

# The PLAIN User's Guide

Version 2.0 (September 2014)

[www.plain-nlp.de](http://www.plain-nlp.de)

© Peter Hellwig and Heinz-Detlev Koch

## ***Abstract:***

PLAIN (Programs for Language Analysis and Inference) is an integrated development environment (IDE) which provides comprehensive facilities to (computational) linguists for creating and processing lingware. PLAIN adheres to Dependency Unification Grammar (DUG), a particular formal approach to describe natural languages. DUG aims at a simple and at the same time broad coverage of linguistic phenomena. Dependency Representation Language (DRL) is the formalism of DUG. The PLAIN IDE provides an interpreter of resources written in DRL for various purposes. The main functions are the Parser, the Transducer, and the Generator. The parser reads natural language input, consults the grammatical resources (lexicon, "synframes" and "templates") written in DRL and yields a syntactic description in DRL format. The transducer reads a DRL construct, consults rules written in DRL and outputs new DRL constructs. The generator turns DRL constructs in natural language surface.

This paper explains how to use the software.

## Contents

1	Background .....	3
2	Starting the PLAIN IDE .....	4
3	The File menu .....	5
3.1	Project administration .....	6
3.2	New Project .....	6
3.3	Open Project .....	9
3.4	Project Settings .....	9
3.5	Augment Project .....	9
3.6	Recreate Database .....	10
3.7	Activate Logfile .....	11
4	The Categories menu .....	11
4.1	Show Attributes .....	11
4.2	Attribute Filtering .....	12
5	The Lists menu .....	13
5.1	Enter a List .....	13
5.2	Read and Write Lists .....	14
5.3	Find Lists in the Database .....	15
6	The Morphology menu .....	17
6.1	Show Paradigms .....	17
6.2	Paradigm Content .....	17
6.3	Lookup .....	18
6.4	Generate Word Forms from Root .....	19
6.5	Morphology layers and converters .....	20
7	The Parser menu .....	20
7.1	Parse Manual Input .....	21
7.2	Parse a file .....	21
7.3	Tools .....	23
7.4	Show Short Result .....	23
7.5	Show Long Result .....	24
7.6	Show Full Result .....	24
7.7	Show Rooted Chart Elements .....	24
7.8	Show All Chart Elements .....	24
7.9	Show Sorted Chart Elements .....	25
7.10	Show Bulletin .....	25
7.11	Show all Bulletins of a Result .....	25
7.12	Compare Bulletins .....	25
7.13	Show Diagnosis .....	25
8	The Tagger menu .....	26
8.1	Lexical Lookup Only .....	27
8.2	Parse the Input .....	27
9	The Transducer menu .....	28
9.1	Apply Rule to List .....	29
9.2	Replace List .....	31
10	Generator .....	33
10.1	Generate Word Forms from Lexeme .....	33
10.2	Generate Natural Language from DRL .....	34
11	Converters .....	36
11.1	SGML to XML .....	36
11.2	CardIForms to MorphUnits .....	36
11.3	MorphUnits to Paradigms .....	37
11.4	CardIForms to Paradigms .....	38
11.5	Test Examples to Parser Input .....	39

## 1 Background

The purpose of the computer system PLAIN ("Programs for Language Analysis and Inference") is drawing up and processing models of natural languages. The program interprets resources written in Dependency Representation Language (DRL) and performs the corresponding linguistic actions, among which are morphological and syntactic analysis and semantic processing.

DRL is the formalism of Dependency Unification Grammar (DUG). DUG is a particular formal approach to natural languages. For details please consult Peter Hellwig "Dependency Unification Grammar" In *Dependency and Valency. An International Handbook of Contemporary Research*. Edited by V. Agel, L.M. Eichinger, H.-W. Eroms, P. Hellwig, H.-J. Heringer, H. Lobin. Berlin, Mouton 2003, pp 593-635.

On the one hand, DRL is designed to cover a broad range of linguistic phenomena, including complements and adjuncts, nucleus and raising, compounds, discontinuous constituents, coordination, ellipsis. On the other hand, DRL is perspicuous and simple so that grammarians and lexicographers can easily draw up large data resources. The acronym "plain" has been chosen in order to allude to this simplicity. Please consult "The Linguist's Guide to PLAIN" for details about the linguistic resources necessary to run PLAIN.

In connection with PLAIN as an interpreter, DRL is kind of a high level programming language which is used by the linguist in order to make the computer analyze natural language input. This may happen in a teaching environment or in order to test the quality of linguistic descriptions as well as in real NLP applications.

The PLAIN IDE is an *Integrated Development Environment* first of all for the linguist. It provides many tools for drawing up, testing and debugging the linguist's descriptions. The main linguistic modules of the system are:

- the morphological analyzer,
- the parser,
- the transducer,
- the generator of surface text.

In addition, the system provides the following infrastructure:

- the user interface,
- the database.

The morphological analyzer (also called the *scanner*) segmentizes input text and classifies the segments according to the linguist's lexical description. The *parser* detects the syntagmatic relationships between the segments and classifies them according to the linguist's grammatical description. The *transducer* derives one formal representation from another. Depending on the type of these transformations (e.g. paraphrasing, inferring, translating, summarizing), this is a way to model the lexical-semantic, the logical-semantic and the text-linguistic system of the language. The *generator* turns the resulting formal representations into natural language text. Data is passed from one module to the other via input and output queues. The *Plain IDE* provides a graphical user interface implemented in tcl. All linguistic resources are stored in a single *database* optimized for retrieval.

## **2 Starting the PLAIN IDE**

In the sequel the functionality of the PLAIN IDE is sketched. Starting the PlainIDE.exe results in Fig. 1.

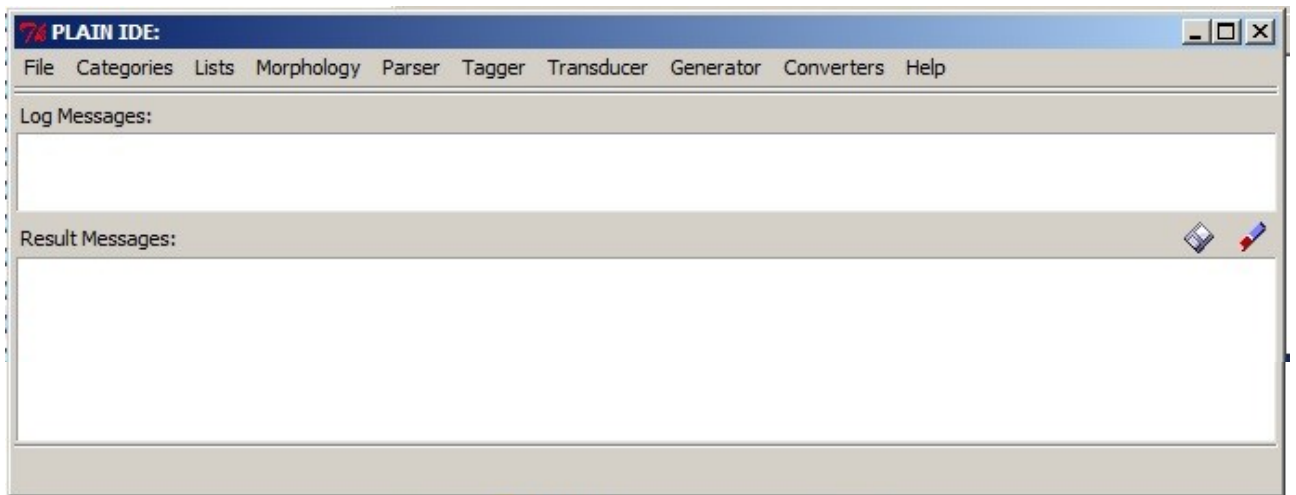




Fig. 1 Initial Window

The *Main Window* contains the menu bar and two boxes. The *Log Messages* box displays certain messages during execution including error messages. The *Result Messages* box displays the results of the operation requested via the menu lists. The contents of the *Result Messages* box can be stored by clicking the  icon. The box can be cleared by clicking the  icon.

### 3 The File menu

Clicking on the *File* button results in Fig. 2.

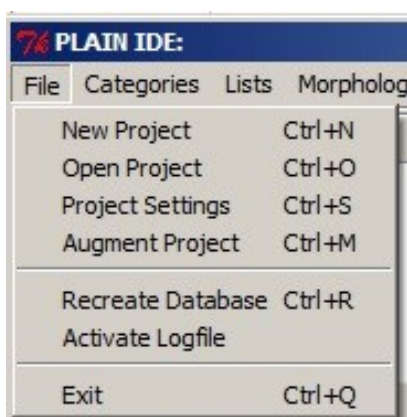


Fig. 2 The *File* drop-out menu

### 3.1 Project administration

The first few items on the list in Fig. 2 serve the purpose of project administration. A *project* is a particular constellation of linguistic resource files. These files are turned into an internal database for the program's execution. Each database consists of two automatically created files carrying the name of the project and the extensions .dat and .idx. A third file with the name of the project and the extension .prj saves the definition of the resource files that provide the input to this particular database.

If projects exist then the PLAIN IDE will automatically open the last project worked on. The latter information is stored in a file with the extension *.plainide*. If no project exists or if a database with a new constellation of resources should be built then a new project must be defined.

### 3.2 New Project

The *New Project* command results in a pop-up menu as displayed in Fig. 3. The files that are to form the project must be entered in the appropriate boxes.

First, a file carrying the name of the project with the extension .prj must be entered in the box *Save Project as*. This file is updated automatically and is going to collect all parameters of the project.

Next, the *Database Directory* must be specified in which the .dat and .idx files forming the database should be stored.

All other entries indicate resource files which have been drawn up previously by the linguist using her/his external editor. See details about the format of these files in "The Linguist's Guide to PLAIN".

A first group of files forms the *PLAIN Lexicon*. A file containing the *Category Definition* of the symbols to be used in the DRL must always be specified. Various files which form *Lexicon Subsets* may be added. They cover the morphology of

the language in question. If the database contains DRL lists that should be retrieved by means of *Partitioning and Indexing* then a *Definition File* must be existing that contains the database partitions and list indexes. The resources of the *Parser* and the *Transducer* require this scheme.

The next group of files supplies the resources for the *PLAIN Parser* and for the *Generator*. In addition to a *PLAIN Lexicon* we need a file with *Synframes*, i.e. a set of syntactic frames for the lexemes of the language. These frames define the combination capability of words in terms of complements and adjuncts. Furthermore we need a file with *Templates*, i.e. a detailed description of any complement and adjunct mentioned in the Synframes.

The last group comprises files for the *PLAIN Transducer*. These files must be entered in the box *Instances and Rules*. They describe operations like paraphrasing, translation, logical deduction.

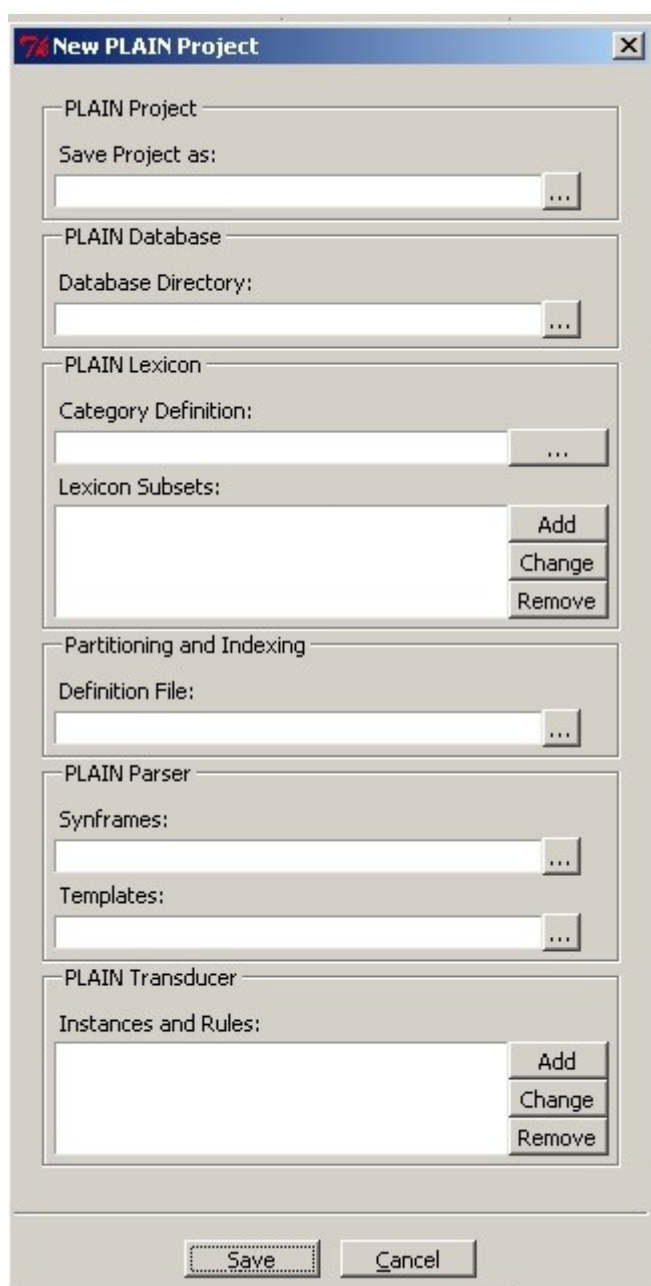


Fig. 3 Window for defining a new project

Press the *Save* button after all resources for the new project have been specified. As a result, the specification is stored in the indicated *.prj* file and a new *.dat* and a new *.idx* file are created in the indicated *Database Directory*. The text files forming the resources of the new project are all turned into an internal format and stored in these two data sets.



### 3.3 Open Project

If a new project is saved then the database of this project is opened. If you start PLAIN then the last project worked on is automatically opened. If you want to work with another database you have to issue the *Open Project* command in the *File* drop-out menu. A browser appears from which you can choose an existing project file.

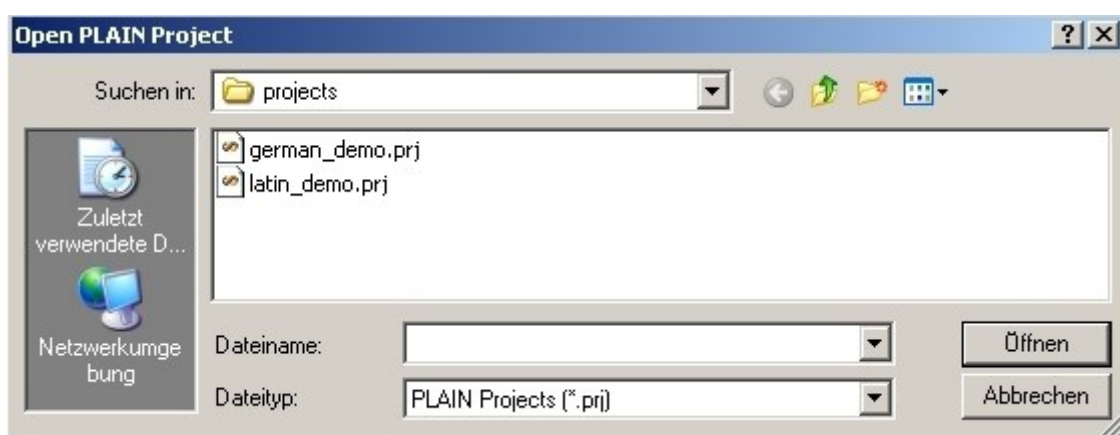


Fig. 4 Choosing an existing project

### 3.4 Project Settings

A project must be opened before you can issue the *Project Settings* command. The pop-up menu you get when clicking *Project Setting* is similar to the one for *New Project*. The resource files of the project are displayed in the corresponding boxes. You can now add, change and remove files from the project. If you subsequently press the *Save* button then the whole database is recreated. The former .dat and .idx files of the project are deleted, the text files now forming the resources of the project are transformed into the internal format and stored in a new version of the two data sets.

### 3.5 Augment Project

A project must be opened before you can issue the *Augment Project* command. The pop-up menu you get when clicking *Augment Project* is similar to the one for

*New Project*. The boxes *Save Project as* and *Database Directory* are predefined. All other boxes are empty. You can enter any resource file you want to add to the project.

As opposed to *Project Settings*, pressing the *Save* button does not result in a recreation of the database (the .dat and .idx files). Instead the files specified in the *Augment Project* pop-up menu are transformed into internal format and added to the existing .dat and .idx file. This saves computer time, especially if you have a very big database. However, the user must be sure that he/she does not augment the database with any data that has already been included. The latter would result in double entries which suggest ambiguity.

### 3.6 Recreate Database

A project must be opened before you can issue the *Recreate Database* command. Recreating the Database means that the existing .dat and .idx files are deleted, that the resource files forming the project are transformed into internal format and then stored in a new version of the .dat and .idx files.

The database is recreated if you changed the resource files specification by means of the *Project Settings* menu and then press the *Save* button. The database is also recreated if the program detects a modification of any resource file when opening a project and you answer "yes" to the following message:

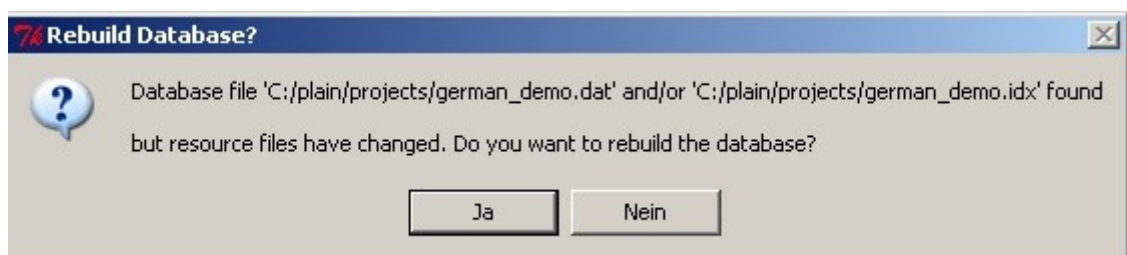


Fig. 5 Prompting the recreation of the database

It is advisable to issue the *Recreate Database* Command at restart after each abnormal end of the program, even if the Log message *Database is up to date* appears. The database may be corrupt due to the crash cause strange errors.

### 3.7 Activate Logfile

If the *Activate Logfile* command is issued, a documentation of all subsequent processes is written into a log file. This file is stored in the directory *C:/Plain/log*. Its name is *plain\_yyyymmdd-hhmmss.log* (y=year, m=month, d=day, h=hour, m=minute, s=second of the execution time.) The user might consult this file if the program does not behave as desired and the reason is not clear. The file could also be sent to the PLAIN service team in order to get help or to indicate a bug.

## 4 The Categories menu

The expressions of the DRL consist of complex categories. Each category comprises a set of attribute and values. See details about the DRL syntax in "The Linguist's Guide to PLAIN". Any attribute must be defined before it can be used in a DRL expression. This is necessary because different attributes may be processed differently. The definition of attributes must be drawn up in the file that is specified in the *Categories Definition* box of the *New Project* menu. The format of this file is determined in "The Linguist's Guide to PLAIN".

Pressing the *Categories* button results in the following drop-out menu:

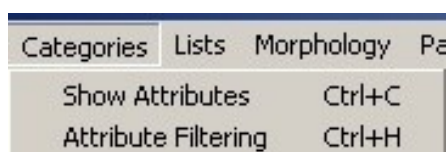


Fig. 6 The *Categories* drop-out menu

### 4.1 Show Attributes

*Show Attributes* displays the set of attributes defined for the current project, preceded by a code for the attribute type. This command is used, first of all, to check whether the definition of attributes succeeded and is correct.

## 4.2 Attribute Filtering

*Attribute Filtering* is a device to vary the DRL representation. The amount of attributes processed by the scanner, the parser or the transducer may be large. By attribute filtering the display can be tuned to just those attributes the user is interested in.

The *Attribute Filtering* command results, for example, in the following pop-up window.



Fig. 7 Attribute Filtering

The two boxes of the window contain the attributes to be visible or to be hidden. The user has to mark an attribute and then click on the appropriate arrow to move it from one box to the other. Pressing the *On* button at the end activates the choice.

If *Attribute Filtering* should be abandoned and all attributes should be shown again then just press the *Off* button. No attribute filtering is the default as long as *Attribute Filtering* has not been invoked.

Note: The sequence of the attributes in a DRL term is the same as the sequence of their definitions in the *Category Definition* resource file specified in the *New*

*Project* menu. If you want to change the sequence in the display then change the sequence of these definitions in the resource file and recreate the database.

## 5 The Lists menu

The formal representation of DUG consists of labeled dependency trees. The format for input and output of such trees consists of bracketed expressions containing attributes and values. We call this format "Dependency Representation Language" (DRL). The bracketed expressions are implemented as linked lists. That is why we use the term "list" synonymous to dependency tree.

Pressing the *Lists* button of the main menu bar results in Fig. 8.



Fig. 8 The *Lists* drop-out menu

### 5.1 Enter a List

This function is used as a tool for testing lists. Lists can be complex. The linear specification may contain errors. The *Enter a List* function checks whether a DRL input is correct and can be stored as a linked list. The function outputs the list in pretty print. Compare Fig. 9 and Fig. 10.

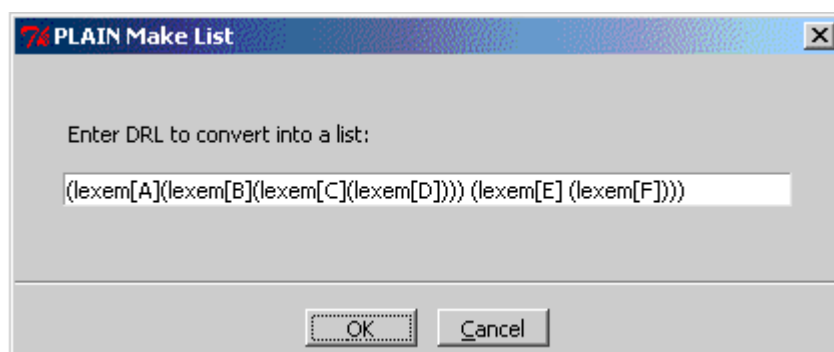


Fig. 9 Testing a DRL expression

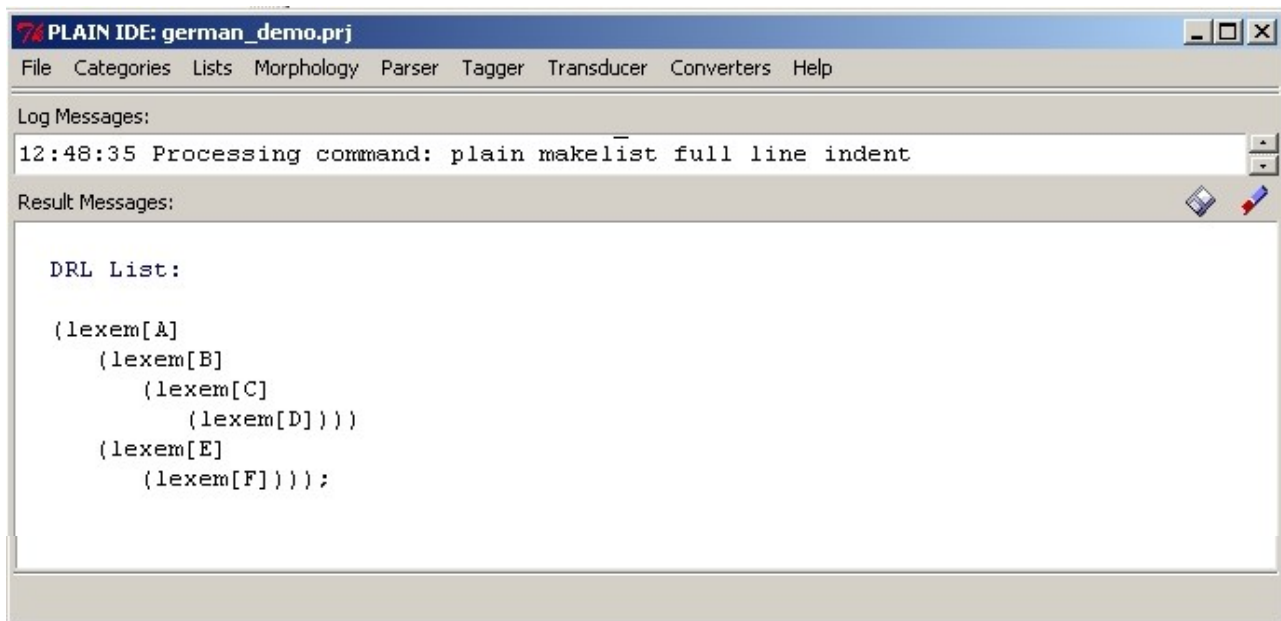


Fig. 10 The DRL expression in pretty print

## 5.2 Read and Write Lists

Many PLAIN functions create DRL lists and store them in files. The *Read and Write Lists* function allows to read such files in order to inspect the lists or to copy the lists to new files in various formats.

The *Read and Write Lists* command yields the following pop-up window.



Fig. 11 The *Read and Write Lists* window

The lists are read from the file specified in the *File* box. If *Copy to file* is activated then the lists are written into a new file. You will be prompted for the name and directory of the new file. If *Selected attributes* is activated then only those categories are displayed and copied that are visible according to the actual state of *Category Filtering* (cf. 4.2): If *Line format* is activated then the output is divided in lines with a length of 80 characters. If *XML tags* is activated then each list is augmented by the prefix "<instance><drl>" and the postfix "</drl></instance>\n". If *Roles & lexemes only* is activated then just the role attribute and the lexeme and reading attributes are visible. The role attribute displays the syntactic function of a list element, like "subject, object, predicate" etc. The lexeme and reading attributes show the lexical meaning of the element. This list format is most perspicuous. If *Pretty print* is activated then the list depicts the tree structure by means of indenting. Each list element is displayed on a new line and indented according to the bracketing structure.

### 5.3 Find Lists in the Database

Lists are widely used by the PLAIN modules. That is why an efficient device for storing and retrieving lists in the database is implemented in PLAIN. *Find Lists in the Database* offers these facilities to the user.

The *Find Lists in the Database* command results in the following pop-up window.



Fig. 12 The *Find List* window

There are three devices for optimizing the access to tree-like data. First, the database can be divided into arbitrary *partitions*. For example, the data of different languages may be saved in different partitions. The synframes and templates of the parser certainly form particular partitions as well as the rules used by the transducer.

Second, one or more indexes are created for each list stored in the database. Each index consists of an attribute and a value. For example, *Attribute* "role", *Value* "predicate", or *Attribute* "lexeme", *Value* "love".

Third, it can be stipulated at which *Location* in the tree the indexing attribute and value occur. The following possibilities exist:

- *Head*: the attribute occurs in the top-most term of the list.
- *Ante*: the attribute occurs in the antecedent term, i.e. the first child of the topmost term of the list.
- *Post*: the attribute occurs in the postcedent term, i.e. the second or any further child of the topmost term of the list.
- *Depend*: the attribute occurs in any child of the top-most term (this includes *Ante* as well as *Post*).

Partitions and the types of indexes for each partition must be defined by the linguist in a special file. This file must be specified in the *Definition File* box of the *New Project* pop-up menu. The format of this file is described in "The Linguist's Guide to PLAIN".

In order to retrieve a list or a set of lists enter or choose a partition in the *Partition* Box of Fig. 12. Enter an *Attribute* and a *Value* or use the asterisk as wildcard. Choose the *Location* of the index attribute in the list. Press OK.

Note: Partitioning and Indexing may be defined in such a way that every list is indexed according to many attributes, values and locations. However, the *Find List* function only allows retrieving a list by one *Attribute*, *Value* and *Location* at a time.



## 6 The Morphology menu

Pressing the *Morphology* button of the main menu bar results in Fig. 13.

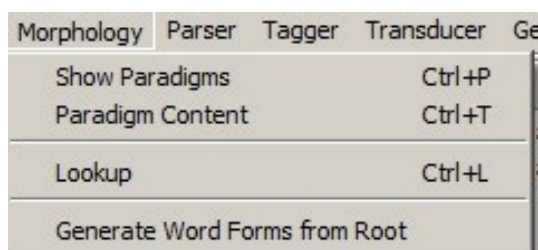


Fig. 13 The *Morphology* drop-out menu

### 6.1 Show Paradigms

The morphological lexicon is divided in so-called *Paradigms*, e.g. a set of stems or a set of endings of a particular inflectional class etc. Each paradigm has an *id* (a name). See "The Linguist's Guide to PLAIN" for advice to draw up a lexicon of paradigms.

The command *Show Paradigms* displays the ids of all paradigms stored in the database. This function is useful, first of all, for checking whether the definition of paradigms succeeded and is correct.

### 6.2 Paradigm Content

You will be prompted for a paradigm name if you issue the *Paradigm Content* command. A copy of the specification of the paradigm in question is displayed. This copy is drawn up from the internal database. It should be identical with the specification of the paradigm in the external resource file. Hence, the function serves for assessing the correct contents of the morpho-syntactic resources and the successful storage in the project's database.

### 6.3 Lookup

The *Lookup* function retrieves the morpho-syntactic classification assigned to a given natural language string by the actual lexicon. The *Lookup* command results in the following pop-up window.

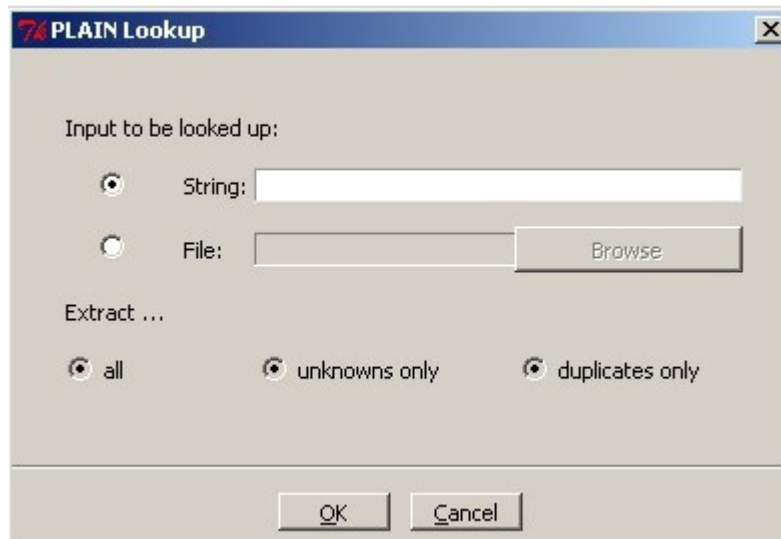


Fig. 14 The *Lookup* window

The input to be looked up in the lexicon must be entered in the *String* box of the pop-up menu. Alternatively the input can be provided by a file which contains a list of strings or a full text. The file must be specified in the *File* box of the menu.

There are three options that restrict the information displayed by the *Lookup* function. If *All* is activated then all strings in the input are classified and all attributes and values are shown. The verbosity of the classification can be tuned by *Category Filtering* (cf. 4.2).

If *Unknowns only* is activated then only those strings in the input are displayed that are NOT in the lexicon and hence are classified as "unknown". This option is useful for checking the completeness of the lexicon. If applied to a text corpus, the result is a list of words that must still be included in the lexicon.

If *Duplicates only* is activated then only those strings and attributes are displayed which yield the same classification several times. This can happen because the subsets of the lexicon in the resource files overlap. Duplicates also reveal redundancy due to inappropriate assignments. Hence, this option is useful for checking the conciseness of the lexicon.

## 6.4 Generate Word Forms from Root

The *Generate Word Forms from Root* command in the *Morphology* menu initiates a function that generates the inflected strings together with their morpho-syntactic description given a root and a paradigm. The pop-up menu invoked by the command looks as follows:

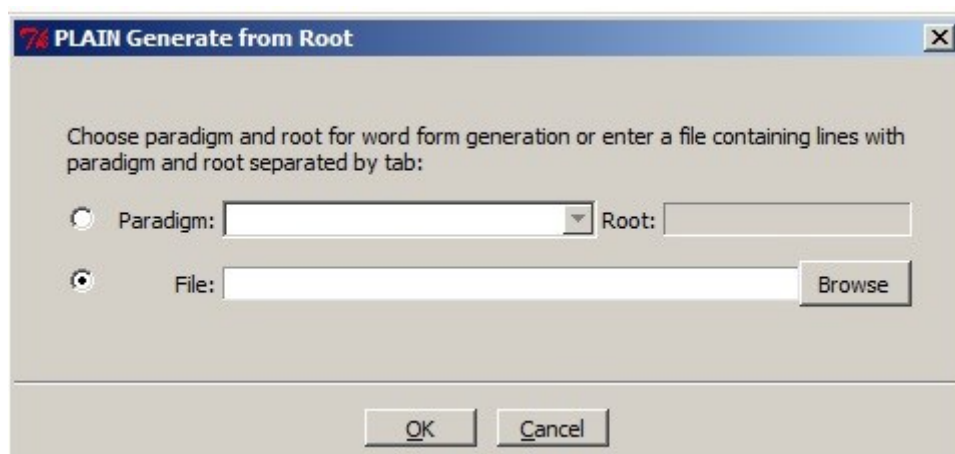


Fig. 15 The *Generate Word Forms* window

The morphological lexicon is technically a Finite Transition Network (FTN). The most simple case of contents is a set of word stems stored in one paradigm. Each word stem leads to a particular set of endings stored in other paradigms that combine with the stem in question. By traversing the FTN via the stem subnet to the endings subnets all word forms can be derived. This is in fact what the function *Generate Word Forms from Root* is doing. For details on a more sophisticated elaboration of the lexicon see "The Linguist's Guide to PLAIN".

You must specify the paradigm which contains the stem (or root) in the *Paradigm* box and you must enter the stem itself in the *Root* box. If you want to test a

large amount of word forms you might draw up a text file in your editor. Each line in this file must contain a paradigm id and a root separated by the tab character.

Note: The usefulness of this function is limited. It generates just those word forms that are hooked to a particular stem. If you want to check exactly this, fine. Using this function for generating all forms of a word with changing stems would mean that you have to enter all the stems. PLAIN offers a far more comprehensive function of word form generation: *Generate Word Forms from Lexeme*. See section 10.1.

## 6.5 Morphology layers and converters

DUG offers a sophisticated treatment of morphology which is mirrored by several layers of descriptions. The detailed specification of these descriptions is presented in "The Linguist's Guide to PLAIN". Tools to turn one layer of description into another are displayed if you press the *Converters* button of the main menu bar (cf. chapter 11).

## 7 The Parser menu

Pressing the *Parser* button of the main menu bar results in Fig. 16.

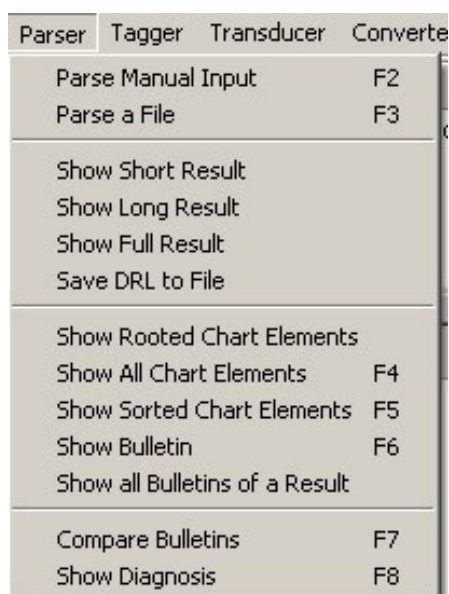


Fig. 16 The *Parser* drop-out menu

## 7.1 Parse Manual Input

The *Parse Manual Input* command is used for testing the parser on-line. The command invokes the following window:

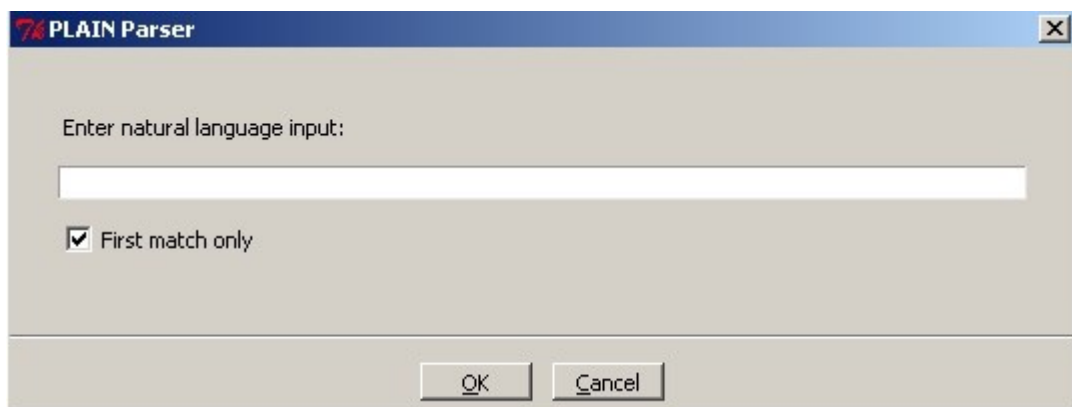


Fig. 17 Manual input to the parser

Any text to be analyzed may be entered in the input box. The output of the parser is a DRL list which will appear in the *Result Message* box of the Main Window. You find details on the analysis and the necessary resource files in "The Linguist's Guide to PLAIN".

If *First match only* is activated then the parser stops after reaching one coherent and grammatical analysis of the whole input. If you turn off this option then any possible analysis is tried. This may result in excessive processing time. On the other hand it is necessary to turn off the option in order to detect existing ambiguities.

## 7.2 Parse a file

The *Parse a file* command yields the following pop-up window:



Fig. 17 Input to parsing from a file

In this case the parser receives its input from the file specified in the *File* box. The output of the parser are DRL lists which will appear in the *Result Message* box of the Main Window. At present, the unit in the file to be parsed are lines. If you want to parse sentences you have to separate the sentences by the line feed character.

There are three options regarding the way each resulting list is displayed. If more then one option is chosen then more then one result is displayed.

If *Short* is activated then the result is printed as a tree with each element on a separate line and indented according to bracketing. Just the role attribute and the lexeme and reading attributes are visible. The role attribute displays the syntactic function of the list element, like "subject, object, predicate" etc. The lexeme and reading attributes show the lexical meaning of the element. This list format is most perspicuous.

If *Long* is activated all attributes and values of each list element are displayed. If you activated *Category Filtering* (4.2) before then only the visible attributes are shown.

If *Full* is activated then all available Information about the resulting list is displayed, including the corresponding surface string of each list element and its offsets in the input.

Note: The input file may contain comments which must adhere to the following format: `<test topic="x"/>`, where x is any text. The comment is displayed in the parsing output in front of the examples that followed the comment in the input file.

If you use the converter *Text Examples to Parser Input* to turn examples of templates and synframes into a test file (cf. 11.5) such comments are inserted automatically.

### **7.3 Tools**

The rest of the commands of the Parser menu in Fig. 16 are tools for the linguist as a user of the PLAIN IDE. Drawing up a lexicon and a grammar of a language is a complicated task. It requires extensive testing and debugging. Facilitating this work is the focus of PLAIN.

Some of the tools require knowledge about certain aspects of the PLAIN algorithms. Please refer to "The PLAIN Technical Guide".

The following commands require the previous execution of the *Parse Manual Input* command. As long as *Parse Manual Input* command is not issued again any of the following commands can be used in order to display aspects of the previous analysis.

### **7.4 Show Short Result**

The parsing result is displayed (again) in pretty print, but in a special format restricted to role, lexeme and reading attributes. The role attribute displays the syntactic function of the list element, like "subject, object, predicate" etc. The lexeme and reading attributes show the lexical meaning of the element. This list format is most perspicuous.

## 7.5 Show Long Result

The parsing result is displayed (again) as a DRL list in pretty print, including all visible attributes and values of each list element. The *Category Filtering* (4.2) command may be executed at any time. Subsequently the *Show Long Result* command yields only those attributes visible in the Category Filtering box (Fig. 7).

## 7.6 Show Full Result

The parsing result is displayed (again). For each list element the corresponding surface string and its offsets in the input is shown followed by the complete DRL classifying of this element. The list elements on the same hierarchical level are sorted according to the sequence of words on the surface.

## 7.7 Show Rooted Chart Elements

The *chart* is a central data structure of the parser. Its purpose is to administrate intermediate results. For details see "The PLAIN Technical Guide". The linguist can learn from the intermediate results whether the analysis meets the expectations. *Show Rooted Chart Elements* displays only those chart elements that form intermediate results in the course of building the result with the "root" (i.e. the top-most element of a dependency tree). You will be prompted for the number of the root element. If you enter "0" you will get the chart elements of the final result.

## 7.8 Show All Chart Elements

*Show All Chart Elements* displays all chart elements in the sequence of their emergence.



## **7.9 Show Sorted Chart Elements**

*Show Sorted Chart Elements* displays all chart elements sorted according to the length of the intermediate result they represent, the longest first.

## **7.10 Show Bulletin**

A bulletin is an object that collects all information about a chart element. *Show Bulletin* displays this information. Making use of this information requires the study of "The PLAIN Technical Guide".

## **7.11 Show all Bulletins of a Result**

This is a combination of *Show Rooted Chart Elements* and *Show Bulletin*. All the bulletins are displayed that form intermediate results in the course of building the "root" bulletin (i.e. the one containing the top-most element of a dependency tree). You will be prompted for the chart number of the root element. If you enter "0" then you will get all intermediate bulletins of the final result.

## **7.12 Compare Bulletins**

You will be prompted for the number of two chart elements. *Compare Bulletins* displays the similarities and differences of the two bulletins corresponding with the chart elements. This tool is useful if some intermediate results look alike. Identical intermediate results must be avoided because they lead to a useless combinatorial expansion.

## **7.13 Show Diagnosis**

You will be prompted for the number of two chart elements. *Show Diagnosis* resets the parser to the situation of processing these two elements. The parser tries anew to combine the corresponding constituents in order to form a new constituent covering both. This is the basic mechanism of the parser. All possibilities according to the bulletin data are tried, e.g. interpreting one

constituent as head and the other one as complement, or one as head the other one as adjunct etc. Failing and succeeding steps are documented by extensive comments.

This tool is most useful if the linguist is puzzled why the parser does not work as expected for a particular syntactic construction. It is also helpful to understand why useless results are created.

## 8 The Tagger menu

A tagger is a program that usually associates simple part-of-speech symbols with words, often by means of statistical methods. The lexicon *Lookup* module of PLAIN yields much more sophisticated classifications of words with all kinds of attributes and values. Lemmatization as well as segmentation of compounds is included. Since categories can be defined and filtered at will by the user, the classification can focus on particular features including semantic ones. Last but not least, the tagger can invoke the parser module. In this case tags are more reliable because they are disambiguated by the syntactic context. In addition syntactic role tags are created which should be more useful than parts-of-speech tags. *Category Filtering* (4.2) may be applied previous to invoking the tagger and thus provide a large variance of tag sets.

Some users may be interested in these capabilities, although this is only a by-product of a comprehensive description of languages which is the main focus of PLAIN.

Pressing the *Tagger* button of the main menu bar results in Fig. 18.

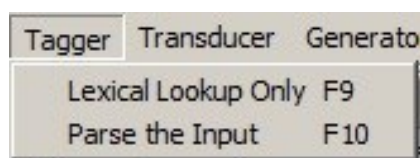


Fig.18 The *Tagger* drop-out menu

## 8.1 Lexical Lookup Only

In this case the tagging task is performed by the lexicon *Lookup* module alone. The device is the same as the *Lookup* command in the *Morphology* menu. If you issue the *Lexical Lookup Only* command you will be prompted for an input file which contains the text to be tagged.

An output xml-file is created containing the words (more precisely: the lexical segments) of the input, each one on a separate line and surrounded by `<char>` and `</char>`. The `<char>` element is augmented by an attribute named "offset". Its value is the offset of the word from the beginning of the input text. The `<char>` element has another attribute named "length". Its value is the length of the word. A number of classifications in DRL notation follow, each one on a separate line and surrounded by `<drl>` and `</drl>`. The `<drl>` element is augmented by an attribute named "valid". Its value is "1" if the classification is ambiguous, its value is "4" if a unique classification has been found.

If the check box *First match only* is activated then only the first `<char>` `<drl>` pair of alternative classifications is displayed, i.e. for any segment there is just one classification.

Note: You can reduce ambiguity and hence alternative classifications by moving attributes from the visible to the hidden part with the command *Attribute filtering* in the menu *Categories*.

## 8.2 Parse the Input

In this case full or partial parsing is employed in order to create reliable or more perspicuous tags. If you issue the *Parse the Input* command you will be prompted for an input file which contains the text to be tagged.

An output xml-file is created containing the sentences of the input file surrounded by `<instance>` and `</instance>`. At present, each character string followed by a line feed character is treated as sentence.

Within the `<instance>` element a number of pairs follow. Each pair is formed by a word (more precisely: a lexical segment) of the input on a separate line and surrounded by `<char>` and `</char>` and a classification in DRL notation on a separate line and surrounded by `<drl>` and `</drl>`. The `<char>` element is augmented by an attribute named "offset". Its value is the offset of the word from the beginning of the sentence (i.e. the string covered by the `<instance>` tags). The `<char>` element has another attribute named "length". Its value is the length of the word.

Note: In the case of *Lexical lookup only* "offset" is calculated from the beginning of the input file. In the case of *Parse the Input* "offset" is calculated within each `<instance>` element.

The `<drl>` element is augmented by an attribute named "valid". Its value is "2" if the classification is based on partial parsing because no complete syntactic analysis of the input has been achieved. The value is "3" if the parser yielded more than one complete syntactic analysis, i.e. the sentence is ambiguous and hence the words classification not completely safe. The value is "4" if the tagging is based on just one complete parser result.

The words in the output are displayed in the same order as in the input. Three additional attributes of the `<drl>` tag allow to reconstruct the syntactic structure: "root", "son" and "brother". The attribute 'root="yes"' marks the head of the dependency tree representing the sentence. The value of the attribute "son" is the "offset" attribute of the first element that is dependent of the element in question. The value of the attribute "brother" is the "offset" attribute of another element that is dependent of the same head as the element in question.

## **9 The Transducer menu**

The transducer creates new DRL representations (e.g. a translation into another language) on the basis of old ones. The transformation is guided by rules. For more details see "The Linguist's Guide to PLAIN".

Pressing the *Transducer* button of the main menu bar results in Fig. 19.

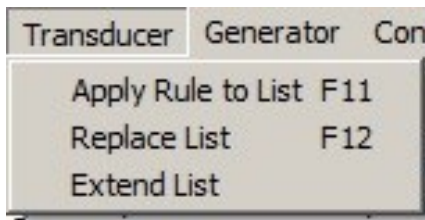


Fig. 19 The *Transducer* drop-out menu

### 9.1 Apply Rule to List

The command *Apply Rule to List* causes the following window to appear:

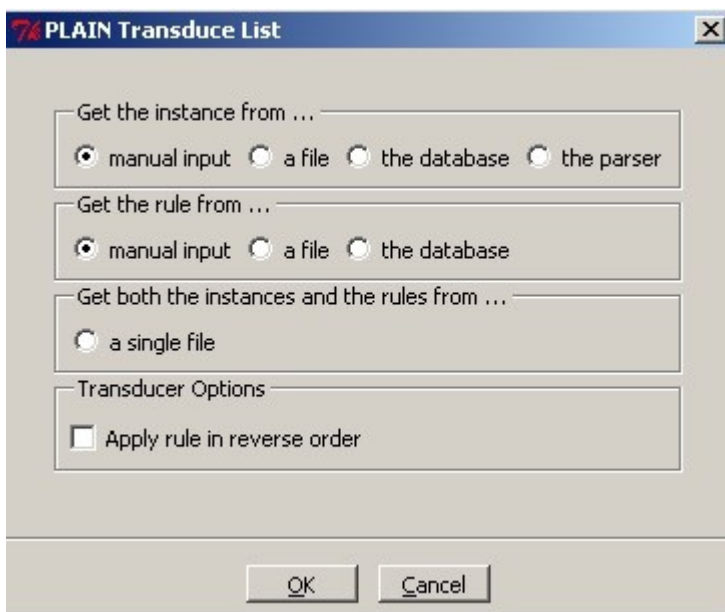


Fig. 20 The *Apply Rule to List* window

The purpose of the *Apply Rule to List* function is testing and debugging individual transducer rules. Any DRL list may function as an *instance*. A *rule* consists of two parts: a pattern describing the applicable instances, and a pattern of the result to be created.

First we have to get the instance that is to be the subject of a rule application. If *manual input* is activated then an entry box pops up. A DRL expression representing the instance has to be entered in this box.

If getting the instance from *a file* is activated then an entry box pops up. A file containing the instance must be specified in this box.

If getting the instance from *the database* is activated then a pop-up menu appears identical to the *Find List* menu in Fig. 12. In order to retrieve the instance list choose a partition in the *Partition Box*., enter an *Attribute* and a *Value* or use the asterisk as wildcard. Choose the *Location* of the index attribute in the list. Press OK. The first instance meeting the criteria is shown and the following window pops up:

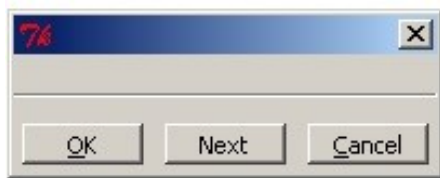


Fig. 21 Choose an instance or continue

If you want to use the displayed instance then press *OK*. If you rather want to retrieve the next instance meeting the criteria then enter *Next*.

If instances are to be retrieved then they must have been stored in the database before. The corresponding file or files must have been made official parts of the actual project. This is achieved by specifying the instance files in the *Instances and Rules* box of the *NewProject* menu, the *Project Settings* menu or the *Augment Project* menu. The *Partitioning and Indexing* method for these files must be included in the file specified in the *Definition File* box.

If getting the instance from *the parser* is activated then the latest parsing result is made the instance. Of course, the parser must have been invoked previously.

Second we have to get a rule. If *manual input* is activated then an entry box pops up. A DRL expression representing the rule has to be entered in this box.

If getting the rule from *a file* is activated then an entry box pops up. A file containing the rule must be specified in this box.

If getting the rule from *the database* is activated then a pop-up menu appears identical the *Find List* menu in Fig. 12. In order to retrieve the rule choose a partition in the *Partition Box.*, enter an *Attribute* and a *Value* or use the asterisk as wildcard. Choose the *Location* of the index attribute in the list. Press OK. A window pops up like in Fig. 21 so that you can choose the current rule or to retrieve another one.

If rules are to be retrieved then they must have been stored in the database before. The corresponding file or files must have been made official parts of the actual project. This is achieved by specifying the rule files in the *Instances and Rules* box of the *NewProject* menu, the *Project Settings* menu or the *Augment Project* menu. The *Partitioning and Indexing* method for these files must be included in the file specified in the *Definition File* box.

If large test batteries are to be run it is convenient to associate pairs of instance and rule in the same file. For this alternative you have to activate *Get both the instances and rules from a single file*. You will be prompted for the file name.

If replacement rules are symmetrical then they can be applied in both directions. For example the same rule can be used for and English-to-German translation and a German-to-English translation. Activate the option *Apply rule in reverse order* if desired.

As opposed to the command *Replace List* below, *Apply Rule to List* applies a single rule to a single instance.

## 9.2 Replace List

This function repeatedly applies rules to all parts of an instance, changing the instance until no further replacement is possible. The command *Replace List* causes the following window to appear:

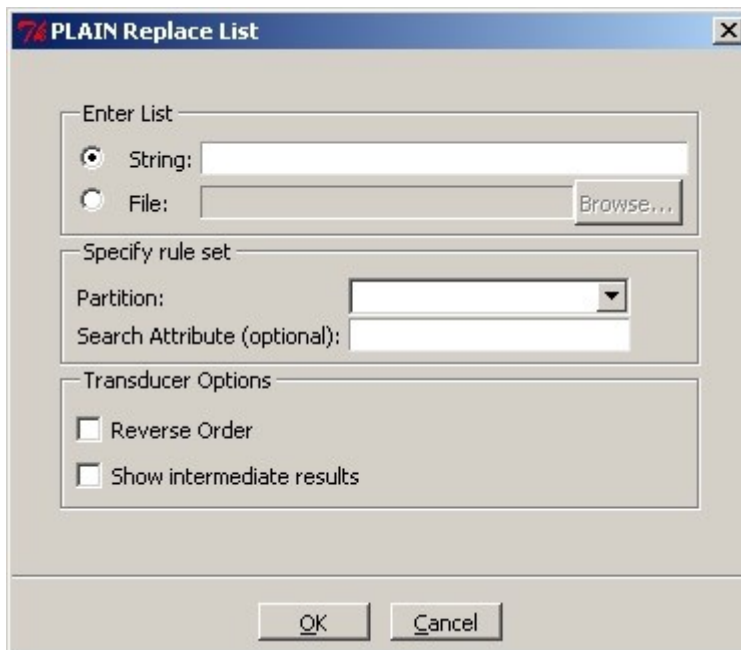


Fig. 22 The *Replace List* window

First, we need lists to be replaced. One can enter a DRL list manually in the *String* box or specify a file containing a collection of DRL lists in the *File* box.

Second, we need to *specify a rule set*, e.g. all the rules necessary to translate the instance into another language. The rules to be used must be part of the database and they must have been declared in the *Instances and Rules* box of the project. Rules must be stored in a particular partition. The appropriate partition must be chosen in the *Partition* scroll box of Fig.22.

The algorithm of *Replace Lists* proceeds from the smallest to the largest sublists in the instance. The smallest sublists are the leaves in the dependency tree. If a replacement is possible then a new list is formed in which the corresponding substitution has taken place. The process starts over again, looking up the leaves in the new tree first. If no replacement is possible, the next higher sublist is tried. In this way replacement proceeds bottom-up until no further substitution is possible.

The efficiency of the replacement function depends on a constrained search of rules. The principle of lexicalized rules agrees with the lexicalized approach of



the dependency grammar. Therefore it is recommended to draw up replacement rules for lexemes and specify the lexeme attribute in the *Search Attribute* box.

Activating the *Reverse Order* check box has the effect that the rules are applied from right to left while the normal direction is from left to right.

Activating *Show intermediate results* displays each rule that is used and each state of replacement.

## 10 Generator

The generator is the reverse process of the parser. While the parser turns strings into DRL representations, the generator turns DRL representations into natural language surface strings. The generator uses exactly the same resources as the parser, i.e. the morphosyntactic lexicon, the set of synframes and templates.

Of course other processes may intervene between parser and generator. For example, the parser creates a dependency tree, the transducer transfers this tree into a DRL representation of another language, and the generator creates the string that corresponds to the latter tree. Or the parser analyzes a full text, the transducer derives a summary, the generator outputs the text of the summary.

Pressing the *Generator* button of the main menu bar results in Fig. 23.

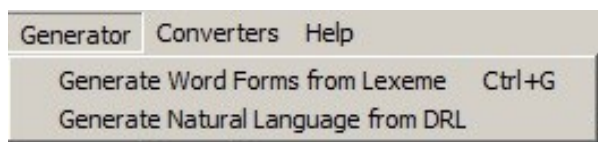


Fig. 23 The *Generators* menu

### 10.1 Generate Word Forms from Lexeme

This is the morpho-syntactic subtask of generation. Any word form categorized by a particular lexeme is created together with its attributes. The command *Generate Word Forms from Lexeme* causes the following window to appear:

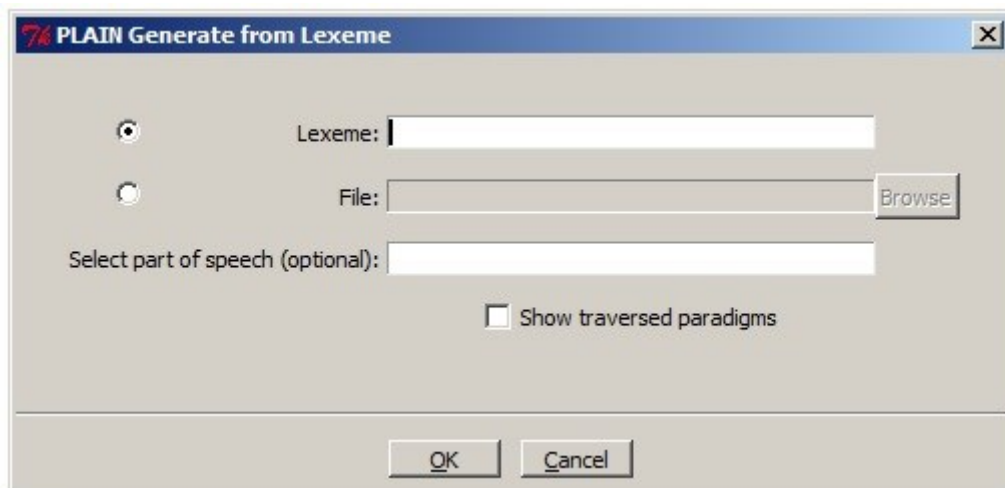


Fig. 24 The *Generate from Lexeme* window

The lexeme in question has to be entered in the *Lexeme* box. Alternatively one can draw up a file with a list of lexemes (each on a separate line). If you have a complete list of lexemes of your implementation then you can derive a word forms lexicon with this function, i.e. the set of all existing word forms together with their categorization.

Word forms may include derivations, e.g. the noun *lover* or the adjective *loving* from the verb *love*. If you want to restrict the generation to a particular part of speech, e.g. just the nouns derived from the verbal lexeme then enter the desired category in the *Select part of speech* box.

Remember that the morphological lexicon is technically a Finite Transition Network (FTN). If the lexicon is structured by complex transitions between paradigms it might be difficult to check the correctness and debug errors. If the check box *Show traversed paradigms* is activated then the transitions that lead to the particular word form are displayed.

## 10.2 Generate Natural Language from DRL

This function is the full fledged PLAIN generator based on the available morphological and syntactic description. If the output of the parser is passed as input to the generator than the output of the generator should be the original

input of the parser. (If it is not then the resources are insufficient. They do not define, in a formal sense, exactly the grammatical strings of the language.)

The command *Generate Natural Language from DRL* causes the following window to appear:

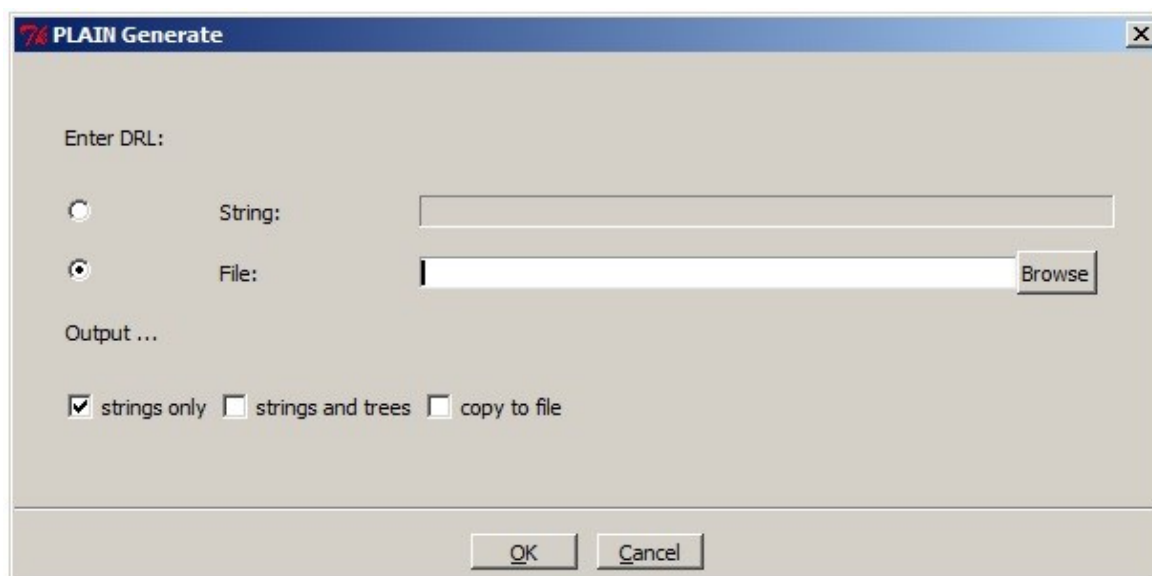


Fig. 25 The *Generate Natural Language from DRL* menu

A DRL expression can be entered in the *String* box. Usually such expressions are created by other functions and saved in a file. In this case the file must be specified in the *File* box. The DRL expression must consist of a dependency tree that is a valid analysis of a natural language string according to the associated lexicon, synframes and templates. The tree may represent a sentence or any substring of a sentence including a single word. (In the latter case the tree consists of a single term.) As compared to the parsing output the DRL input to the generator may be underspecified though.

The minimum information each term must comprise is a role attribute and a lexeme attribute. Syntagmatic roles and lexemes are considered to be the basic knowledge representation of the DUG. The generator augments these underspecified expressions with the compatible attributes from the grammatical resources.

The output of the function can be limited to the generated *strings only*, or it can include *strings and trees*. In the latter case the trees are shown which have been augmented with the unified attributes which the generator retrieved from *templates* and *synframes*.

You may save the output by activating the *copy to file* box.

## 11 Converters

The *Converters* menu gathers a few tools for handling variations of input files.

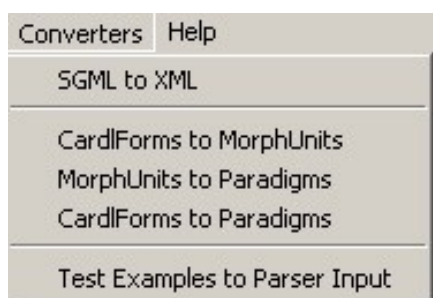


Fig. 26 The *Converters* menu

### 11.1 SGML to XML

In the course of developing PLAIN some legacy data has been drawn up according to the DTD "plain-sgml.dtd". At present the program accepts only data according to the DTD "plain-xml.dtd". A wizard leads you through the conversion from sgml-files into xml-files. The osx utility is used for this purpose.

### 11.2 CardlForms to MorphUnits

The resources for the morphological lexicon are organized in three "layers". There are reasons for different layers from a theoretical point of view. The theoretical background and the syntax of the necessary files are explained comprehensively in "The Linguist's Guide to PLAIN".

Cardinal forms (*CardlForms*) are an easy way of drawing up morphological classifications. The morphological units (*Morphunits*) provide an explicit

classification of lexical items. They have been introduced mainly as an interface for data sharing with the world outside of PLAIN and the DUG. Patterns of cardinal forms (*CardIPats*) describe the possible cardinal forms and associate each matching form with a morphological class.

The following pop-up window initializes the converter *CardIForms to MorphUnits*:

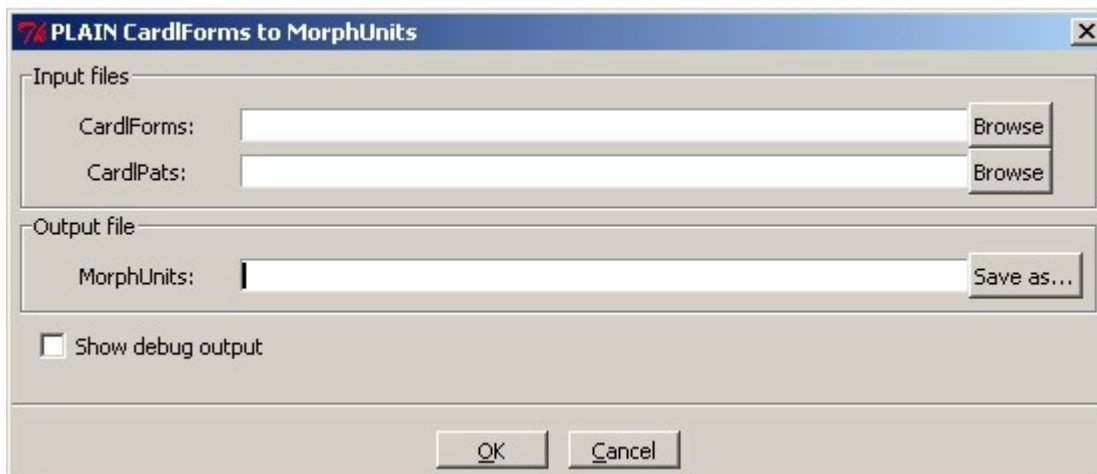


Fig. 27 *CardIForms to MorphUnits* Converter

A file with cardinal forms must be entered in the *CardIForms* box. A file with patterns for these cardinal forms must be entered in the *CardIPats* box. The full path and name of a file must be entered which is to receive the output of *MorphUnits*.

### 11.3 MorphUnits to Paradigms

So-called *Paradigms* form the immediate input layer of PLAIN. They can be loaded into the database. Technically they are subnets of a finite state transition network. In order to convert *MorphUnits* into *Paradigms*, the converter must resolve the morphological classes that are specified in the morphological units. A set of paradigms corresponds to each class. This description of classes (*MorphClasses*) must be drawn up in a separate file.

The following pop-up window initializes the converter *MorphUnits to Paradigms*:

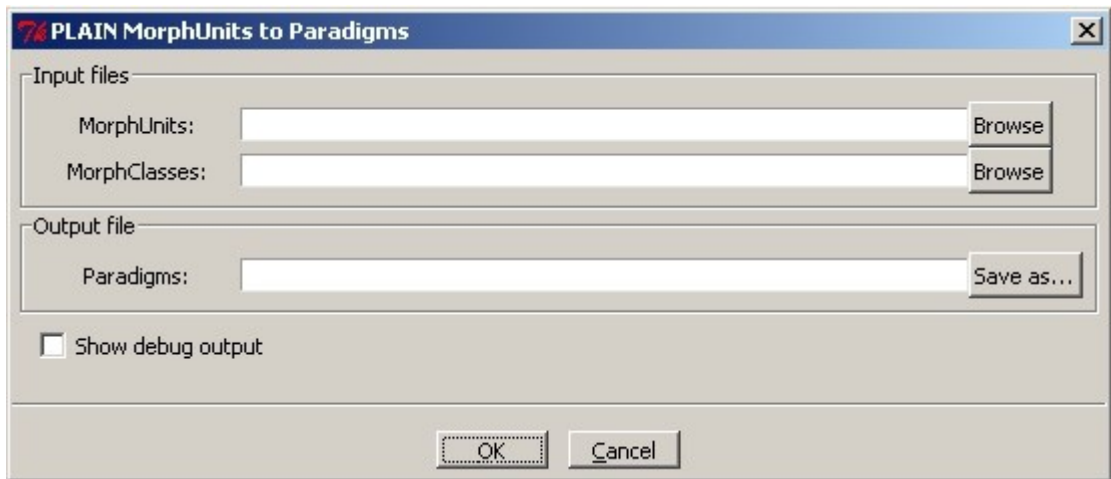


Fig. 28 *MorphUnits to Paradigms* converter

A file with morphological units must be entered in the *MorphUnits* box. A file with descriptions of morphological classes must be entered in the *MorphClasses* box. The full path and name of a file must be entered that is to receive the output of *Paradigms*.

#### 11.4 CardIForms to Paradigms

If the layer of morphological units (*MorphUnits*) is not needed then one can create directly *Paradigms* from *CardIForms*.

The following pop-up window initializes the converter *CardIForms to Paradigms*:

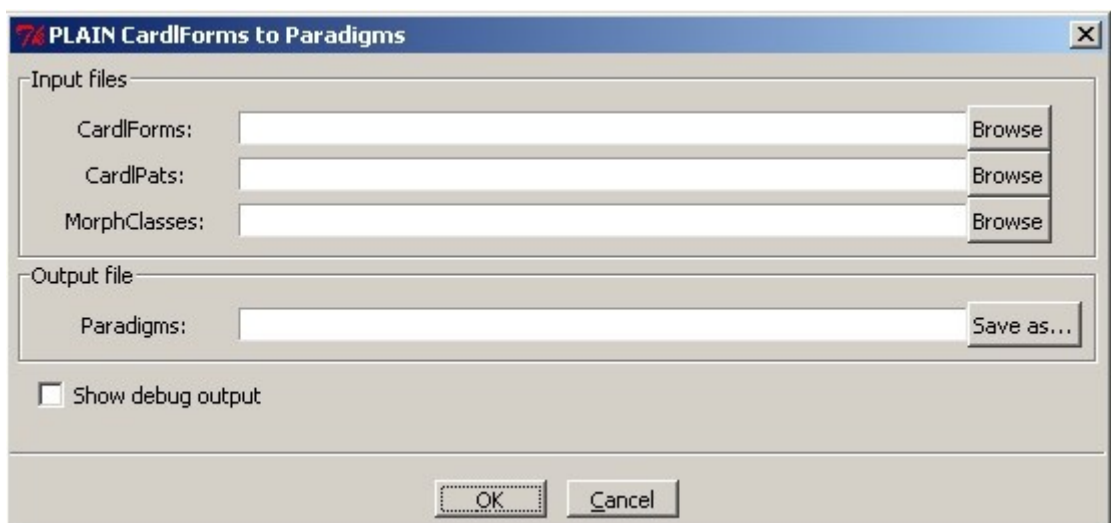


Fig. 29 *CardIForms to Paradigms* converter

A file with cardinal forms must be entered in the *CardIForms* box. A file with patterns for these cardinal forms must be entered in the *CardIPats* box. A file with descriptions of morphological classes must be entered in the *MorphClasses* box. The full path and name of a file must be entered that is to receive the output of *Paradigms*.

## 11.5 Test Examples to Parser Input

This is a facility to relate writing and testing resources. While writing the grammar it is possible to add test examples to any synframe and to any template. You have to use a test tag for this purpose. It has the following format:

```
<test topic="x">y</test>
```

x and y is any text. x is supposed to indicate the grammatical phenomenon in question, and y is an example. The converter creates a test file containing the examples y.

For example, a template for a dative complement in the *german\_demo* project might be associated with the following test tags:

```
<test topic="+dativ"> Der Mann /gibt/ (dem Kind) das Buch.</test>  
<test topic="+dativ"> Der Mann /gibt/ das Buch (dem Kind).</test>
```

The test tag includes an indication of the grammatical phenomenon covered by the template (topic="+dativ") The test example itself is structured in the following way:

- in brackets = the segment corresponding to the filler of the template,
- between slashes = the segment corresponding to the head of the template,
- anything else = context not covered by the template (described somewhere else).
- 

The resource file with test examples must be specified in the pop-up window:

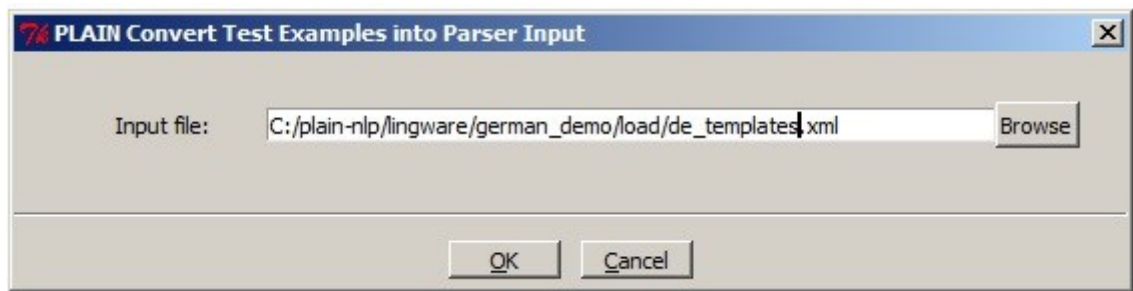


Fig. 30 Deriving test examples from the *templates* resource file.

You will be asked to specify an output file. The test examples are going to be stored in this file in a format that can be processed by the parser. Slashes and brackets are removed. The topic-tag is separated from the parser input. The example above results in the following lines:

```
<test topic="+dativ">
Der Mann gibt dem Kind das Buch.
</test>
<test topic="+dativ">
Der Mann gibt das Buch dem Kind.
</test>
```

This output serves as input for the parser. It can be processed with the *Parse a file* command (7.2). A nice feature is the fact that the topic tag is saved and printed out with the parsing result. Drawing up constructions and testing them can be organized in this way.