed install scripts, removed grammar `Grammars/MOZ04.ul`.

Added new version of `xdag.sty` improved by Robert Grabowski.

Version 1.4.6 (April 20, 2006)

Added third dimension variable to multidimensional principles which can be parametrized lexically to more closely reflect the principle definitions in the thesis.

Version 1.4.5 (April 19, 2006)

Updated `Grammars/SAT.ul` to properly reflect the grammar in the complexity chapter of the thesis.

Added grammar files to online versions of the manual.

Added principle constraint functors to online versions of the manual.

Changed name (to maintain consistency with the diss): *mapping/map types* are now called *vectors*. That is, in UL, `map(T1 T2)` becomes `vec(T1 T2)`, in IL, `type.map` becomes `type.vec`, and in XML, `typeMap` becomes `typeVec`.

Renamed *disjunctive domains* to `unions`. In IL, `type.disj` becomes `type.union`, and in XML, `typeDisj` becomes `typeUnion`.

Version 1.4.4 (April 5, 2006)

Added first support for partial specification of the solutions before parsing. Each input string can now be extended with file names which contain partial descriptions of sets of nodes, which will be type checked and then used as additional information in the solver. For example, using the grammar `Grammars/diss.ul`, parsing the sentence:

```
mary_L+H*_LH% sees the man with a telescope_H*_LL% . Grammars/diss.nodeset.xml
```

leads to the addition of the additional information in `Grammars/diss.nodeset.xml`

This functionality supports all file types (except precompiled grammars) also available for writing grammars, i.e., UL (suffix `ul`), XML (`xml`) or IL (`ilp` for pickles or `ozf` for functors exporting `set`). XML sockets are also supported (`xmlsocket`).

Improved `Grammars/diss.ul` on the id dimension (no distinction between `padv` and `padj`), new names for prepositions.

Version 1.4.3 (April 3, 2006)

Updated manual to reflect the latest changes, removed some obsolete grammars and principles, added the recent papers etc.

Version 1.4.2 (March 31, 2006)

Improved `Decode` output functor to show only the dimension on which it is used.

New output functors `AllDags1`, `AllDags2`, `AllDags3` and `AllDags4`: show all dags of the multigraph, also those without edges. This is especially useful if interface dimensions (such as idlp etc.) are defined and you would like to see the node record of a node for this dimension.

Similarly, added new output functors `AllLatexs1`, `AllLatexs2`, `AllLatexs3` and `AllLatexs4`.

New principles `principle.linkingAboveBelow1or2Start`, `principle.lockingDaughters`.

Lots of changes in `diss1.ul`. Still improved syntax-semantics interface.

Added progress "report" for grammar compilation (too useful to be just debugging output).

Version 1.4.1 (March 18, 2006)

New grammar `diss1.ul` using the new lexicalized order principle `principle.order3` instead of the old one. Surprisingly good propagation (better than the old non-lexicalized one!).

New principle `principle.projectivity1` enforcing convex `yield` sets instead of `eqdown` sets as `principle.projectivity`.

Removed `Projectivity` argument variable and projectivity constraint from `principle.order3` (use `principle.projectivity1` instead please) to bring the principle more in line with the formalization in the thesis.

Improved principles and outputs menus in GUI `xdk.exe`: now they are ordered by dimension.

Improved latex output functors: now, the index/position of each node is also displayed alongside the word.

Important! Changed the type of the constraint functors, replacing the two arguments `DVA2DIDA` and `ArgRecProc` with the argument `Principle` standing for the instance of the principle in whose context the constraint functor is called. The advantage is that more of the information attached to the principle instance is then available to the constraint functors, including the type of its arguments. The (cosmetic) drawback is that the two functions `DVA2DIDA` and `ArgRecProc` must now be dereferenced from `Principle` first.

Adapted all principles in the principle library to these new conventions.

Adapted the manual to reflect the conventions Section 7.4 [Writing new principles], page 135.

Added new principle `principle.partialAgreement` to handle partial agreement in a generalized way. This principle is used in the new thesis grammar `diss1.ul` to establish partial agreement of relative pronouns with their modified nouns (with respect to gender).

Enhanced `xdag.sty` with new command `penode` for parametrized nodes with an extra node label, which can be freely positioned (done together with Robert Grabowski).

Enhanced `xdag.sty` for better support for "ghosting" nodes and edges instead of just omitting them.

Changed `Dag` and `Latex` output functors to "ghost" nodes and edges instead of just omitting them.

Version 1.4.0 (March 2, 2006)

Added disjunctive domain types as syntactic sugar for combining domain types. Useful for the new lexicalized order principle `principle.order3`, which is formulated on top of a strict total order on the set of edge labels plus the special label "^". can be now written down as:

```
deftype "syn.label" {root subj part obj vinf adv}
deftype "syn.label1" "syn.label" | {"^"}
defentrytype {...
              order: set(tuple("syn.label1" "syn.label1"))
              ...}
```

Adapted the thesis grammars `nut.ul`, `nut1.ul`, `ANBN.ul`, `ANBNCN.ul`, `CSD.ul`, `SCR.ul` and `SAT.ul` to reflect this new syntactic sugar.

Adapted the manual (descriptions of UL, XML and IL syntax) to reflect the new syntactic sugar.

Improvements to the scripts code2pic, ozcolor, ulcolor and xmlcolor.

Improved the principles principle.order3, principle.order3Constraints and principle.order3Dist.

Fixed the principle BarriersAttrib (used wrong syntax for type variables).

Made the domain types "position insensitive" by sorting the atoms lexically (using Mozart's Value.'<') of a domain 1) before type checking (TypeCollector.oz), and 2) when creating lattice functor ADTs (Domain.oz).

Added 100 to the priority of all principles to stay in accord with the formalization given in the thesis. Now, constraints with priority higher than 100 are started first, then come the edge constraints with priority 100, and then the remaining constraints with priority less than 100 (currently, only distribution).

Version 1.3.24 (February 24, 2006)

New output functor output.decode for displaying solutions in the Intermediate Language. This was already possible using the Inspect action from the Explorer, but its output could not be redirected e.g. into the Oz Browser for copy and paste.

Improved generation of orderings, now an arbitrary ordered dimension can be selected (was restricted to lp).

Various improvements to the scripts code2pic, ozcolor, ulcolor and xmlcolor.

Version 1.3.23 (February 21, 2006)

Added new principle principle.order3, which implements the order principle in the thesis. Used in the diss grammars nut.ul, nut1.ul, EQAB.ul, ANBN.ul, ANBNCN.ul, CSD.ul, SCR.ul and SAT.ul.

Merged scripts oz2eps, oz2pdf, oz2jpg, oz2epsn, oz2pdfn, oz2jpgn, ul2eps, ul2pdf, ul2jpg, ul2epsn, ul2pdfn, ul2jpgn into one script code2pic.

Added support for XML source code highlighting (xmlcolor). Merged that into code2pic as well.

Version 1.3.22 (February 7, 2006)

Manual: Improved section Section 4.10 [UL syntax], page 47 (e.g. didn't list the new abbreviations for type reference and class reference).

Improved type checker: didn't check for ambiguous set generator expressions due to doubly occurring constants in the corresponding domains.

Improved xdag.sty: added commands pnode (parametrized node) and rednode (red node). The latter draws the node in red, and the former has an additional argument for pstricks parameters, e.g. to magically remove nodes from sight using the parameter linecolor=white.

Cleaned up the Grammars directory. Need to update the manual to reflect this still.

Added new scripts oz2eps, oz2pdf, oz2jpg, oz2epsn, oz2pdfn, oz2jpgn, ul2eps, ul2pdf, ul2jpg, ul2epsn, ul2pdfn, ul2jpgn for source code highlighting in LaTeX (for the thesis).

Version 1.3.21 (Feburary 3, 2006)

Improved the new example grammars to match better with the thesis, e.g., using the new principle principle.order2 instead of the old one (principle.order). It is less efficient, but has better support for lexicalization.

Added new script `ozcolor` to do Oz-source code coloring.

Fixed a small bug in the GUI ("Solve Examples").

Bumped copyrights to 2006.

Version 1.3.20 (January 6, 2006)

Added stuff written to support the thesis: new grammars `csd.ul` (cross-serial dependencies) `scr.ul` (scrambling), new principle `principle.csd`.

Version 1.3.19 (December 23, 2005)

better support for graphs with cycles: principles fixed (`principle.graph`, `principle.order`), dag output functor improved (though by far not perfect yet).

Version 1.3.18 (December 16, 2005)

Experimental version for Bertrand :)

Version 1.3.17 (September 30, 2005)

Renamed `principle.poetry` to `principle.coindex`, and `poetry.ul` to `coindex.ul`.

Completely rewrote `Grammars/pl.ul` and `principle.pl` (former version would not take care of coreferent variables).

Version 1.3.16 (September 12, 2005)

More improvements to `NewTkDag.oz`.

Changed the underlying finite domain for the type `bool` (Section 4.9.1 [Bool], page 40), which used to encode `true` as `1` and `false` as `2`. Now encodes `false` as `1` and `true` as `2`, so that subtracting one gives the right truth value for reified constraints in Mozart (`0` for `false` and `1` for `true`). Adapted the constraints in the principle library to reflect this:

- `Solver/Principles/Lib/Climbing.oz`,
- `Solver/Principles/Lib/Dag.oz`
- `Solver/Principles/Lib/Order1Conditions.oz`
- `Solver/Principles/Lib/Order2Conditions.oz`
- `Solver/Principles/Lib/OrderConditions.oz`

Added new grammar `Grammars/pl.ul` and new principle `principle.pl`. This is a grammar for propositional logic, where you can enter propositional formulae as sentences, and the XDK enumerates the solutions.

Version 1.3.15 (September 12, 2005)

Added new `NewTkDag.oz` functor for Tcl/Tk dag output which tries to move around edge labels if they overlap. Still experimental, please write email if it does not do the right thing for you!

Version 1.3.14 (September 6, 2005)

Added revised grammar `Grammars/MOZ04.ul` again, since the grammar in the corresponding paper does not work with the new XDK anymore.

Fixed biiig bug in profiler: the profiler multiplied the lexical entry variables and propagators introduced per node with the number of entries for that node, which was wrong and resulted in way too many variables and propagators counted...

Added combined count of finite domain and finite set variables to profiler (called `fdfs`). Also adapted the DTD `Extras/statistics.dtd` to reflect this.

Version 1.3.13 (August 23, 2005)

Fixed `Compiler/Lattices/Tuple.oz`: function `as2I` would return an integer one short. This had repurcussions in `Compiler/Lattices/Set.oz`, and in `Solver/Principles/Order2Conditions.oz`.

Added new principles `principle.barriers.attrib` and `principle.barriers.labels` contributed by Denys Duchier.

Added new experimental principle `principle.poetry` which realizes our first ideas on how to handle FB-LTAG features, and an accompanying example grammar `Grammars/poetry.ul`.

Version 1.3.12 (June 30, 2005)

Added new principle `principle.government2` which allows to govern a feature of the daughter of the daughter of a node (e.g. the verb can now govern the noun under the preposition depending on the verb...).

Version 1.3.11 (June 15, 2005)

Fixed bug in the Dag functor diff functionality.

More enhancements to the `diss.ul` grammar.

Added output functor `output.clls1` which just prints out the CLLS constraint obtained from e.g. `diss.ul` without drawing it using daVinci.

Version 1.3.10 (May 23, 2005)

Augmented `diss.ul` grammar with personal pronouns, perfect constructions and additional lexical entries.

Version 1.3.9 (May 20, 2005)

Added code to enable the use of disjunction for features (e.g. important for valencies).

Version 1.3.8 (May 13, 2005)

Dynamic grammar loading via sockets works now :)

You can now run a server which awaits client connections from the XDK. To make the XDK establish a client connection, attempt to load grammars with suffix `xmlsocket`, e.g. `4712.xmlsocket`. This makes the XDK try to establish a connection with a grammar server on port 4712. For each parse, the XDK client now sends a message containing the sentence to be parsed (as a string) to the server, and expects the server to return a grammar for this sentence, which is then used for parsing.

Bumped copyrights to 2005.

Version 1.3.7 (May 11, 2005)

Added experimental support for dynamic grammar loading via sockets, to be finished in 1.4.0.

Changed example grammars such that the outputs are used and chosen on the lex dimension instead of the multi dimension.

Changed Dag(s) and Latex(s) output functors: `conj` and `pred` edges (emanating from the root) are not left out anymore.

Version 1.3.6 (April 27, 2005)

Fixed `Order` and `Order1` principles to support lexicalization (even though this is not recommended...).

Added more information to the debugging section.

Version 1.3.5 (April 11, 2005)

In debugging mode, the grammar file compiler now prints out the names of the unused lexical classes.

Improved xdag2eps, xdag2pdf and xdag2jpg to cope with input files in other directories than the current one.

Further improvements to the dissertation grammar.

Version 1.3.4 (April 9, 2005)

Added new syntactic sugar for generating order relations. In the UL, you can write e.g. <a b c> for the order relation {[a b] [a c] [b c]}. This only works if the type of the expression is a set of pairs of the same type.

New experimental principles Order2 (constraint functor Order2Conditions), Order2Constraints and Order2Dist.

Improved the XML language to deal with variable features, cf. the example grammar Grammars/Acl01.xml. Important: if you use parametrized lexical classes, please change your XML output and use varFeature instead of feature when instantiating the parameters of a lexical class.

Lots of improvements to the dissertation grammar.

Version 1.3.3 (April 7, 2005)

Many more improvements to grammar Grammars/diss.ul.

New principle LinkingDaughter1Above2. Kind of like parametrized climbing.

Made new principle Locking more general.

Updated diffnotime to also exclude the date when comparing.

Added "proportional" Dag output: now each node has its own width, as opposed to the maximum width. Improves usability a lot.

Version 1.3.2 (April 6, 2005)

New principle Locking.

Heavily improved grammar Grammars/diss.ul, now using the new locking principle for a much cleaner syntax-semantics interface (idpa).

Renamed principle.linkingSubgraphs to principle.subgraphs.

Renamed principle.graphExceptDist to principle.graphConstraints, principle.graph1ExceptDist to principle.graph1Constraints, principle.graphOnlyDist to principle.graphDist, principle.graph1OnlyDist to to principle.graph1Dist, principle.orderExceptDist to principle.orderConstraints, principle.order1ExceptDist to principle.order1Constraints, principle.orderOnlyDist to principle.orderDist, and principle.order1OnlyDist to principle.order1Dist.

Version 1.3.2 (April 6, 2005)

Updated description of the LaTeX output.

Removed Grammars/scatter.ul and Grammars/scatter_wwRwwR.ul, Gapfilling and OrderGap principles again.

Added new grammar Grammars/diss.ul, combining Grammars/Chorus.ul (syntax-semantics interface) and Grammars/igk.ul (information structure) which I have been written for my dissertation.

Updated the Dag output functors to check whether node labels have changed from one analysis to another. Very useful if you just cannot see a difference between two analyses (now you can).

New principle Linking2BelowStart.

Version 1.3.1 (March 30, 2005)

Fixed a bug in the Tcl/Tk dag output which would cause the Explorer to hang, at least under Win XP.

Version 1.3.0 (March 28, 2005)

Significant improvement to the type checker: removed the "any" type, and introduced type variables for principle arguments: each principle can introduce a set of type variables for its arguments, and these can be instantiated by either feature paths or type annotations.

Numerous other bugfixes.

Version 1.2.4 (March 21, 2005)

Small changes to the solving statistics.

Bugfix for passives in Chorus.ul.

Version 1.2.3 (March 20, 2005)

Fixed a nasty bug causing the grammar file compiler to hang in case a graph principle was used on a dimension with an empty set of edge labels.

Version 1.2.2 (March 18, 2005)

Removed xdkcount.exe again and folded its functionality into the solving statistics. This also yielded additions in Extras/statistics.dtd.

Changed default for adding profiling information to solving statistics (now on), applies to xdk.exe and xdks.exe.

Version 1.2.1 (March 16, 2005)

Small changes in the XML output for solving statistics (tag solution renamed to string).

Fixed bug in scripts/addprinciple, scripts/mvprinciple and scripts/rmprinciple (occurred when changing the toplevel makefile.oz), and added some nice output to the scripts telling the user what they are doing.

Ported relative clause principle (for Grammars/Diplom.ul) from old TDG parser.

Optimized propagation of the Dag/Tree functors.

Added propagators introduced by the Edge.oz edge constraint functor to the profiling.

Added solving statistics tool xdkcount.exe.

Version 1.2.0 (March 14, 2005)

Added profiling support for counting the additional variables introduced by the constraint functors, and the propagators they introduce. This involved the following changes:

- added procedure `Profile` to all constraint functors (except `NodeConstraints.oz` and `EdgeConstraints.oz` which are quickly becoming obsolete anyway). The procedure (manually) counts the additional variables (in addition to the variables introduced in the node record) and the propagators.
- added lots of additional code to `Extras/Profile.oz`
- changed the solving statistics (functor: `Extras/SolvingStatistics.oz` to reflect the changes in the profiling functor

Notice: the grammar format for compiled grammars changes with this version, so please recompile your grammars if you have any. Sorry for any inconveniences caused.

Version 1.1.4 (March 9, 2005)

Removed hardcoded `multi` dimension (and mapping from dimension variable `Multi` to `multi`).

Added mapping from dimension variable `This` to the dimension id of the currently defined dimension.

Version 1.1.3

Added perl-scripts for adding/removing/renaming principles and accompanying constraints: `scripts/addprinciple`, `scripts/rmprinciple` and `scripts/mvprinciple`. Should ease the tedious addition/removal/renaming of principles.

Updated parameter names for the linking principles according to the current thesis version.

Renamed `principle.linkingBelow12StartEnd` principle to `principle.linking12BelowStartEnd`.

Added `principle.linkingBelow1Start1End2` principle, and grammar `Grammars/ESSLLI04_idlppasc.ul` (without deep syntax, instead slightly more complicated linking via the above new principle).

Version 1.1.2

Fixed small bug in `xdk.oz` which hit when you switched off a dimension, called Generate Orderings, and then selected "Outputs" for one of the analyses.

Version 1.1.1

Added script `scripts/xdag2jpg` to convert LaTeX-output dependency graphs into JPG. This is very useful for those notorious applications which do not support EPS/JPG image import, most notably, Microsoft Office (Word, PowerPoint) and OpenOffice.

Version 1.1.0

Totally re-organized the linking/dominance principles according to their formalizations in my thesis, making it a lot more systematic than before. Now, it should be much clearer which principle does what, and when to use which and so on.

The re-organization has led to the following changes:

- replaced `principle.linking` by `principle.linkingEnd`,
- replaced `principle.directcolinking` by `principle.linkingDaughterEnd`
- replaced `principle.directcontralinking` by `principle.linkingMotherEnd`
- split `principle.immediate` into `principle.linkingDaughter` (co-immediate) and `principle.linkingMother` (contra-immediate)

- split `principle.notImmediate` into `principle.linkingNotDaughter` (co-not-immediate) and `principle.linkingNotMother` (contra-not-immediate)
- split `principle.dominance` into `principle.linkingBelow` (co-dominance) and `principle.linkingAbove` (contra-dominance)
- split `principle.ldominance` into `principle.linkingBelowStart` (l-co-dominance) and `principle.linkingAboveEnd` (l-contra-dominance)
- added `principle.linkingBelowEnd` and `principle.linkingAboveStart`
- replaced `principle.colinking` by `principle.linkingBelowStartEnd`
- replaced `principle.contralinking` by `principle.linkingAboveStartEnd`
- replaced `principle.dominance1` by `principle.chorus`
- replaced `principle.sisters` by `principle.linkingSisters`
- replaced `principle.sublinking` by `principle.linkingSubgraphs`
- replaced `principle.colinking1` by `principle.linkingBelow12StartEnd`

Please also notice, that the features of the principles are different, as are their defaults. Please check the principles list (Section 7.2 [Principles list], page 103) to adapt your grammars, and do not hesitate to ask me if any problems occur. Sorry in advance for any inconveniences this may cause.

This renovation affected the principle library of course, the grammars (I adapted all grammars in `Grammars/` to reflect the changes), Ondrej's PDT grammar generation code (I have adapted that as well) and the manual. VAST changes to do, but hopefully not very difficult to adapt to for you users.

The debug output at grammar compilation stage (`Compiler/Encoder.oz`) is now more informative: it prints out the number of actual lexical entries stored in the compiled out grammar.

Version 1.0.6

Heavily improved the type system according to what I write in my upcoming thesis:

- added new type `card` for cardinalities, and the corresponding lattice functor `Card.oz`
- removed the superfluous `Map.oz` lattice functor, its work was already done by `Record.oz`
- removed the `Valency.oz` lattice functor, its work is now done by `Record.oz` and `Card.oz`
- removed the following `Ints.oz` lattice functor, and pushed its functionality into the set lattice functor `Set.oz` (sub lattice `intSet`)

The result is a cleaner and leaner type checker. This also fixed some bugs in the type system which were due to the slightly faulty design. E.g. it was impossible to write down sets of integers so far. And it was possible to write down valency sets which were no sets.

I guess I could have introduced some new bugs though, since I undertook quite a lot of changes. Please test your stuff and report any new errors, or even unusable error messages etc.

Version 1.0.5

Removed the non-monotonic behavior of the greatest lower bound (glb) operations of strings and lists. Now, all glb operations are commutative again.

The result of the prior change is that lists cannot be concatenated anymore. Since nobody has actually ever used this functionality, this should not come as a big loss.

Strings on the other hand still can: I introduced an explicit concatenation operator (`@` in the UL, `<concat>` in the XML language). An example of how to use this can be found in `Grammars/igk.ul`.

These changes make the XDK cleaner, and more similar to the actual formalization which I am currently working out for my thesis.

Version 1.0.4

Added new experimental TAG-encoding grammars `Grammars/scatter.ul` and `Grammars/scatter_wwRwwR.ul`.

Added new experimental principles `Gapfilling` and `OrderGap` to accompany the two grammars above.

Added window resizing functionality to the ManualOracle-code, and fixed a bug there.

Version 1.0.3

Slightly optimized the propagation behavior of the Order and Climbing principles.

The Climbing principle is now parametrizable. Attention if you use the climbing principle on dimensions which are not trees: you must now explicitly set the parameter `MotherCards` to `false` in this case to turn off an optimization restricted to trees.

Version 1.0.2

Added automatic resizing of Dag output windows to fit on the screen, e.g. to make MacOS X users' life easier 8-) This was also pointed out by Mark, thanks again!

Added version information to the About window in the `xdk`-executable.

Version 1.0.1

Set/DomainTupleSet could not encode sets but only set generators. Fixed. Pointed out by Mark Pedersen, thanks!

Re-added `duchier-select.pkg` and the native functor object files (`*.o`) to the Windows binary distribution in order to be able to compile it using ozmake.

Version 1.0.0

It's done!

Version 0.9.16

Changed shortcut for `--port` (in `xdg` and `Oracles/ManualOracle/Server.exe`) from `-t` to `-p`.

Made profiling optional for parse statistics.

Added commandline option `--(no)profile` (`-f`) to toggle profiling in the `xdk` and `xdks` programs.

Added menu checkbutton `Profile` to the `xdk` program.

Made `gprofile` and `sprofile` elements optional in `Extras/statistics.dtd`.

Renamed `sentence` element to `solution` again.

Version 0.9.15

Added new profiling functionality: the counting of fd/fs variables introduced. This involved changes in the lattice functor implementation in `Compiler/Lattices` (added a new function `Count` to each lattice).

This is also used in the new solving statistics output, and thus, I changed the DTD (`Extras/statistics.dtd`):

- the `statistics` element has now a new obligatory daughter element `gprofile` (grammar profile)
- I renamed the `solution` element to `sentence` (which makes more sense)
- this `sentence` element has now a new obligatory daughter element `sprofile` (sentence profile)

Further improvements in the manual (types, lattices, lattice functors).

Version 0.9.14

Updated the parts of the manual pertaining to 1) the grammars, 2) the outputs, and 3) the principles. Especially the latter is now much better and contains much more comprehensible explanations of the principles and their behaviors.

Version 0.9.13

Added `principle.sameorder` to equate the positions on ordered dimensions.

Added boolean parameter `Yields` to the order principles (true by default). If true, the yields of a node are ordered, if false, the daughters.

Version 0.9.12

Renamed XDK executables for consistency: xdg `->` xdk, xdgc `->` xdkc, xdgs `->` xdks and xdgconv `->` xdkconv.

Version 0.9.11

Added check for undefined values in lexical entries (throws an exception if so). Quite optimized so—only those features which *can* be undefined at all.

Added same check for undefined values also for principle arguments (except for constraint arguments which are not checked, too much effort for too little gain, IMHO).

Version 0.9.10

Made the order of the dimensions more consistent (always alphabetically ordered now), e.g. in the GUI menus and in the Dags output functors.

Increased the associativity strength of the conjunction (`&`) and disjunction (`|`) operators in the UL to make for less bracketing.

Changed the behavior of the List and String types. Their top value is now resp. the empty list and the empty string (instead of, as before, being undefined), and their greatest lower bound is now the concatenation of resp. the two lists and strings under consideration (instead of, as before, bottom if the two were different and none of them was the top value). I.e. now you can even do some simple morphology in the XDG lexicon.

The Dag and Latex output functor families behave more consistently.

Latex functor now escapes LaTeX special characters properly (`$`, `&`, `%`, `#`, `_`, `{`, `}`, `~`, `^`, `\`).

New Latex, Latex1, Latex2, Latex3 output functors for individual dimensions (corresponding to Dag, Dag1, ...).

New Latexs, Latexs1, Latexs2, Latexs3 output functors for multiple dimensions (corresponding to Latex, Latex1, ...).

Improved xdag2eps, xdag2pdf.

Added ulcolor script (uses LaTeX color package to do UL code highlighting).

Added big new German grammar done by Regine Bader, Christine Foeldesi, Ulrich Pfeiffer and Jochen Steigner (building on my Diplom grammar).

Added Benoit Crabbe's first version of his French grammar (Benoit.ul).

Added the first attempt to handle Information Structure (igk.ul) done at the IGK annual meeting in Edinburgh (Ciprian Gerstenberger, Oana Postolache, Stefan Thater and Maarika Traat).

Put all scripts in a separate directory (`scripts`).

And many many more improvements.

Version 0.9.8

Introduced the multi dimension for collecting multi-dimensional principles, corresponding lexical attributes, and outputs, in addition to the lex dimension.

Added ESSLLI 2004 example grammars, removed COLING04 and DG04 grammars, updated Chorus grammar.

Added French example grammar by Benoit Crabbe.

New principles: CoLinking, CoLinking1 and ContraLinking

Lots of additional fixes and stuff.

Version 0.9.7

Renamed `latex2eps` to `xdag2eps`, and also `latex2pdf` to `xdag2pdf`.

Much improved type checking (and error output) for type annotations and feature paths (esp. useful for principle type checking).

Version 0.9.6

Introduced the type 'type.any'. This can be used to bypass the type checker, and is useful e.g. for the principles surrounding the notion of agreement (`principle.agr`, `principle.agreement`, `principle.government`). Here, 'type.any' is used for principle arguments which are not known a priori. Notice that this type cannot be used for values, only for feature paths (i.e. value descriptions).

Version 0.9.5

Due to popular request, type references (`ref`) and class references (`useclass`) can now be invoked just by the identifier, i.e. you can omit the respective keywords.

Examples: `ref("id.attrs")` can be written tersely as `"id.attrs"`. Similarly, `useclass "noncan" {Word: "koennen"}` can be written tersely as `"noncan" {Word: "koennen"}`.

You can use the shell script `ulterse` to convert an "old style" UL grammar file into a "new style", terser one, e.g. `./ulterse Acl01.ul` converts `Acl01.ul`.

References

Regine Bader, Christine Foeldesi, Ulrich Pfeiffer and Jochen Steigner 2004, *Modellierung grammatischer Phaenomene der deutschen Sprache mit Topologischer Dependenzgrammatik*, Softwareprojekt-Bericht, Saarland University, `http://www.ps.uni-sb.de/~rade/fopra/softproj.pdf`.

Ondrej Bojar 2004, *Problems of Inducing Large Coverage Constraint-Based Dependency Grammar*, Proceedings of CSLP 2004, Roskilde/DK, `http://www.ps.uni-sb.de/~rade/papers/related/Bojar04.pdf`.

Benoit Crabbe 2003, *Lexical Classes for Structuring the Lexicon of a TAG*, In Prospects and Advances in the Syntax/Semantics Interface, Nancy/FRA, `http://www.loria.fr/~duchier/Lorraine-Saarland/crabbe.ps`.

Benoit Crabbe and Denys Duchier 2004, *Metagrammar Redux*, Proceedings of CSLP 2004, Roskilde/DK, `http://www.loria.fr/~crabbe/`.

Marie-Helene Candito 1996, *Generating an LTAG out of a Principle-based Hierarchical Representation*, In Proceedings of ACL 1996, Santa Cruz/US, `http://acl.ldc.upenn.edu/P/P96/P96-1045.pdf`.

Ralph Debusmann 2001, *A Declarative Grammar Formalism For Dependency Grammar*, Diplomarbeit, Saarland University, `http://www.ps.uni-sb.de/~rade/papers/da.pdf`.

Ralph Debusmann 2001, *Movement as well-formedness conditions*, Proceedings of the ESSLLI 2001 Student Session, Helsinki/FI, `http://www.ps.uni-sb.de/~rade/papers/esslli01.pdf`.

Ralph Debusmann 2001, *Topologische Dependenzgrammatik*, In Proceedings of StuTS 2001, Saarbrcken/DE, `http://www.ps.uni-sb.de/~rade/papers/stuts01.pdf`.

Ralph Debusmann 2003, *Dependency Grammar as Graph Description*, In Prospects and Advances in the Syntax/Semantics Interface, Nancy/FRA, `http://www.ps.uni-sb.de/~rade/papers/passi03.pdf`.

Ralph Debusmann 2003, *A Parser System for Extensible Dependency Grammar*, In Prospects and Advances in the Syntax/Semantics Interface, Nancy/FRA, `http://www.ps.uni-sb.de/~rade/papers/passi03demo.pdf`.

Ralph Debusmann and Denys Duchier 2002, *Topological Dependency Analysis of the Dutch Verb Cluster*, unpublished manuscript, `http://www.ps.uni-sb.de/~rade/papers/acl02.pdf`.

Ralph Debusmann and Denys Duchier 2003, *A Meta-Grammatical Framework for Dependency Grammar*, unpublished manuscript, `http://www.ps.uni-sb.de/~rade/papers/acl03.pdf`.

Ralph Debusmann and Denys Duchier 2003, *Towards a Syntax-Semantics Interface for Topological Dependency Grammar*, unpublished manuscript, `http://www.ps.uni-sb.de/~rade/papers/taln03.pdf`.

Ralph Debusmann 2004, *Taking Linguistic Dimensions More Seriously: The New Grammar Formalism of Extensible Dependency Grammar*, unpublished manuscript, `http://www.ps.uni-sb.de/~rade/papers/acl04.pdf`.

Ralph Debusmann 2004, *Multiword expressions as dependency subgraphs*, In Proceedings of the ACL 2004 Workshop on Multiword Expressions: Integrating Processing, Barcelona/ES,
`http://www.ps.uni-sb.de/~rade/papers/acl04mwe.pdf`.

Ralph Debusmann and Denys Duchier and Geert-Jan M. Kruijff 2004, *Extensible Dependency Grammar: A New Methodology*, In Proceedings of the COLING 2004 Workshop on Recent Advances in Dependency Grammar, Geneva/CH,
`http://www.ps.uni-sb.de/~rade/papers/dgcoling04.pdf`.

Ralph Debusmann and Denys Duchier and Marco Kuhlmann and Stefan Thater 2004, *TAG Parsing as Model Enumeration*, In Proceedings of TAG+7, Vancouver/CA,
`http://www.ps.uni-sb.de/~rade/papers/tag7.pdf`.

Ralph Debusmann and Denys Duchier and Alexander Koller and Marco Kuhlmann and Gert Smolka and Stefan Thater 2004, *A Relational Syntax-Semantics Interface Based on Dependency Grammar*, In Proceedings of COLING 2004, Geneva/CH,
`http://www.ps.uni-sb.de/~rade/papers/coling04.pdf`.

Ralph Debusmann and Denys Duchier and Marco Kuhlmann 2004, *Multi-dimensional Graph Configuration for Natural Language Processing*, In Proceedings of CSLP 2004, Roskilde/DK,
`http://www.ps.uni-sb.de/~rade/papers/cslp04.pdf`.

Ralph Debusmann and Denys Duchier and Joachim Niehren 2004, *The XDG Grammar Development Kit*, In Proceedings of MOZ04, Charleroi/BEL,
`http://www.ps.uni-sb.de/~rade/papers/moz04.pdf`.

Ralph Debusmann and Oana Postolache and Maarika Traat 2005, *A Modular Account of Information Structure in Extensible Dependency Grammar*, In Proceedings of CICLING 2005, Mexico City/MX,
`http://www.ps.uni-sb.de/~rade/papers/igk.pdf`.

Ralph Debusmann and Denys Duchier and Andreas Rossberg, *Modular Grammar Design with Typed Parametric Principles*, In Proceedings of FGMOL 2005, Edinburgh/UK,
`http://www.ps.uni-sb.de/~rade/papers/fgmol05.pdf`.

Ralph Debusmann and Gert Smolka 2006, *Multi-dimensional Dependency Grammar as Multigraph Description*, In Proceedings of FLAIRS 2006, Melbourne/US,
`http://www.ps.uni-sb.de/~rade/papers/flairs.pdf`.

Ralph Debusmann 2006, *Extensible Dependency Grammar - A Modular Grammar Formalism Based On Multigraph Description*, PhD thesis, Saarland University,
`http://www.ps.uni-sb.de/~rade/papers/diss.pdf`.

Ralph Debusmann 2007, *Scrambling as the Combination of Relaxed Context-Free Grammars in a Model-Theoretic Grammar Formalism*, ESSLLI 2007 Workshop Model Theoretic Syntax at 10, Dublin/IE,
`http://www.ps.uni-sb.de/~rade/papers/mts10.pdf`.

Ralph Debusmann 2007, *Ralph Debusmann The Complexity of First-Order Extensible Dependency Grammar*, unpublished manuscript,
`http://www.ps.uni-sb.de/~rade/papers/fg07.pdf`.

Ralph Debusmann and Marco Kuhlmann 2007, *Dependency Grammar: Classification and Exploration*, Project Report (CHORUS, SFB 378), Saarland University,
`http://www.ps.uni-sb.de/~rade/papers/sfb.pdf`.

Peter Dienes, Alexander Koller and Marco Kuhlmann 2003, *Statistical A-Star Dependency Parsing*, Workshop: Prospects and Advances in the Syntax-Semantics Interface, Nancy/FRA,
`http://www.ps.uni-sb.de/~rade/papers/related/DienesEtal03.pdf`.

Denys Duchier 1999, *Axiomatizing Dependency Parsing Using Set Constraints*, In Proceedings of MOL6, Orlando/US,
`http://www.ps.uni-sb.de/~rade/papers/related/Duchier99.pdf`.

Denys Duchier 2001, *Lexicalized Syntax and Topology for Non-projective Dependency Grammar*, In Proceedings of FGMOL-01, Helsinki/FI,
`http://www.ps.uni-sb.de/~rade/papers/related/Duchier01.pdf`.

Denys Duchier 2003, *Configuration Of Labeled Trees Under Lexicalized Constraints And Principles*, Journal of Research on Language and Computation,
`http://www.ps.uni-sb.de/~rade/papers/related/Duchier03.pdf`.

Denys Duchier 2003, *A Metagrammatical Formalism for Lexicalized TAGs*, In Prospects and Advances in the Syntax/Semantics Interface, Nancy/FRA,
`http://www.loria.fr/~duchier/Lorraine-Saarland/duchier-demo.ps`.

Denys Duchier and Ralph Debusmann 2001, *Topological Dependency Trees: A Constraint-Based Account of Linear Precedence*, In Proceedings of ACL 2001, Toulouse/FRA,
`http://www.ps.uni-sb.de/~rade/papers/acl01.pdf`.

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,
`http://www.ps.uni-sb.de/~rade/papers/jolli04.pdf`.

Alexander Koller and Kristina Striegnitz 2002, *Generation as Dependency Parsing*, In Proceedings of ACL 2002, Philadelphia/US,
`http://www.ps.uni-sb.de/~rade/papers/related/KollerStriegnitz02.pdf`.

Christian Korthals and Ralph Debusmann 2002, *Linking syntactic and semantic arguments in a dependency-based formalism*, In Proceedings of COLING 2002, Taipei/TW,
`http://www.ps.uni-sb.de/~rade/papers/coling02.pdf`.

Christian Korthals 2003, *Unsupervised Learning of Word Order Rules*, Diploma thesis, Saarland University,
`http://www.ps.uni-sb.de/~rade/papers/related/Korthals03.pdf`.

Christian Korthals and Geert-Jan M. Kruijff 2003, *Unsupervised, graph-based learning of topological fields*, unpublished manuscript,
`http://www.ps.uni-sb.de/~rade/papers/related/KorthalsKruijff03.pdf`.

Marco Kuhlmann 2002, *Towards a Constraint Parser for Categorial Type Logics*, MSc. Thesis, University of Edinburgh,
`http://www.ps.uni-sb.de/~rade/papers/related/Kuhlmann02.pdf`.

Pierre Lison 2006, *Implmentation d'une Interface Smantique-Syntaxe base sur des Grammaires d'Unification Polarises*, MSc. Thesis, Universit Catholique de Louvain,
`http://www.ps.uni-sb.de/~rade/papers/related/Lison06.pdf`.

Mathias Moehl 2004, *Modellierung natuerlicher Sprache mit Hilfe von Topologischer Dependenzgrammatik*, Forschungsprojekt-Bericht, Saarland University,
`http://www.ps.uni-sb.de/~rade/papers/related/Moehl04.pdf`.

Renjini Narendranath 2004, *Evaluation of the Stochastic Extension of a Constraint-Based Dependency Parser*, BSc. Thesis, Saarland University,
`http://www.ps.uni-sb.de/~rade/papers/related/Narendranath04.pdf`.

Marwan Odeh 2004, *Topologische Dependenzgrammatik frs Deutsche*, Forschungspraktikum-Bericht/BSc. Thesis, Saarland University,
`http://www.ps.uni-sb.de/~rade/fopra/Arabic.pdf`.

Jorge Marques Pelizzoni and and Maria das Gracas Volpe Nunes 2005, *N:M Mapping in XDG - The Case for Upgrading Groups*, In Proceedings of the Workshop Constraint Solving and Language Processing, CSLP 2005, Sitges/ES,
`http://www.ps.uni-sb.de/~rade/papers/related/PelizzoniDasGracasVolpeNunes05.pdf`.

Jochen Setz 2007, *A Principle Compiler for Extensible Dependency Grammar*, BSc. Thesis, Saarland University,
`http://www.ps.uni-sb.de/~rade/papers/related/Setz07.pdf`.

Concept index

K

L

M

N

O

P

R

S

T