Dictionary Entry Parsing Using the LexParse System

Ralf Hauser*
Seminar für Sprachwissenschaft
der Universität Tübingen

Angelika Storrer†
Institut für deutsche Sprache
Mannheim

December 15, 1994

Abstract

Dictionary entry parsers are software systems which recognize and represent the hierarchical microstructure of dictionary entries as stored in typesetting tapes by consulting a user-defined grammar.

This article deals with the specific problems related to dictionary entry parsing. We will compile a list of general requirements which a dictionary entry parser has to meet in order to cope with these problems and then describe the LexParse system, a dictionary entry parser specifically designed to cope with these problems.

- The LexParse program supports an easy to learn yet powerful grammar formalism, providing operators and techniques for a straightforward handling of structure indicators, fonts and typefaces changes and sense/homograph counters.
- LexParse promotes rapid prototyping of dictionary entry grammars and offers comprehensive tools for improving and debugging grammars by and by.
- The flexible output component of LexParse displays the hierarchical microstructure of parsed entries in several user-definable styles, e.g., hierarchical attribute-value structures or annotated SGML structures.

The functionality, the design and the underlying concepts of the system will be described and illustrated by examples.

*E-mail: affie@sfs.nphil.uni-tuebingen.de
†E-mail: storrer@mx300e.ids-mannheim.de

Contents

1 Introduction

Dictionary entry parsing, i.e., the automatic conversion of a typesetting representation of dictionary text into a format which explicitly represents the hierarchical structure of dictionary entries, has become an increasingly important topic in Computational Lexicography. Whereas earlier programs were tailored to the conversion of one specific dictionary, the growing interest in machine readable dictionaries led to the development of general-purpose dictionary entry parsers, which may be used in converting any type of typesetting representation of dictionary text (cf. NEFF & BOGURAEV 1990 [16] and BLÄSI & KOCH 1991 [3]).

Dictionary entry parsers analyze an input stream coming from typesetting tapes using a dictionary entry grammar and then generate a representation of the dictionary text in which the hierarchical structure of the entries is explicitly shown so that each lexicographic item can be individually accessed by a software system. Such a representation of both dictionary form and dictionary subject is generally called *Lexical Database* and is used in different fields:

Computational Lexicography: Methods and techniques are developed and tested to transform the lexical information stored in lexical databases into a format which can be processed by a natural language processing (NLP) system (cf. BOGURAEV & BRISCOE 1989 [5]). Since the manual construction of NLP lexicons is a time and labor consuming task, one hopes that the automatic exploitation of machine readable dictionaries (MRDs) using these methods will reduce the effort required in the development of NLP lexicon components (cf. BOGURAEV 1991 [4]).

Traditional Lexicography: Lexical databases are fundamental components of lexicographic workbenches, i.e., software systems for computer-aided dictionary publishing. Dictionary entries stored in the database can be edited and updated directly thereby speeding up the process of revising existing dictionaries and making new dictionaries. Furthermore, the multiple ways in which lexical databases can be queried allow for a more consistent and reflected description of the lexical items in the dictionaries (cf. BLÄSER & WERMKE 1990 [2]).

Electronic editions of dictionaries based on lexical databases can easily be developed offering a faster and more flexible access to the dictionary text using techniques such as hierarchical, relational or object-oriented databases or hypertext systems.

Dictionary entry parsers automatically assign structure to the dictionary text represented on typesetting tapes using a dictionary entry grammar. Dictionary entry grammars define conditions for well-formedness of dictionary entries and specify partitive and precedence relations between the constituents of the dictionary entry structure. Dictionary entry structures not licenced by the dictionary entry grammar will be marked as non-wellformed. As a consequence, the process of dictionary entry parsing — aside from its main goal of converting the typesetting tape into a lexical database — has the side-effect of detecting errors and inconsistencies in the structural encoding of the dictionary. Thus, not only can complete and correct dictionary entry grammars be used in the conversion step, but they may be reused for checking structural wellformedness of revised entries in future editions of the dictionary in question as well. A prerequisite for this use of dictionary entry grammars is that the grammar formalism is simple and easy to learn, so that changes in the dictionary entry structure, introduced in connection with the revision of the dictionary, can be adapted without difficulty.

Our paper deals with the problems of dictionary entry parsing and presents the main features of the LEXPARSE system, a dictionary entry parser developed by Ralf Hauser within the framework of the *ELWIS* project.[1]

LEXPARSE supports a context-free grammar formalism supplemented by special operators which are needed for the particular requirements of dictionary entry parsing. It has been tested with parts of the DUDEN-Stilwörterbuch (DUDEN-2) and the DUDEN-Bedeutungswörterbuch (DUDEN-10).[2] and is currently being used for the parsing of the "Deutsches Rechtswörterbuch" (Akademie der Wissenschaften in Heidelberg/Germany) and the "Frühneuhochdeutsches Wörterbuch" (German language department of the University of Heidelberg) and will be applied to a bilingual dictionary in an EC funded R&D project. In most cases, our examples will refer to the dictionary entry grammar which was developed for the DUDEN-Stilwörterbuch (cf. ENGELKE 1994). Although the system was designed for the parsing of dictionary text, the system can also be used for the parsing of text types with similar structural properties such as bibliographies and encyclopedias.

The development of the system and of the above mentioned dictionary entry grammars was based on the theory on lexicographic texts by H.E. Wiegand. In the following section we will briefly introduce some basic notions of this theory needed in describing the key problems of dictionary entry parsing. Most of these problems can be traced back to the fact that intellectual segmentation of dictionary text into meaningful lexicographic text segments presupposes an understanding of its semantic content, whereas automatic segmentation can only rely on formal properties of the text segments. We will show that the problems of dictionary entry parsing differ considerably from those related to the parsing of natural languages and we will compile a list of specific requirements which a parsing system has to meet in order to cope with these problems. We will then present a detailed description of the design and of the general concepts of LEX-

[1] *ELWIS* is a research project on the corpus-based development of lexical knowledge bases (the acronym stands for *Korpusunterstützte Entwicklung lexikalischer Wissensbasen*) carried out at the University of Tübingen since 1992 and funded by the Ministry of Science and Research Baden-Württemberg (cf. STORRER, FELDWEG & HINRICHS 1993 [19]).

[2] We would like to thank Dr. Matthias Wermke and the DUDEN editorial for supplying us with this material.

Parse explaining how these concepts together with the grammar formalism lead to a powerful tool for dictionary entry parsing which will allow a straightforward treatment of these problems. Finally, we will describe the modular architecture of the system implemented in the object-oriented programming language C++ and outline how the modules of the system interact.

2 Dictionary Entry Parsing: Problems and Requirements

2.1 Dictionary Entry Parsing

A Dictionary entry parser is a software system which segments dictionary text stored on typesetting tapes into functional text segments and generates an explicit representation of the dictionary structure by consulting a user-supplied dictionary entry grammar. The development of dictionary entry grammars must be based on a theory which provides methods and categories for the analysis of the dictionary form and the dictionary subject. In recent metalexicographic research on printed dictionaries, the idea that dictionaries can be observed as a special text type to be described with textlinguistic categories has succeeded in being accepted. A formalized theory on lexicographic texts worked out by H.E. Wiegand provides:

- a sound theoretical framework for the development of dictionary entry grammars by formally describing the various types of text structures to be found in standardized dictionaries;
- methods to intellectually analyze and segment entries of standardized monolingual dictionaries;
- a substantial terminological framework to elaborate the problems of dictionary entry parsing as well as the different strategies for their solution.

In the following we will informally introduce those basic notions of the theory which are needed to understand the basic concepts and strategies of the LEXPARSE approach to dictionary entry parsing.[3]

2.2 Structural Properties of Dictionary Texts

The main part of a general monolingual dictionary is the word list, consisting of lexicographic descriptions for lemma signs, i.e., those lexical signs which are represented as **lemmata** (headwords) in the dictionary. The basic textual units of a word list are the **dictionary entries**. Dictionary entries can be further segmented into functional text segments which are either **structure indicators** carrying information on the dictionary form, or **lexicographic items** carrying information on the dictionary subject. The **microstructure** and the **macrostructure** are both order structures which are characteristic for the word list of a dictionary and have to be recognized by a dictionary entry parser.[4]

[3] Detailed and formal descriptions of different parts of the theory are given in different publications, e.g., in WIEGAND 1989a [20]; 1989b [21]; 1989c [22]; an overview is given in WIEGAND 1991 [24].

[4] Other types of structures described in the above mentioned theory on lexicographic texts, e.g., addressing structures, cohesion structures, theme-rheme structures, cross-reference struc-

2.2.1 The Macrostructure

The macrostructure defines the ordering of the lemmata in the word list by means of a specific **access alphabet**, i.e., an alphabet in which all kinds of diacritics are integrated. Strictly-alphabetically ordered dictionaries arrange their lemmata strictly according to this alphabet, whereby two kinds of arrangements have to be distinguished: **Niching** dictionaries cluster sub-entries, i.e., each sub-lemma immediately follows the preceding sub-entry without beginning a new line, whereas **straight-alphabetical** dictionaries have no lexicographic grouping. **Nested** dictionaries reveal the same type of grouping as niched ones, but are not strictly alphabetical, because the alphabetic order can be interrupted in order to exhibit morphosemantic relationships between the (sub)-lemmata.[5]

2.2.2 The Microstructure

The microstructure of a dictionary is determined by the classes of lexicographic items (**item classes**) used in the dictionary and specific relations which hold between them.[6] Two types of relations are relevant for the development of dictionary entry grammars to be used by a dictionary entry parser:

Partitive relations are defined on the set of classes of items and determine the partitive microstructure.

Precedence Relations are defined on the set of classes of items which cannot be further segmented into smaller functional text segments (**classes of elementary items**) thereby determining the precedential microstructure.

Both relations together determine the **hierarchical microstructure** of dictionary entries which can be represented by tree diagrams.[7] Classes of elementary items are the terminal nodes, classes of non-elementary items, i.e., complex categories such as example groups or lexicographic comments, are the non-terminal nodes of such a tree representation. Hierarchical microstructures can be described as a context-free dictionary grammar DG ⟨CEI,CNI,R,WA⟩ where:

CEI (the set of *C*lasses of *E*lementary *I*tems) is the terminal alphabet of DG;

CNI (the set of *C*lasses *N*on-elementary *I*tems) is the set of non-terminal symbols of DG;

tures have not yet been dealt with within the context of dictionary entry parsing.

[5] For a detailed description of the different kinds of access structures and macrostructures see WIEGAND 1989a [20].

[6] In our dicussion we use the term 'microstructure' to refer to the 'abstract microstructure' in the sense of WIEGAND 1989b [21] without further differenciating in abstract und concrete microstructures.

[7] A detailed description of different kinds of microstructures in general monolingual dictionaries can be found in WIEGAND 1989b [21]; 1989c [22].

R (*R*ules) is a set of context-free rewrite rules;

WA (*W*örterbuch*a*rtikel = dictionary entry) is an element of CNI and the initial symbol of DG.

2.2.3 Structure indicators

Structure indicators (SIs) carry information about the form of the dictionary and have the genuine function of supporting the user's perception of the macrostructure and the microstructure. Two types of SIs must be distinguished:

- Symbols like punctuation marks, brackets or other special characters are used as **nontypographic structure indicators** (SI_{ntyp}) which give structural information, e.g., separate lexicographic items from each other. Along with the lexicographic items, they constitute the main types of lexicographic text segments.

- The fonts and typefaces in which text segments appear in the dictionary have the function of **typographic structure indicators** (SI_{typ}). With respect to the printed dictionary, $SI\text{s}_{typ}$ are not segments of the dictionary text, but rather attributes of segments such as items and $SI\text{s}_{ntyp}$. They belong — from a semiotic point of view — to a secondary system of signs, which can only fulfill their sign function with the help of a primary system of signs.[8] This, however, does not hold true for the way in which the dictionary text is represented on typesetting tapes: In this representation, typographic information is encoded directly in the form of control codes denoting different kinds of fonts and typefaces.

Together $SI\text{s}_{typ}$ and $SI\text{s}_{ntyp}$ both help human users to reconstruct the textual structure of a dictionary. An elaborate system of SIs allows for rapid access to the desired information and can therefore considerably speed up the process of dictionary look-up. SIs play a central role in dictionary entry parsing because the automatic segmentation and analyses of dictionary text is mainly controlled by the interpretation of SIs. Whereas human users of dictionaries can rely on their linguistic and metalexicographic knowledge, dictionary entry parsers have to cope without this knowledge. The quality of the parsing results are substantially impaired if the system of SIs is not carefully considered or if it is used in an inconsistent way.

2.3 Dictionary Entry Parsing vs. Sentence Parsing

On a very high level of abstraction, the parsing of syntactic structures of natural language sentences and the parsing of dictionary entry structures are similar

[8]Cf. WIEGAND 1991:26f [24].

tasks, namely the automatical assigning of non-linear structure to a linear input stream of text segments by consulting a grammar. A closer look, however, reveals significant differences between the problems related to sentence parsing and those related to dictionary entry parsing:[9]

- A sentence parser has to cope with recursion, whereas a dictionary entry parser has to adequately handle the iteration of an arbitrary number of items belonging to the same item class, all being direct constituents of the same item group (e.g. a number of example sentences all of which are direct constituents of an example group item or a number of translations belonging to the same translation group). Recursion is only needed for dictionaries with niched or nested entries and only in those cases where the structure of the sub-entries is identical to those of the main entry.
- A sentence parser generally consults a lexicon in order to assign the words of the sentence to syntactic categories. A dictionary entry parser, however, when classifying the lexicographic text segments as belonging to specific item classes, has to manage without a lexicon: It is not feasible to list all items of unrestricted item classes, like example or definition sentences in a lexicon, since this would mean a listing of an infinite number of phrases and sentences which can, in priniciple, be used in order to describe or exemplify the meaning of the lemmata in the dictionary.[10] Lists of permissible values can only be specified for 'closed' item classes (e.g. items giving the gender of a noun or items giving information about inflection) — the parser can then test a lexicographic text segment in question against these lists and assign it to the appropriate item class.
- An important issue of sentence parsing is the recognition and resolution of various types of structural and semantic ambiguities. These phenomena do not have a counterpart in dictionary entry parsing where the hierarchical microstructure is unambiguously specified. However, dictionary entry parsing has to cope with two types of functional ambiguities both related to *SI*s, i.e., those text segments which play a crucial role for the identification and interpretation of the dictionary entry structure:
  - One type of ambiguity occurs if the same *SI* has potentially different functions: The slash in the DUDEN-Stilwörterbuch for example is used, on the one hand, to separate lexical variants from each other (e.g. two variants of a phraseme component as in 'den starken Mann mimen/markieren') and, on the other hand, it indicates the beginning and end of an item giving pragmatic-semantic information (as in 'grüner Star /eine Augenkrankheit/'). The dealing with this type of ambiguity is simple as long as the two functions are — linguistically speaking

[9] On this subject see also NEFF, BIRD & RIZK 1988 [17].

[10] The permissible values of these item classes can only be specified by means of very general data types.

— in complementary distribution, i.e., if each function is bound to a specific context. Error-prone, with respect to dictionary entry parsing, are those cases where contextual features do not suffice to identify the function of the SI.

– Most dictionary texts contain symbols which — depending on the context — can either be a SI giving information on the dictionary form or can be part of an item giving information on the dictionary subject.[11] An example of this second type of ambiguity is the colon in the DUDEN-Stilwörterbuch which can either function as a SI separating the *comment on form* from the rest of the dictionary entry, or can be part of an example group, e.g., in 'die Mannschaften trennten sich 0:0' (the teams parted with a score of 0:0). In the following we will use the term **indefinite structure indicator** (SI^{ind}) for the symbols which reveal this type of ambiguity between dictionary form and dictionary subject in order to distinguish them terminologically from symbols called **definite structure indicators** (SI^{def}), which may also be functionally ambiguous but which always carry information about the dictionary form (like the slash mentioned above).

Dictionary entry parsers have to provide facilities to handle both types of ambiguities, because the automatic identification of the dictionary entry structure is mainly controlled by the correct interpretation of SIs. The wrong interpretation of $SI\text{s}^{ind}$ will inevitably result in bad segmentation results and in inadequate structural analyses. In the following section we will show that the LEXPARSE systems provides straightforward solutions to these problems based on a differentiated handling of $SI\text{s}^{def}$ and $SI\text{s}^{ind}$.

2.4 Requirements for a Dictionary Entry Parser

In order to cope with the specific problems related to the automatic conversion of typesetting tapes into lexical databases, a dictionary entry parser must meet the following requirements:

- The segmentation of the typesetting tape into separate dictionary entries and the structural analyses of these entries must be based primarily upon the automatic recognition and the correct interpretation of SIs.
- Ambiguities due to $SI\text{s}^{ind}$ have to be recognized and handled adequately.
- The grammar formalism supported by the dictionary entry parser has to provide operators and mechanisms to cope with phenomena — such as the handling of fonts and typefaces and the management of counters — that cannot be expressed by means of a context-free grammar formalism.

[11] For the distinction between dictionary form and dictionary subject see WIEGAND 1991:18f [24].

- Iteration of an arbitrary number of items belonging to the same item class is typical for hierarchical microstructures of dictionary entries and has to be supported by the grammar formalism. Recursive rules are not appropriate, because they result in a misleading representation of iterative structures.

In addition, a dictionary entry parser must meet some general requirements:

- A simple and easy to learn yet powerful grammar formalism should assist the user in developing dictionary entry grammars.

- In order to meet specific user needs, the format of the parse trees should be user-configurable. The system should at least be capable of displaying parse trees as hierarchical attribute-value structures for an easy-to-read view, and as annotated SGML structures for a convenient conversion of the data into other formats.

- The parser should provide special functions and protocol files in order to facilitate the development and the debugging of dictionary entry grammars.

- Since the process of parsing is usually carried out *offline*, speed of execution plays a minor role for dictionary entry parsing. However, brief response times are an advantage, especially during the time period in which the dictionary entry grammar is being developed and the user heavily interacts with the system.

- The program should be capable of permanently saving all settings and options in order to facilitate subsequent parsing sessions with identical settings and options.

- The implementation of the program should be independent of a specific host architecture or operating system so as to obtain a maximum portability.

In the following we will explain how the LexParse parsing system meets these requirements. In contrast to former approaches to dictionary entry parsing, the LexParse system was not built upon an existing natural language parser but was developed from scratch, taking into account the specific problems related to the parsing of dictionary text stored on typesetting tapes.

3 LexParse System Design and Concepts

3.1 The System's Architecture

The LexParse system consists of two files: the configuration file and the program executable comprising the following modules:

- The **main function** initializes all data structures, error handlers and uses default values for the program's settings. It also contains the main loop for dictionary entry parsing.
- The **parsing engine** itself is split into three different components:
  - the preprocessor
  - the scanner
  - the parser
- The **output function** displays the resulting parse tree in different user-definable styles and performs textual conversions to the terminal nodes of the parse tree to meet user-specific needs.

Figure 1 on page 11 illustrates the architecture of the LexParse system.

The **configuration file** is a pure ASCII text file, maintained by a standard text editor, which contains all settings, commands and the dictionary entry grammar. The configuration file may be split into several files if a separation between general settings controlling the general parse process and dictionary-specific settings (the dictionary entry grammar and control of the scanner and the preprocessor) is desired.

At runtime, LexParse reads and interprets the configuration file, verifies the grammar and checks all settings for completeness. If no errors did occur so far, the program opens the specified input file for parsing.

The **preprocessor** reads the input file on a line-by-line basis and converts tape specific format sequences and codes, i.e., removes superfluous control codes and changes escape sequences into readable characters.

The **scanner** splits the lines being read by the preprocessor into atomic items, the so-called *tokens*. These tokens may represent textual elements (e.g. words, numbers, punctuation characters) or structure indicators (which may be defined as XCodes; cf. the next section for details).

The **parser** matches these tokens with the rules of the dictionary entry grammar and creates a parse tree by expanding the starting symbol of the grammar.

The **postprocessor** applies textual conversions to the terminal nodes of the parse tree, i.e., the representation of the lexicographic items. For instance German 'Umlaute' may be converted into the respective TeX or LaTeX representation.

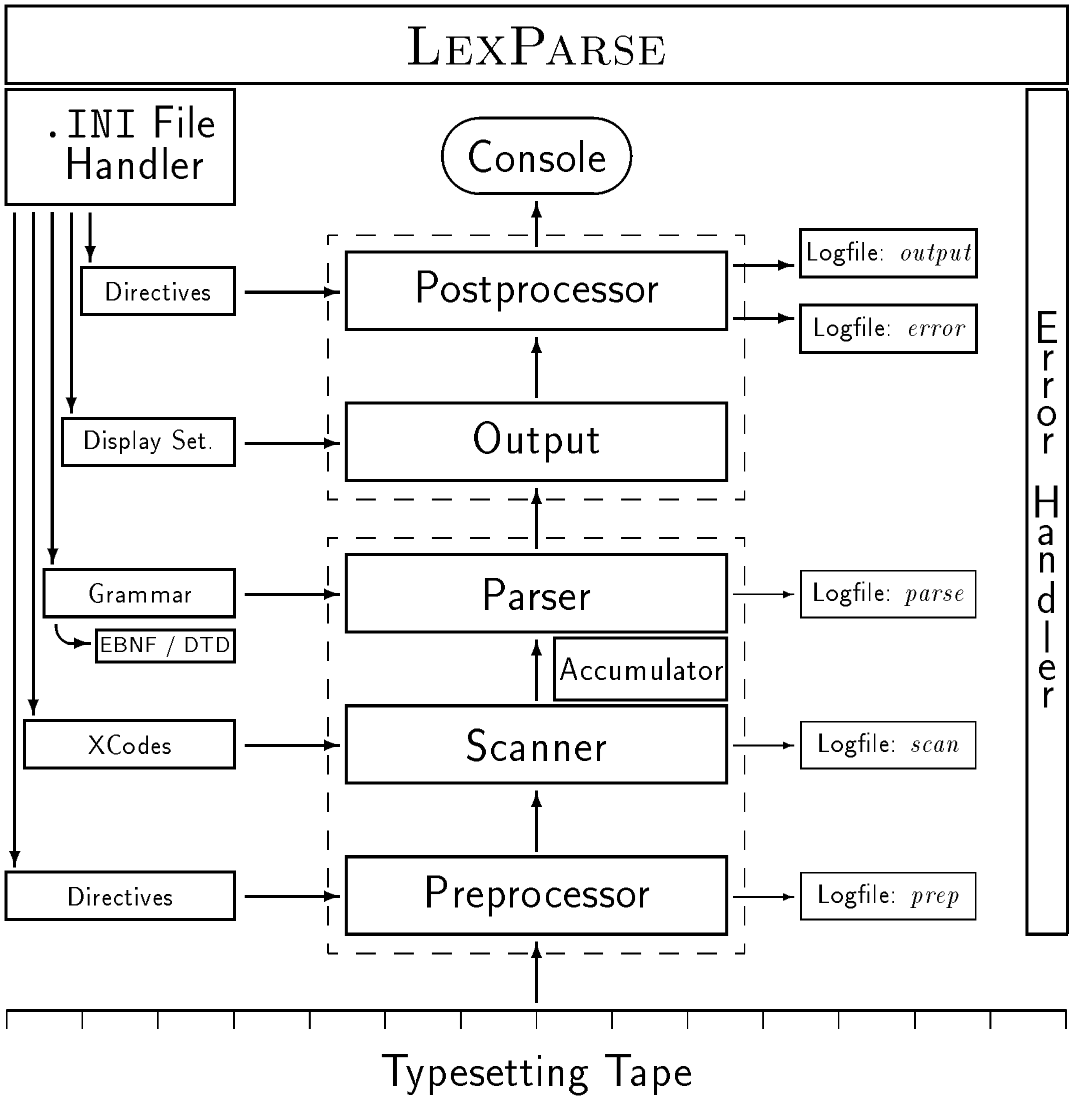

Figure 1: The Design of the LexParse Program

The program displays the parse tree according to user-definable style settings. Entries being marked as non-wellformed are displayed with a maximum parse tree so that the source for the error is almost immediately apparent.

All actions performed by the system are logged in different **protocol files** in order to facilitate debugging. The user can specify whether the protocol shall comprise all entries or only those which were marked as non-wellformed. A final **status report** informs the user about the total number of correct and incorrect entries and the time spent on parsing.

As a further option, the LexParse grammar may be displayed and saved as a file in **EBNF** (*Extended Backus Naur Form*) notation for documentation purposes.

As a further option, the LexParse grammar may be displayed and saved as a file in **SGML DTD** (*Document Type Declaration*) notation for conversion purposes.

3.2 LexParse Concepts

Right after the preprocessor has applied all directives, the scanner splits the input lines into separate tokens, i.e., atomic strings being enclosed by token delimiters. LexParse distinguishes two classes of tokens, the so-called **XCode tokens** and the so-called **literal tokens**. XCodes are predefined symbols which are generally used to represent *SI*s. A literal-token is a string containing a word, a number, or a punctuation character.

The following conditions control the scanner to separate tokens from each other:

- Word delimiters always separate consecutive tokens. Word delimiters are:
  - the space character (ASCII 32),
  - the newline character,[12]
  - the formfeed character (ASCII 12), and
  - the tabulator character (ASCII 9).[13]
- A change from a letter to a digit and vice versa results in separate tokens.

[12] On UNIX systems, the linefeed character (ASCII 10) represents the newline character. On OS/2™ and MS-DOS systems, a combination of a linefeed character (ASCII 10) followed by a carriage return character (ASCII 13) represents the newline character.
The configuration switch `[Settings] SkipNLchar` determines whether the newline character is treated as a word delimiter or whether it is skipped.
The configuration switch `[Settings] NLAsXCode` determines whether the newline character is generated as an XCode `X_NL` or whether its is treated as a word delimiter.

[13] The configuration switch `[Settings] TabAsXCode` determines whether the tabulator character is generated as an XCode `X_TAB` or whether it is treated as a word delimiter.

- A change from a letter to a punctuation character and vice versa results in separate tokens.
- An occurence of an XCode-token.

Note: By default the set of characters being considered as 'letters' contains all uppercase letters **A–Z** and all lowercase letters **a–z**. This set may be further expanded using the configuration option `[Scanner] AddLetters`.

The difference between XCodes and literals is crucial to the LexParse approach and will be dicussed in more detail in the following sections.

3.2.1 XCode Tokens

XCodes are predefined symbols provided by the LexParse system which are generally used to represent SIs. In the typesetting representation of the dictionary text, SIs are represented either as control codes (this applies to all $SI\mathrm{s}_{typ}$) or as ordinary characters (this applies to most $SI\mathrm{s}_{ntyp}$).

In general, the grammar writer assigns XCodes to all patterns denoting $SI\mathrm{s}^{def}$ and, in specific cases, also to the patterns denoting $SI\mathrm{s}^{ind}$.[14]

The grammar writer who is in charge of this assignment has to consider the predefined semantics of the XCodes:[15] Punctuation characters must be assigned to the appropriate XCodes (e.g. the colon `':'` must be assigned to the XCode `XPCOL` denoting the punctuation character 'colon' (:)) and control codes must be assigned to their appropriate XCodes (e.g. a control code `&234&16` indicating the beginning of a new dictionary entry in the typesetting tape must be assigned to the XCode `XFLBE` denoting the beginning of dictionary entries).

LexParse offers, aside from user-definable XCodes, a wide variety of predefined XCodes labels, each of which consists of a combination of exactly five letters and belongs to a specific XCode class:

[14] A detailed discussion of strategies for the assignment of XCodes to $SI\mathrm{s}^{ind}$ can be found in Hauser 1993 [9].

[15] This does not apply to XCodes of class `XU***`, i.e., user-definable XCodes.

XCode Classes	
XCodes	Class
XB***	*B*rackets
XP***	*P*unctuation characters
XC***	(Other) *C*haracters
XF***	*F*ormat
XFT**	*F*ormat: *T*ypefaces (Subclass)
XU***	*User-definable*
X_***	Special file characters

A complete list of all predefined XCodes and their semantics can be found on page 59 in the appendix.

Various operators provided by the LexParse grammar formalism allow for a differenciated treatment of XCodes: The *skip* operators consume XCode tokens whereas the *$* operators block the consumption of XCode tokens.[16] In section 4.3 (p. 38) we will show how this can be applied to the functional disambiguation of SIs^{ind}.

3.2.2 Literal Tokens

All tokens not assigned to XCodes are regarded as literal tokens or **literals**. These literals can be words, numbers or punctuation characters; a word is defined as being any sequence of 'letters'. The set of characters being regarded as 'letters' contains all uppercase letters **A**–**Z**, all lowercase letters **a**–**z**, and and all characters as defined by the configuration option `[Scanner] AddLetters`.

Any sequence of digits (0-9) is regarded as being a number.

All characters which are neither digits nor letters and which are not defined as XCodes are treated as punctuation characters which always result in single literal tokens.

In contrast to XCodes, literals are consumed by both *skip* operators and *$* operators.

3.2.3 Typeface Changes and Typeface States

Typesetting tapes represent typographic information in the form of control codes which indicate transitions from one typeface to another typeface.[17] In most cases, the end of a specific typeface is not explicitely encoded but rather implicitely indicated by various types of events, e.g., the end of a paragraph, the end of an entry or the beginning of another typeface. The scope of a specific typeface

[16] For a detailed discussion of these operators refer to section 4.1 on page 19.

[17] In this paper, we use the term 'typeface' meaning both fonts and typefaces.

thus reaches from the occurence of the respective control code up to one of these events.

All control codes indicating typographic information are assigned to XCodes of class `XFT**`. LexParse provides a large set of XCodes to cover all possible typefaces and supports two alternative approaches to the handling of typographic information:

Operation mode 'TypefaceChanges': Within the operation mode 'TypefaceChanges'[18], XCodes denoting typefaces (XCodes of class `XFT**`) are treated like all other XCodes: if an XCode is specified as a terminal in a grammar rule, the parser expects (and consumes) the respective XCode token in the input. The rule fails, if the next token available in the input does not match this XCode terminal.

An example grammar which makes use of the 'TypefaceChanges' mode is given in `(1)`:

```
(1)  S  ->    XFTbo,  A
        |     XFTit,  B
        |   [ XFTst ] C.
```

According to the syntax and semantics of the LexParse grammar formalism to be specified in the following section, this rule is interpreted in the following way: Category `S` is expanded to category `A` if the control code for a typeface change to 'bold' (which is assigned to the XCode `XFTbo`) is available as the next token in the input; category `S` is expanded to category `B` if the control code for a typeface change to 'italics' (XCode `XFTit`) is available as the next token, and category `S` is expanded to category `C` if the control code for a typeface change to 'standard' (XCode `XFTst`) is available as the next token in the input. Since the terminal `XFTst` is optional, `S` may also be expanded to category `C` if no XCode token is available at all. This optionality is needed for cases in which the transition to typeface 'standard' has occured before the rule is applied, because in these cases the respective XCode token is no longer available in the input.

However, the optionality of category `C` may yield undesired results: If a transition to typeface 'bold' or 'italics' has occured before the rule is applied, `S` will mistakenly be expanded to `C`, because the `XFTbo` or `XFTit` tokens are no longer available in the input and an expansion to `A` or `B` is, therefore, prevented.

Operation mode 'TypefaceStates': When the operation mode 'TypefaceStates'[19] is set, XCodes denoting typographic information (i.e. all XCodes of

[18] This operation mode is the default mode and corresponds to the configuration switch `[Settings] TypeFaceStates = Off`.

[19] This operation mode must be explicitly set by the configuration switch `[Settings] TypeFaceStates = On`.

the class XFT**) cannot be used as terminals in the grammar. Instead they are interpreted as conditions controlling whether a rule may be applied or not.

The parser maintains an internal 'typeface indicator' which stores the current typeface state. Each XFT** token occuring in the input is consumed immediately and the value of the 'typeface indicator' is changed accordingly. Thus, the system can retrieve at any time the current value of the 'typeface indicator'.

In order to emphasize the special role of these XCodes, the grammar formalism demands that colons are placed right after all XFT** codes. The rule following the colon can be applied only if the condition (expressed by the XFT** code and the colon) is true, i.e., if the 'typeface indicator' is set to the value corresponding to this XFT** code. An example is given in (2):

```
(2)  A  ->  XFTst: X.
```

Category A may be expanded to category X if and only if the 'typeface indicator' has the value 'standard', i.e., if the last XCode token XFT** which occured in the typesetting tape so far was XFTst. In contrast to the mode 'TypefaceChanges', it is of no importance whether the 'typeface indicator' has reached this state right before the expansion of category A to category X or at an earlier time. The crucial thing is that XFTst was the last XFT** token occurring in the typesetting file, which means that the 'typefaces indicator' has the typeface 'standard' as its current value.

With the help of operation mode 'TypefaceStates', we can now reformulate the example grammar (1) in the following way:

```
(3)  S  ->  XFTbo: A
        |   XFTit: B
        |   XFTst: C.
```

Category S may be expanded to category A if the parser has reached typeface state 'bold'; S may be expanded to B if the parser has reached typeface state 'italics' and S may be expanded to C if the parser has reached typeface state 'standard'. If the 'typeface indicator' is in any other state, category S cannot be expanded by the rules given in (3).

Typeface changes and the scope of typographic information cannot be directly expressed in context-free rewrite rules; this is why specific facilities overcoming these limitations are required. Managing typeface states with a 'typeface indicator' is more suitable than using terminal symbols for typeface changes, because it allows for a deterministic description of typographic information in the dictionary entry grammar. Moreover, this approach reflects the difference between the semiotic status of SI_{typ} and SI_{ntyp}, as discussed in section 2.3 (p. 6).

3.2.4 The Accumulator

Preterminal categories of dictionary entry grammars may be expanded by rules containing terminals and/or operators. Terminals may match with tokens being generated by the parser's scanner; operators are symbols which represent specific patterns of terminals and may match with a single token or a sequence of tokens.

Tokens or sequences of tokens, which match the terminals or operators specified within a rule, are consumed by the parser and are collected in a container object called **accumulator**. Each category has its own accumulator which consecutively 'collects' all tokens that can be matched and consumed from the input while expanding this category. The result of the accumulation process for each category is finally displayed in the parse tree.[20] The management of the accumulator takes care that all tokens are correctly assigned to the accumulator of a category resp. to the accumulators of its daughter categories.

The following example illustrates how the accumulators of the category `A` and `B` are managed:

```
(1)  A  ->  $$ ":" B $$ ".".

     B  ->  "(" $$ ")".

(2)  a b c: (d e f) g h i.

(3)  [a] [b] [c] [:] [(] [d] [e] [f] [)] [g] [h] [i] [.]

(4)  +--> A: "a b c : g h i ."
           +--> B: "( d e f )"

(5)  Accumulator of A: "a b c : g h i ."

(6)  Accumulator of B: "( d e f )"
```

In `(1)` category `A` may be expanded to any sequence of literals (expressed by the $$-operator), a colon, category `B`, a second sequence of literals, and a dot. Category `B` as in `(2)` may be expanded into any sequence of literals enclosed by parentheses. The input `(2)` thus results in the token list `(3)`. Expanding category `A` with this token list results in the parse tree `(4)`.

By default, literal tokens are added to the accumulator of a category and displayed in the parse tree, whereas XCode tokens are not added to the accumulator and are thus not shown in the parse tree. This reflects the conceptual status of XCodes denoting SIs which — in contrast to items — are not elements of the hierarchical microstructure of a dictionary entry. In particular cases, this default

[20] In the configuration's option `[Display] Format` the variable `%$` represents the contents of the category's accumulator and may be used in the format string accordingly.

setting may be inverted by marking an XCode or a string terminal with the circumflex character (^).[21] Literal tokens, for example, which function as $SI\text{s}^{ind}$ and are thus not assigned to XCodes, may be marked with the circumflex so as to prevent them from being displayed in the parse tree. The effect of applying the circumflex character can be demonstrated by the following example:

```
(7)  A1  ->  XBRPO  * XBRPC.
     A2  ->  XBRPO^ * XBRPC^.

(8)  (a b c)

(9)  [XBRPO] [a] [b] [c] [XBRPC]

(10) +--> A1: "a b c"
     +--> A2: "( a b c )"
```

In `(7)`, both categories `A1` and `A2` cover a sequence of literal tokens, which is enclosed by the XCode tokens `XBRPO` (*BRacketParenthesisOpen*) and `XBRPC` (*BRacketParenthesisClose*) denoting a pair of parentheses. The input chain `(8)` results in the token list `(9)`. As shown in (10), the resulting parse tree for category `A1` only displays the literal tokens, whereas the parse tree for category `A2` also contains the parentheses '(' and ')', because the respective XCodes are marked with the circumflex character in the expansion rule for `A2`. More examples for the application of the circumflex character are given in section 4.3 (p. 38).

[21] If an XCode of class `XB***`, `XP***`, or `XC***` is to be added to the accumulator, the character corresponding to the semantics of this XCode is copied to the accumulator. If an XCode of class `XF***` or `XU***` is to be added to the accumulator, the XCode's name is copied to the accumulator.

4 The LexParse Grammar Formalism

4.1 Formal Description of the LexParse Grammar

In essence, the LexParse grammar formalism uses simple context-free production rules supplemented by additional features for the handling of optionality, alternation and multiple repetition of rules as well as for the specification of closed sets of terminals. In addition, the formalism allows parser directives to be built into the grammar, e.g., to reset counters or to designate a dictionary entry as being incorrect.

4.1.1 Formal Syntactic Description

In the following, we describe the LexParse grammar formalism using an EBNF (*Extended Backus Naur Form*)[22] notation:

```
Category    = CatSpec "->" [ Exceptions ":" ]
                           [ SimpleRule "->" ] Rules ".".

Exceptions  = Exception { "," Exception }.

Exception   = "-" XCode
            | "+" XCode.

Rules       = Rule { "|" Rule }.

Rule        = SimpleRule [ "?" TermSpec ] [ "@" ].

SimpleRule  = RuleNode { "," RuleNode }.

RuleNode    = CatSpec
            | TermSpec
            | "[" SimpleRule "]"
            | "<" SimpleRule ">"
            | "#"
            | "$"
            | "$$"
            | "$\" """"String""""
            | "*" TermSpec
            | "*\" TermSpec
            | "\" TermSpec
            | Counter
```

[22]This Extended BNF uses the following additional meta characters: The definition character "=" replaces the BNF definition characters "::=". Square brackets "[ ]" indicate optionality of an item whereas curly brackets "{ }" indicate the possible repetition of an item. A dot terminates each rule. Cf. Jensen & Wirth 1985 [15].

```
              | State
              | .

TermSpec     = Terminal
              | "{" TerminalSet "}".

TerminalSet = Terminal { "," Terminal }.

Counter      = "%" [ "R" ] CounterType.

CounterType = "I"
              | "i"
              | "A"
              | "a"
              | "1".

State        = StateCond
              | StateAssign.

StateCond    = StateId "=" StateVal ":"
              | StateId "!=" StateVal ":".

StateAssign = StateId ":=" StateVal.

CatSpec      = [ "_" ] CapLetter { Char }.

StateId      = LowLetter { SChar }.

StateVal     = LowLetter { SChar }.

Char         = Letter
              | Digit
              | "_".

SChar        = Letter
              | "-"
              | "_".

String       = ALetter { ALetter }
              | Digit { Digit }.
              | PChar
              | RegularExpression.
```

The term `Letter` denotes an element out of the set of letters `A` to `Z` and `a` to `z`.

The term `CapLetter` denotes an element out of the set of letters `A` to `Z`.

The term `LowLetter` denotes an element out of the set of letters `a` to `z`.

The term `Digit` denotes an element out of the set of digits `0` to `9`.

The term `ALetter` denotes an element out of the set of letters `A` to `Z`, `a` to `z` and all characters as defined by the configuration option `[Scanner] ALetters`.

The term `PChar` denotes any other character not being a `ALetter` or a `Digit`.

The term `CatSpec` denotes a category specifier. Whenever the specifier is preceded by an underscore character ('_') the corresponding category is considered a (user-defined) virtual category.[23] Virtual categories will not be displayed in the resulting parse tree by default.[24]

The term `StateId` denotes a state identifier as specified in section `[States]` of the configuration.

The term `StateVal` denotes a value for a state definition as specified in section `[States]` of the configuration.

The term `RegularExpression` denotes a regular expression `rep` which must be of the form `^rep$`. All special characters being part of a regular expression are described together with their semantics below.

Within the mode 'TypefaceChanges' an additional rule is defined:

```
Terminal     = XCode [ "^" ]
             | """" String """" [ "^" ].
```

The term `XCode` denotes an element out of the set of XCodes. The complete list of all XCodes and their semantics can be found on page 59 in the appendix.

Within the mode 'TypefaceStates' an additional rule is defined:

```
Terminal     = XCode [ "^" ]
             | XCodeFT ":"
             | XCodeFT "!"
             | """" String """" [ "^" ].
```

The term `XCode` denotes an XCode of any class but class `XFT**`. The term `XCodeFT` denotes an XCode of class `XFT**`.

Invalid Sequences The following sequences of operators are not allowed:

```
$$, $$
$$, $\
$$, $
$$, #
$$, ?
$$, %
```

[23] User-defined virtual categories may be used only if the configuration switch `[Settings] VirtualCategories` is enabled.

[24] All virtual categories can be made visible in the generated parse tree by enabling the configuration switch `[Display] VirtualCategories`.

4.1.2 Semantic Description

Note: All parser directives are ϵ productions, that is, they do not consume any tokens from the input.

XCode-Exceptions: All XCode-exceptions are parser directives.

```
C -> -XCODE: R.
```

Within the scope of category `C`, the XCode `XCODE` is considered to be a literal and not an XCode.

```
C -> +XCODE: R.
```

Within the scope of category `C`, the XCode `XCODE` is considered to be an XCode and not a literal.

Note: XCode exceptions may be nested up to any level.

Rule Abbreviation:

```
C -> X -> R1 | R2.
```

This rule will be replaced by the rule

```
C -> X, R1 | X, R2.
```

which means that both rules are equivalent.

Generals:

```
C -> R1 | R2.
```

If the expansion of category `C` to `R1` fails, `C` may be expanded to `R2`.

```
C -> [ R1 ] R2.
```

Category `C` may be expanded to `R1 R2` or solely to `R2`. (`R1` is optional and *may* be expanded at this position once.)

```
C -> < R1 > R2.
```

Category C may be expanded to multiple instances of R1 followed by R2 or solely to R2. (R1 is optional and *may* be expanded at this position several times.)

```
C -> { t1, t2, t3 }.
```

Category C expands to *one* of the three terminals t1, t2 or t3.

```
C -> * t.
```

Category C expands to a sequence of tokens up to and including a specific token which matches the terminal t.

```
C -> *\ t.
```

Category C expands to a sequence of tokens up to a specific token which matches the terminal t. However, this matching token is not consumed by the parser and added to the accumulator of category C but remains in the input.[25]

```
C -> \ t, R.
```

If the next token available in the input does **not** match the specified terminal t, category C expands to R. The token matching the specified terminal t is not removed from the input.

```
C -> $.
```

Category C expands to a single literal token.

```
C -> $$.
```

Category C expands to a sequence of (at least one) literal tokens.

```
C -> $\ "string".
```

Category C expands to a sequence of (at least one) literal tokens up to a specific literal token which matches the quoted string string. However, this matching literal token is not consumed by the parser and added to the accumulator of category C but remains in the input.[26]

[25] The LexParse rule `C -> *\t.` with $C \in N, t \in T$ corresponds to the contextsensitive rule $Ct \longrightarrow \alpha t$ with $C \in N, t \in T$ where α covers any sequence of tokens except the token matching t.

[26] The LexParse rule `C -> $\t.` with $C \in N, t \in T$ corresponds to the contextsensitive rule $Ct \longrightarrow \alpha t$ with $C \in N, t \in T$ where α covers any sequence of literal tokens except the token matching t.

```
C -> #.
```

Category `C` expands to a single literal token which must be a number, i.e., a sequence of digits.

```
C -> "string".
```

Category `C` expands to a literal token which matches the quoted `string`.

Note: The `string` is considered to be a regular expression `rep` only, if it is specified in the form `^rep$`.

```
C -> R ? t.
```

Category `C` expands to `R` if and only if `C` could be expanded to `R t`.[27]

Counters:

```
C -> %I.
```

Category `C` expands to a single literal if this literal matches an uppercase roman number (i.e. `I, II, III, IV` ...).

```
C -> %i.
```

Category `C` expands to a single literal if this literal matches a lowercase roman number (i.e. `i, ii, iii, iv` ...).

```
C -> %A.
```

Category `C` expands to a single literal if this literal matches an uppercase latin letter (i.e. `A, B, C, D` ...).

```
C -> %a.
```

Category `C` expands to a single literal if this literal matches a lowercase latin letter (i.e. `a, b, c, d` ...).

```
C -> %1.
```

Category `C` expands to a single literal if this literal matches an arabic number (i.e. `1, 2, 3, 4` ...).

[27]The LexParse rule `C -> R ? t.` with $C \in N, R \in \Sigma^*, t \in T$ corresponds to the contextsensitive rule $Ct \longrightarrow Rt$ with $C \in N, R \in \Sigma^*, R \neq \epsilon, t \in T$.

Note: If a specific counter symbol could be derived successfully, the corresponding counter is incremented.

Resetting the Counters:

```
C -> %RI, Rule.
```

The counter for uppercase roman numbers is reset to `I` before category `C` is expanded to `Rule` .

```
C -> %Ri, Rule.
```

The counter for lowercase roman numbers is reset to `i` before category `C` is expanded to `Rule` .

```
C -> %RA, Rule.
```

The counter for uppercase latin letters is reset to `A` before category `C` is expanded to `Rule` .

```
C -> %Ra, Rule.
```

The counter for lowercase latin letters is reset to `a` before category `C` is expanded to `Rule` .

```
C -> %R1, Rule.
```

The counter for arabic numbers is reset to `1` before category `C` is expanded to `Rule` .

Note: All directives resetting a counter are parser directives.

States:

```
C -> StateId = StateVal: R.
```

Category `C` is expanded to `R` if and only if the state `StateId` currently has the value `StateVal`.

```
C -> StateId != StateVal: R.
```

Category `C` is expanded to `R` if and only if the state `StateId` currently has any other value but value `StateVal`.

```
C -> StateId := StateVal, R.
```

Category `C` is expanded to `R` right after the value `StateVal` has been assigned to state `StateId`.

Note: All states must have been defined together with their possible values in section `[States]` of the configuration.

Typeface States:

```
C -> XFT** : R.
```

Category `C` is expanded to `R` if and only if the internal indicator for typeface states is in the state of `XFT**`.

```
C -> XFT** ! R.
```

Before category `C` is expanded to `R`, the internal indicator for typeface states is set to the specified value `XFT**`.

Note: All directives handling typeface states are parser directives.

Additional Parser Directives:

```
C -> t^.
```

Category `C` expands to a token matching the specified terminal `t`. If this token is an XCode token and it is marked as indicated, it will be added to the accumulator of category `C`. By default, all XCode tokens are *not* added to the accumulator. If this token is a literal token and it is marked as indicated, it will not be copied to the accumulator. By default, all literal tokens are added to the accumulator.

If an XCode of class `XB***`, `XP***`, or `XC***` is to be added to the accumulator, the character corresponding to the semantics of this XCode is copied to the accumulator. If an XCode of class `XF***` or `XU***` is to be added to the accumulator, the XCode's name is copied to the accumulator.

```
C -> R @.
```

Category `C` expands to `R`. After `C` has been expanded, the current entry is marked as being non-wellformed and backtracking is terminated for the parsing of this entry.

4.1.3 Regular Expressions in LexParse

The LexParse program supports the following characters as being part of a regular expression:[28]

[28]The term 'source string' denotes a line as being read from the input data file as far as the regular expression is used in the context of a preprocessor or a postprocessor directive. The term 'source string' denotes a literal token as far as the regular expression is used within the specification of string terminals for the grammar.

`^`	This expression matches the beginning of the source string.
`$`	This expression matches the end of the source string.
`.`	This expression matches any character.
`[abcA-Z]`	This expression matches a set consisting of the characters `a`, `b`, `c` and the sequence of characters `A` to `Z`.
`[^abcA-Z]`	This expression matches any character except the set consisting of the characters `a`, `b`, `c`, and the sequence of characters `A` to `Z`.
`\`	This expression matches the character `c`. This may be used to override any special meaning given to a character.
`*`	If one of the expressions as given above is followed by this character, then this expression matches a sequence of zero or more instances of this expression.

Note: A regular expression `rep` as being used within the specification of terminals for the grammar must always be specified in the form `^rep$` so that it can be distinguished from other strings.

4.2 Annotations to the Formalism

The LexParse grammar formalism uses several features in writing dictionary entry grammars and in handling the special requirements of dictionary entry parsing. In this section, these features will be explained in detail.

4.2.1 Single and Multiple Optional Rules

The grammars in (1) and (2) generate the same language.

```
(1)  A    ->  [ X ].

(2)  A    ->  Xaux.

     Xaux ->  X
          |   .
```

However, the resulting parse trees differ in that the parse tree in (2) contains an additional node `Xaux`, whereas in (1) category `A` expands directly to category `X`. Optional rules, thus, simplify the grammar and lead to a representation of the dictionary entry structure, which is more adaquate because the semantically empty category `Xaux` is not needed.

The grammars in (3) and (4) also generate the same language.

```
(3)  B      ->  < X >.
```

```
(4)  B     ->  Xaux.

     Xaux  ->  X, Xaux
           |   .
```

However, the resulting parse trees differ in that the parse tree in `(4)` contains additional nodes **Xaux**, whereas in `(3)` category `B` expands directly to category `X`. Due to the right-recursion used in `(4)`, the derivation of category `B` results in an asymmetrical parse tree with a depth of n, whereas the expansion of category `B` in `(3)` results in a symmetric parse tree with a uniform depth of *2* and n nodes being immediate constituents of `B`. This greatly simplifies the grammar and leads to a representation of the dictionary entry structure, in which no auxiliary category `Xaux` is required. The parse trees `(5)` and `(6)`, both obtained by deriving `X X X` from category `B`, may illustrate the difference: The parse tree in `(5)` corresponds to the grammar given in `(4)`; the parse tree in `(6)` corresponds to the grammar given in `(3)`.

```
(5) +--> B                              (6) +--> B
         +--> Xaux                                +--> X
              +--> X                              +--> X
              +--> Xaux                           +--> X
                   +--> X
                   +--> Xaux
                        +--> X
```

Another example is the category `Number` which can be defined by a pure context-free grammar as in `(7)` or — much more easily — by using the LexParse formalism as in `(8)`:[29]

```
(7)  Number ->  Sign, Digit, Digits.

     Sign   ->  "-"
            |   .

     Digits ->  Digit, Digits
            |   .

(8)  Number ->  [ "-" ] Digit < Digit >.
```

4.2.2 Using a Set of Terminals

The grammars in `(1)` and `(2)` generate the same language:

```
(1)  A    ->  X { t1, t2, t3 }.
```

[29]The category `Digit` will be defined in the next section.

```
(2)  A     ->  X, Tset.
     Tset  ->  t1
           |   t2
           |   t3.
```

Set descriptions as used in (1) lead to an additional simplification of the grammars and to a representation of the dictionary entry structure, which is more adequate because additional categories like Tset are avoided. In (1), category A may be directly expanded to one of those terminals which are a member of the respective set.

Set descriptions offer the possibility to specify finite sets with a given domain, as used for the definition of the notion *digit* in (3):

```
(3)  Digit ->  { "0", "1", "2", "3", "4",
                 "5", "6", "7", "8", "9" }.
```

Using a regular expression for the terminal string, one could also replace rule (3) by rule (4):

```
(4)  Digit ->  "^[0-9]$".
```

In the context of dictionary entry parsing, a distinction between numbers and words is quite helpful. Therefore, the LexParse grammar formalism offers a special operator to consume numerical tokens, which makes the derivation rule for category Digits again more simple. The #-operator in (5) corresponds to the regular expression string in (6).

```
(5)  Digits ->  #.

(6)  Digits ->  "^[0-9]*$".
```

Therefore, the rule in example (8) of the previous section can be replaced by the following rule:

```
(7)  Number ->  [ "-" ] #.
```

4.2.3 Using the 'Skip' Operators

The *skip*-operator enables the derivation of sequences of tokens up to a specific 'delimiter' token. 'Delimiter' tokens generally function as SIs marking the boundary of a lexicographic item and can be both, XCode or literal tokens. Two types of *skip*-operators are provided:

- The *inclusive skip*-operator (*) reads up to and including the 'delimiter' token.
- The *exclusive skip*-operator (*\) reads up to the 'delimiter' token without removing it from the input. In contrast to the *inclusive skip*-operator, this 'delimiter' token remains in the input and may be consumed by another operator.

The *inclusive skip*-operator is useful for error handling routines — as in `(1)` — or for categories covering any sequence of tokens enclosed by two tokens representing SI^{def}, as in `(2)`.

```
(1)  A    ->  X1, X2, X3 ":"
          |   * ":".

(2)  B    ->  "(" * ")".
```

In `(1)`, category `A` may be expanded to a sequence of categories `X1 X2 X3`, and a trailing colon. If this expansion fails, an alternate rule can be applied: `A` expands to any sequence of tokens (expressed by the *inclusive skip*-operator `*`) up to the colon which terminates the consumption. This enables the expansion of `A` although the constituents `X1`, `X2` and `X3` could not be derived from `A`. In this case, the alternate derivation rule for `A` serves as an error handler rule.

In `(2)`, category `B` expands to a left parenthesis and the *inclusive skip*-operator which consumes all tokens up to the next literal token matching the right parenthesis.

The *inclusive skip*-operator is quite useful for the step-by-step development of dictionary entry grammars: the grammar writer can rough in the dictionary entry structure and deal with the structural details of complex item classes at a later stage.

The *exclusive skip*-operator leads to the expansion of a category into an arbitrary sequence of tokens up to a specific 'delimiter' token, which is not included by the expansion but indicates the next token of the input stream.

```
(4)  S    ->  A, X1
          |   A, X2
          |   A, X3.

(5)  A    ->  *\ { "(", "<", "[" }.

(6)  X1   ->  "(" * ")".
     X2   ->  "<" * ">".
```

```
X3    ->  "[" * "]".
```

In `(4)`, category `S` may be expanded to category `A` followed by either category `X1`, `X2`, or `X3`. `X1`, `X2`, and `X3` can be distinguished by the delimiter tokens specified in `(6)`. If the *exclusive skip*-operator is used, category `A` covers any sequence of tokens up to one of the delimiter tokens (defined in the set under `(5)`) which terminate the consumption of this sequence. However, the delimiter token is not consumed by the parser but remains in the input. In the consecutive expansion of either `X1`, `X2`, or `X3` this specific token will be consumed and either `X1`, `X2`, or `X3` may be further expanded .

Note: If an XCode token `X_EOF` (*EndOfFile*) occurs during a *skip* operation, the current entry is marked as being erroneous. The *skip* operation terminates, the current rule fails, and the parser issues an error message (error #404: `Critical XCode detected while skipping`).

Note: If an XCode token `XFLEN` (*FormatEndofLemma*) occurs during a *skip* operation, the *skip* operation terminates and the current rule fails. In this case, the configuration switch `[Parser] SkipCriticalIsError` determines whether the current entry is marked as being erroneous and whether an error message will be issued or not.

4.2.4 Using the 'Except' Operator

The *except*-operator (\) — also called the *accept-all-except*-operator — offers the possibility to specify a condition on which rules may be applied as an exception. By naming a terminal or a set of terminals, a rule may be derived only if a token is *not* available as the next token of the input.

```
(1)  A    ->  ":" X1
          |   X2.

(2)  B    ->  \ ":" X2
          |   X1.
```

The grammars in `(1)` and `(2)` generate the same language in that they both expand a category `A` to a category `X1` if the next token available in the input is the literal '`:`'. If the next token available in the input is *not* the literal '`:`' (but any other literal token), category `A` is expanded to category `X2`.

But there is a semantic difference between the grammars in `(1)` and in `(2)`: If the next token available in the input were the literal '`:`', grammar `(1)` would derive this token directly from category `A`, and the literal '`:`' would be added to the accumulator of category `A`. The grammar in `(2)`, by contrast, would derive the literal ':' from category X1, because the first rule of category B in (2) cannot be expanded with this token being available in the input.[30]

4.2.5 Using the 'Test' Operator

The *test*-operator (?) can be used to test whether the next token available in the input is a specific token. If the test is positive, i.e., the next token available is of the type specified, it is *not* consumed by the parser but remains in the input until it can be consumed by another operator.

```
(1)  A    ->  XBRPO * XBRPC ? "!".
```

In (1), category A expands to any sequence of tokens enclosed by the XCodes XBRPO (*BRacketParenthesisOpen*) and XBRPC (*BRacketParenthesisClose*) only, if the next token following this sequence is the literal '!'. The parser does not consume this token; it remains in the input and can be consumed by another operator.

Note that the *test*-operator can occur only at the end of a rule. Only the *Error*-directive may be placed right after the *Test*-operator.

4.2.6 Using the 'Error' Directive

The *error*-directive (@) offers the possibility to construct error handler routines which mark dictionary entries as being incorrect (non-wellformed) even if they were completely derived.

An error handler routine is a category with a single derivation rule expanding to the *skip*-operator and the *error*-directive. Error handler routines enable the parser to generate maximum parse trees.

```
(1)  A    ->  X1, X2, X3, XPCOL
          |   * XPCOL @.
```

In (1), category A may be expanded to a sequence of categories X1, X2, X3, and a trailing colon which is assigned to the XCode XPCOL (*PunctuationCOLon*). If this expansion fails, an alternate rule may be applied: A expands to the *skip*-operator (*) and the colon (XPCOL). If this expansion succeeds, the *error*-directive is applied to the current entry, i.e., the current entry is marked as being incorrect.

This construction offers the possibility to continue the parsing of the entry, even when it is regarded as being non-wellformed because the categories X1, X2, and X3 could not be derived from A.

[30] Assuming — of course — that X1 can cover a ':' token at all.

Marking an entry as being non-wellformed is also necessary if the derivation of a specific rule indicates an invalid input.

```
(2)  B    ->  XPCOL, X
          |   XPSEM, X @.
```

In (2), category B may be expanded to category X if the required XCode token XPCOL is available as the next token of the input. However, if the next token being available is the invalid XCode token XPSEM, then B also expands to X but the current entry must be marked as being incorrect.

4.2.7 Using Counters

Dictionary entries describing homonymous and/or polysemous lemma signs generally use different types of counters to identify different levels of homonym and sense distinctions. Each of these levels correspond to a different degree of semantic proximity.

If homonym/sense counters are treated as 'normal' categories within a context-free grammar, they cannot be incremented after being derived and, therefore, cannot be checked for correctness.

For this reason, LexParse offers several built-in counters each of which can be incremented and reset automatically or manually by using a directive in the grammar. In example (1), we assume a three level sense distinction, in which roman numbers are used at the first level, arabic numbers at the second, and lowercase letters at the third level.

```
(1)  I.
          1.
               a)
               b)
          2.
               a)
               b)
     II.
          1.
          2.
```

During the parsing process, LexParse must set the counters to their initial value each time a deeper sense level is reached. The procedure is illustrated in (2):

```
(2)  I.   {Reset the counter: arabian numbers}
          1.   {Reset the counter: lowercase letters}
               a)
```

```
          b)
     2.   {Reset the counter: lowercase letters}
          a)
          b)
II.  {Reset the counter: arabian numbers}
     1.   {Reset the counter: lowercase letters}
     2.   {Reset the counter: lowercase letters}
```

The three-level system of sense counters in (1) can be described using the LexParse grammar formalism by the rules given in (3):[31]

```
(3)  NUM  ->  %I, ".", %R1 [ NUM ]
          |   %1, ".", %Ra [ NUM ]
          |   %a, ")"  .
```

4.2.8 Using the '$$' Operator

The *$$*-operator consecutively consumes literals and adds them to the category's accumulator. The *$$*-operator does not consume any XCode tokens, i.e., if an XCode token occurs in the input it stops the consumption of tokens.

Input (1) thus results in the token list (2). The consumption of this token list by the grammar in (3) results in (4):[32]

```
(1)  This-and more- is text!

(2)  [This] [-] [and] [more] [-]
     [is] [text] [!] [XBRPC]

(3)  A    ->  XBRPO $$ XBRPC.

(4)  +--> A: "This - and more - is text !"
```

As illustrated in (5) and (6), the *$$*-operator may be followed by a string constant. In this case, all literals up to and including this string constant are consumed by the parser.

```
(5)  B1  ->  $$ ";" .
(6)  B2  ->  $$ "." .
```

[31] For reasons of clarity, this example is kept simple. It is incomplete since category `NUM` would also accept invalid input patterns such as '`I. II. 1. 2. 3. a) b)`'.

[32] As already mentioned before, XCode tokens are by default not added to the accumulator and, thus, do not appear in the resulting parse tree in (4).

```
(7)  example1; example2; example3.

(8)  [example1] [;] [example2] [;] [example3] [.]

(9)  +--> B1: "example1 ;"
(10) +--> B2: "example1 ; example2 ; example3 ."
```

The input `(7)` results in the token list `(8)`. Given this input, the expansion of `B1` results in `(9)`, whereas the expansion of `B2` results in `(10)`.

Note that there is no upper limit on the number of literals that may be consumed by the *$$*-operator, but at least one literal is required for its successful application.

4.2.9 Using the '$' Operator

The *$*-operator consumes single literals and may therefore be used to derive a specific number of literals from a category.

```
(1)  A    ->  $, $$.
```

In `(1)`, category `A` can be expanded only if at least two literals are available in the input. The first literal is consumed by the *$*-operator, the next and all consecutive literal tokens are consumed by the *$$*-operator.

4.2.10 Using the '$\' Operator

The *$*-operator consecutively consumes literals and adds them to the category's accumulator up to a specific literal token. The *$*-operator does not consume any XCode tokens, i.e., if an XCode token occurs in the input it stops the consumption of tokens and the expansion of the rule has failed. If the specified literal token cannot be found in the input, the expansion of the rule has failed, too.

Input `(1)` thus results in the token list `(2)`. The consumption of this token list by the grammar in `(3)` results in `(4)`:

```
(1)  This is text and there is more.

(2)  [This] [is] [text] [and] [there] [is] [more] [!]

(3)  A    ->  B1, B2.
     B1    ->  $\ "and" .
     B2    ->  $$ "!" .

(4)  A
```

```
+--> B1: "This is text"
+--> B2: "and there is more!"
```

Note that the `and` literal is not added to the accumulator of category `B1` but to the accumulator of category `B2` whereas the `!` literal is added to the accumulator of category `B2` because the *$$*-operator was used.

As with the *$$*-operator, there is no upper limit on the number of literals that may be consumed by the *$*-operator, but at least one literal is required for its successful application.

4.2.11 Using XCode Exceptions

XCode exceptions are required when a SI^{ind}, assigned to an XCode, does not function as SI in a specific and limited context. Within this context, XCode exceptions can be used to disable XCodes with the effect that the scanner, instead of generating XCode tokens, generates one (or several) of those literal tokens which are denoted by the respective XCode. XCode exceptions may be nested without any restrictions, i.e., an XCode being disabled by a specific category may be enabled again by a nested daughter category.

```
(1) S   ->  X1, A, X2 [ B ].

    A   ->  -XPCOL: [ B ] $$.

    B   ->  +XPCOL: "(" * XPCOL * ")".
```

Assuming that the colon is defined as the XCode `XPCOL`, the derivation rule for category `A`, in `(1)`, disables `XPCOL`, i.e., in the scope of category `A`, the colon will not be generated as the XCode token `XPCOL` but as the literal token ':'.

Within the scope of category `B` (which is an immediate daughter of category `A`), the XCode `XPCOL` is treated in the usual way: when the token `XPCOL` occurs in the input, the consumption of tokens for the *inclusive skip*-operator is terminated because the colon functions as a SI_{ntyp} within this context.

XCode exceptions can only be specified for XCodes of the type `XB***` (brackets), `XP***` (punctuation characters) or `XC***` (other characters), i.e., defined XCodes the denotation of which is known to the scanner. Disabling or enabling undefined XCodes does not change the parser's behaviour in any way.

4.2.12 Using States

States are an add-on to the grammar formalism to support context-sensitive derivation rules and may be considered as "global variables" managed by the parser.

A state consists of a state specifier and a set of possible state values (state domain). Each state may be assigned to a specific state value (state assignment) or the state's current value may be matched against a specific value (state condition). By inserting state assignments or conditions into grammar rules the expansion of these rules may be controlled in a context-sensitive manner.

The scope of all states is global. Before a state can be used within the grammar it must be defined together with its domain of possible values and initialized in the `[States]` section of the configuration.[33]

```
[States]
verbform = { v-trans, v-itrans, v-refl, no-verb } no-verb
gender   = { m, f, nt }     ; default gender is 'm'
```

In the previous example, a state `verbform` is defined together with a domain of four values `v-trans`, `v-itrans`, `v-refl`, and `no-verb`. The default value is specified as the value `no-verb`.

Another state `gender` is defined together with a domain of three values `m`, `f`, and `nt`. Since no specific default value is given, the first value (`m`) is taken as the default value.

```
GA  -> "der", gender := m,  verbform := no-verb
     | "die", gender := f,  verbform := no-verb
     | "das", gender := nt, verbform := no-verb.

GrA -> verbform != no-verb: VInfl
     | verbform  = no-verb: NInfl.
```

4.2.13 Using Virtual Categories

Virtual categories are categories which will not be displayed in the resulting parse tree. There are two kinds of virtual categories:

Virtual categories defined by the user: Whenever a category's specifier is preceded by an underscore character ('_') the corresponding category is considered a user-defined virtual category. User-defined virtual categories may be used only if the configuration switch `[Settings] VirtualCategories` is enabled.

Virtual categories generated by the grammar handler: Whenever a rule abbreviation (ref. section 4.1.2 (p. 22)) is applied within the grammar, the grammar handler generates a virtual category.

A rule as specified in the grammar as

[33] Refer to section ?? (p. ??) for details.

```
C -> X -> R1 | R2.
```

is equivalent to the rule

```
C -> X, R1 | X, R2.
```

The grammar handler generates the following rules for category `C`:

```
C      -> X, _vcr_C.
_vcr_C -> R1 | R2.
```

Note that category `_vcr_C` is a virtual category. The prefix `_vcr_` stands for *virtual common rule* and is attached to the category's original specifier `C`.

By default, virtual categories will not be displayed in the resulting parse tree. If the configuration switch `[Display] VirtualCategories` is enabled, all virtual categories are made visible in the parse trees.

If a virtual category is removed from the parse tree, all its immediate daughters are made immediate daughters of the virtual category's mother.

If a virtual category is removed from the parse tree, its accumulator is copied to the accumulator of the virtual category's mother.

4.3 Applying the LexParse Grammar Formalism

In the following, we will show how the LexParse grammar formalism can be applied. The examples are taken from the LexParse grammar developed for the Duden-Stilwörterbuch (Duden 2, 7.th edition) (cf. Engelke 1994 [8]).

With respect to the Duden-Stilwörterbuch, the symbol 'semicolon' is considered to be a SI^{def}_{ntyp} and is therefore assigned to the corresponding XCode `XPSEM` (*PunctuationSEMicolon*). The symbol 'colon' is an SI^{ind}_{ntyp} which, in the majority of its occurences, functions as SI and is, in consequence, also assigned to its corresponding XCode `XPCOL` (*PunctuationCOLon*).

4.3.1 Applying Optional and Multiple Optional Rules

The application of (multiple) optional rules can be illustrated by the derivation rule for category `BeiGA` (*Beispielgruppenangabe*):

```
BeiGA  ->  [ UGrA ] BeiA < XPSEM [ UGrA ] BeiA > [ "."^ ] .
```

`BeiGA` expands to an optional instance of category `UGrA` (*Unspezifizierte Grammatische Angabe*) followed by one instance of category `BeiA` (*Beispielangabe*). This sequence can be repeated any number of times; the sequences are separated from each other by the SI^{def}_{ntyp} ‘semicolon’ (`XPSEM`), the series is terminated by an optional dot. The series of sequences of `UgrA BeiA` is expressed using the symbols for multiple optionality `< >`.

With respect to the DUDEN-Stilwörterbuch, the symbol ‘dot’ is an SI^{ind}_{ntyp} which, in most cases, is part of a lexicographic item and not a SI_{ntyp}. In consequence, the dot is *not* assigned to an XCode, but treated as a string terminal. In the scope of the category `BeiGA`, however, where the dot actually functions as SI_{ntyp}, the token representing the dot must be suppressed in the resulting parse tree by means of the ‘^’ operator.

In the dictionary entry for the lemma treiben, the category `BeiGA` is expanded to category `BeiA`, category `UGrA`, and another `BeiA`:

```
+--> BeiGA
   +--> BeiA "ich ließ mich von den Verhältnissen treiben"
   +--> UGrA "jmdm. etwas treiben; mit Raumangabe"
   +--> BeiA "der Sturm trieb mir den Schnee ins Gesicht"
```

A series of six `BeiA` derived from `BeiGA` can be found in the entry for the lemma Treffen:

```
+--> BeiGA
   +--> BeiA "regelmäßige, seltene Treffen"
   +--> BeiA "ein Treffen der Abiturienten"
   +--> BeiA "ein Treffen der Außenminister"
   +--> BeiA "ein Treffen verabreden, veranstalten"
   +--> BeiA "an einem Treffen teilnehmen"
   +--> BeiA "zu einem Treffen kommen"
```

4.3.2 Applying a Set of Terminals

Sets of terminals are useful for item classes having a closed set of possible values. In the DUDEN-Stilwörterbuch, the gender of German nouns is specified by means of the definite articles *der*, *die*, and *das*. Therefore, the rule for category `GA` (*Genusangabe*) which denotes a class of items giving the gender of nouns contains the following set description:

```
GA  ->  XFTst: { "der", "die", "das" }.
```

Category `GA` (*Genusangabe*) expands to one of the string constants specified in the set description. Note that `GA` can be expanded only if the current value of the typeface indicator is in the typeface state ‘standard’. This condition is expressed by the typeface feature `XFTst:`.

4.3.3 Applying the 'Skip' Operators

Category `UGrA` (*Unspezifizierte Grammatische Angabe*) occurs within the scope of category `BeiGA` (*Beispielgruppenangabe*) and denotes a class of items which are enclosed by angle brackets. Angle brackets are SIs^{def}_{ntyp} with regard to the DUDEN-Stilwörterbuch and are therefore assigned to the XCodes `XBRAO` (*BRacketAngleOpen*) and `XBRAC` (*BRacketAngleClose*) respectively. Brackets generally appear in pairs, enclosing items of specific item classes — in our example the class of items giving information on grammatical properties. The derivation rule for category `UGrA` may thus be formulated as follows:

```
UGrA  ->  XBRAO * XBRAC.
```

The *inclusive skip*-operator consumes all tokens up to and including the XCode token `XBRAC`; in consequence, `UGrA` expands to any sequence of tokens beginning with an XCode token `XBRAO` and ending with an XCode token `XBRAC`.

An application of the *exclusive skip*-operator will be shown in the following section and in section "*Applying the 'Error' Directive*".

4.3.4 Applying the 'Test' Operator

In the dictionary grammar for the DUDEN-Stilwörterbuch, the *test*-operator was used in the derivation rule for category `BeiA` (*Beispielangabe*), which may be expanded to a sequence of tokens and the optional category `SAA` (*Sprechaktangabe*) enclosed by two slashes. Examples for items belonging to the class `BeiA` containing 'embedded' items belonging to the class `SAA` are given in (1) and (2):

(1) 'du kriegst die Tür nicht zu! /ugs.; Ausruf des Erstaunens/;'

(2) 'meiner Treu! /veraltet; Beteuerungsformel/;'

(3) 'an die Tür/an der Tür klopfen;'

(4) 'er tat so, als wäre nichts gewesen, als ob/als wenn/wie wenn er nichts wüßte, als wüßte er nichts;'

Examples (3) and (4) show, that slashes do not always enclose `SAA` items; they are also used to separate two lexical or morphological variants within an example sentence. But `SAA` items can automatically be distinguished from lexical/morphological variants: category `SAA` is always enclosed by two slashes and can occur only at the end of an example sentence, which means that a semicolon must follow immediately. This semicolon may not be covered by category `SAA` itself, but rather by its mother category `BeiGA`; a constellation which cannot be expressed by means of context-free rules. This is where the *test*–operator comes into play.

```
(5) (a) BeiA  ->  * \ "/" SAA
```

```
(b)        |   * \ XPSEM.

(c) SAA    ->  "/"^ $$ XPSEM^ $$ "/"^ ? XPSEM.
```

In (5a), BeiA may be expanded to any sequence of tokens up to the literal '/' possibly indicating category SAA (acting as SI^{ind}_{ntyp}, the symbol 'slash' is not assigned to an XCode). However, due to the *exclusive skip*–operator, the literal '/' is not derived from category BeiA, but from category SAA.

In (5c), category SAA expands to the literal '/' followed by a sequence of literals, a semicolon (XPSEM), more literals, and a terminating literal '/'. Both instances of the symbol 'slash' should be marked with the '^' character so as to prevent the corresponding literal tokens from being displayed in the resulting parse tree. The *test*–operator makes sure, that SAA is expanded if and only if the next token which is available in the input after the expansion of SAA is an XCode token XPSEM. However, this XCode token is not added to the accumulator for category SAA but remains in the input.

If the application of the rules in (5a) and (5c) fails, BeiA expands to a sequence of tokens up to (but not including) the XCode token XPSEM, as in (5b).

Given the grammar in (5), the examples (1) to (4) result in the structures (6) to (9):

```
(6) +--> BeiA : "du kriegst die Tür nicht zu!"
         +--> SAA : "ugs.; Ausruf des Erstaunens"

(7) +--> BeiA : "meiner Treu!"
         +--> SAA : "veraltet; Beteuerungsformel"

(8) +--> BeiA : "an die Tür/an der Tür klopfen"

(9) +--> BeiA : "er tat so, als wäre nichts gewesen,
                als ob/als wenn/wie wenn er nichts wüßte,
                als wüßte er nichts"
```

4.3.5 Applying the 'Error' Directive

On the typesetting tape for the DUDEN-Stilwörterbuch, the beginning and the end of dictionary entries is indicated by specific control codes. These control codes can be used to specify error recovery rules which help the parser in locating the beginning of the next entry when an error has occured.

```
(1)  WA      ->  XFLBE, FK, SK [ XPAST, PKP ] XFLEN
             |   WA_Err.
```

```
(2)  WA_Err  ->  * XFLEN @.
```

Category `WA` (*Wörterbuchartikel*), representing dictionary entries, is expanded to an XCode indicating the beginning of an entry (`XFLBE`), the category `FK` (*Formkommentar*), the category `SK` (*Semantischer Kommentar*), the optional category `PKP` (*Postkommentar zur Phraseologie*) being preceeded by an asterisk ('*'; `XPAST`), and an XCode indicating the end of the dictionary entry (`XFLEN`).

If the derivation of the first rule in `(1)` fails, an alternate rule is applied: In this case, `WA` expands to a category `WA_Err` which itself expands to the *inclusive skip*-operator, the XCode indicating the end of the dictionary entry (`XFLEN`) and the *error*-directive `(2)`. The use of the *inclusive skip*-operator together with the XCode `XFLEN` leads to the consumption of all tokens up to and including the XCode indicating the end of the entry, i.e., the consumption of the complete current entry up to the beginning of the next entry. The *inclusive skip*-operator does not assign any structure to this sequence but collects the flat stream of tokens in the accumulator of category `WA_Err`.

The *error*-directive, as being part of the derivation rule for category `WA_Err`, has two consequences:

- The current entry is marked as being non-wellformed and an appropriate error message is generated by the parser.
- Backtracking is rejected, i.e., all tokens consumed so far are displayed as a maximum parse tree.

Another error recovery rule can be built for category `PKP` (*Postkommentar zur Phraseologie*) of the Duden-Stilwörterbuch by using the *exclusive skip*-operator:

```
(3)  PKP      ->  X1, X2, X3
              |   PKP_Err.

(4)  PKP_Err  ->  *\ XFLEN @.
```

If none of the daughter categories `X1`, `X2`, or `X3` of category `PKP` can be derived, `PKP` expands to an error handling category `PKP_Err` which itself expands to any sequence of tokens up to the XCode indicating the end of the dictionary entry `XFLEN`, as in `(4)`.

However, this XCode token `XFLEN` is *not* consumed by the parser and derived from category `PKP_Err`. Instead, it remains in the input and may be consumed by another operator, e.g., the one to be found in the derivation rule for category `WA` in `(1)`. The terminal `XFLEN` being part of this derivation rule matches with the XCode token `XFLEN` and may then be consumed by the parser.

4.3.6 Applying Counters

The DUDEN-Stilwörterbuch distinguishes between three different levels of semantic proximity. This distinction is indicated by means of three different types of counters: roman numbers, arabian numbers, and lowercase letters. The LEXPARSE grammar for the DUDEN-Stilwörterbuch reflects these levels as follows, using three distinct built-in counters:

```
PAR  ->  XFTbo: %I, ".", %R1.
PAA  ->  XFTbo: %1, ".", %Ra.
PAB  ->  XFTbo: %a, XBRPC.
```

The categories `PAR`, `PAA`, and `PAB` denote the different homonym/sense levels. Each category can be expanded if and only if the typeface state condition `XFTbo:` is met, i.e., the typeface indicator is in the state 'bold'. All categories cover a separate counter symbol and provide the following counting scheme:[34]

```
I.    {Reset the counter: arabian numbers}
      1.   {Reset the counter: lowercase letters}
           a)
           b)
      2.   {Reset the counter: lowercase letters}
           a)
           b)
II.   {Reset the counter: arabian numbers}
      1.   {Reset the counter: lowercase letters}
      2.   {Reset the counter: lowercase letters}
```

The counter for roman numbers, like all other counters, is reset automatically by the parser at the beginning of a new dictionary entry.[35]

4.3.7 Applying the '$$' Operator

As pointed out already, example groups (denoted by the category `BeiGA`) may consist of several example sentences (denoted by the category `BeiA`) which are separated by the SI_{ntyp}^{def} 'semicolon'. Simplified versions of the derivation rules for `BeiGA` and `BeiA` are given in (1):

```
(1)  BeiGA  ->  BeiA < XPSEM BeiA > [ "."^ ] .
```

[34]Since the counter symbol must be in typeface 'bold', tokens covering the counter cannot be confused with tokens covering abreviated lemmata which can, in principle, also appear at this position. This is particularly useful for lemmata with initial letters **I** or **V** which are abbreviated as "I." and "V." and could be easily confused with roman counter symbols.

[35]The configuration switch `[Parser] AutoResetCounter` determines this behaviour. This switch is enabled by default.

```
BeiA   ->  $$.

(2)  This is example 1;
     This is example 2;
     This is example 3.

(3)  [This] [is] [example] [1] [XPSEM]
     [This] [is] [example] [2] [XPSEM]
     [This] [is] [example] [3] [.]

(4)  +--> BeiGA:
          +--> BeiA: "This is example 1"
          +--> BeiA: "This is example 2"
          +--> BeiA: "This is example 3"
```

Category `BeiA` expands to the *$$*-operator consuming any sequence of literals up to the next XCode token; input `(2)` thus results in the token list `(3)`. Using the grammar in `(1)`, one can successfully expand `BeiGA` and obtain the structure in `(4)`.

All literal tokens being covered by category `BeiA` are collected in the accumulator of `BeiA`. Note that neither the semicolon nor the dot are added to the accumulators of the categories `BeiGA` or `BeiA`. This is due to the fact that, on the one hand, XCode tokens like `XPSEM` are, by default, not added to accumulators and that, on the other hand, the dot is explicitly marked with the '^' character not to be added.

4.3.8 Applying the '$' Operator

The application of the *$*-operator can be illustrated by the expansion rule for the category `LZGA` (*Lemmazeichengestaltangabe*) which denotes items giving the form of the lemma sign.

```
LZGA  ->  XFTxb: $ [ ","^ ].
```

Category `LZGA` expands to a *single* literal token followed by an optional comma. The comma is an SI^{ind}_{ntyp} with regard to the DUDEN-Stilwörterbuch and is not assigned to an XCode. In order not to be added to the accumulator, the comma must be marked with the '^' character.

Note that `LZGA` can be expanded only if the typeface indicator is in the state 'extra bold' at the time of the expansion. This is expressed by the typeface condition `XFTxb:`.

4.3.9 Applying XCode Exceptions

The colon is a SI^{ind}_{ntyp} with respect to the DUDEN-Stilwörterbuch which, in most cases, functions as SI_{ntyp} separating items or groups of items from each other. For this reason, the colon is assigned to its corresponding XCode `XPCOL`. In particular cases only, the colon occurs as part of an example group, like in the example sentence '**die beiden Mannschaften trennten sich 0:0;**', being part of the dictionary entry for '**trennen**'.

To cope with these cases, the XCode `XPCOL` may be disabled by an XCode exception in the scope of category `BeiA`. Herein, the colon is generated as a literal token ':' rather than as an XCode `XPCOL`.

In the following example, semicolons and parentheses are considered to be SI^{def}_{ntyp} and are therefore assigned to their corresponding XCodes. The colon, as well, is considered to be a SI^{ind}_{ntyp} and is assigned to the XCode `XPCOL`. Category `BeiGA` can then be derived using the simplified grammar in `(1)`:

```
(1)  BeiGA  ->  BeiA < XPSEM, BeiA > [ "."^ ] .

     BeiA   ->  -XPCOL:
                [ PragA ] $$.

     PragA  ->  +XPCOL:
                XBRPO $$ XBRPC.

(2)  (sl.) This is example 1;
     The teams left 0:0;
     This is example 3.

(3)  [XPBRPO] [sl] [.] [XBRPC]
     [This] [is] [example] [1] [XPSEM]
     [The] [teams] [left] [0] [:] [0] [XPSEM]
     [This] [is] [example] [1] [.]

(4)  +--> BeiGA:
          +--> BeiA: "This is example 1"
             +--> PragA: "sl."
          +--> BeiA: "The teams left 0:0"
          +--> BeiA: "This is example 3"
```

Category `BeiGA` expands to a sequence of at least one `BeiA` (*Beispielangabe*). Category `BeiA` expands to an optional category `PragA` (*Pragmatische Angabe*) and the $\$\$$-operator. Within the scope of `BeiA`, the XCode `XPCOL` is disabled by the XCode exception `-XPCOL:` in the second rule of `(1)`. Within the scope of category `PragA`, however, the colon functions as SI_{ntyp}, therefore it must be enabled by the XCode exception `+XPCOL:` in the third rule of `(1)` (as well as in other rules for daughter categories of `BeiA`).

The input `(2)` results in the token list `(3)`. Using the grammar in `(1)`, one can successfully expand `BeiGA` and obtain the structure given in `(4)`.

5 The Implementation

LexParse Version 2.501 is implemented in the C++ language and was developed using the BORLAND C++ Compiler for OS/2 Version 1.0. This compiler supports the full AT&T C++ standard version 2.1 (incl. streams and templates). The source code is compatible with the BORLAND C++ Compiler for DOS version 3.1 and with the GNU C++ Compiler version 2.4.1.

The development took place on an IBM compatible PC with Intel '486 CPU under OS/2™2.0. From the beginning, the program was designed to be easily ported to other platforms. The program currently runs under the MS-DOS, OS/2™, and UNIX (SUN OS 4.1) operating systems.

The LexParse program consists of several modules. Each module defines at least one *object*. Each object allows several operations to be applied, the so-called *methods*. The set of methods which may be applied from outside of a module are called *interface methods*. An interface method consists of a name, an arity or argument list, a return value and — sometimes — of a precondition.

The object-oriented design has several advantages: Modules are easier to maintain and may be improved or replaced subsequently as long as the interface methods do not change. Furthermore, this design supports concepts such as data abstraction/encapsulation and structural/behavioral inheritance.

In the following, we give an overview of all program modules, the objects they define and their most important interface methods.

5.1 Module LEXPARSE

This module contains the main routine which initializes data structures and installs signal and error handlers. It reads and interpretes the configuration file by consulting the object *inifile*. The grammar as described in the configuration file is passed to the object *grammar*. The main routine then opens the file containing the typesetting tape and executes a loop in which the parsing engine reads the input file and identifies consecutive entries according to the user-supplied grammar (cf. module PARSE). A special mechanism detects endless loops and aborts the program upon detecting one. All parse trees being generated by the parsing engine and being stored in the object *output* are sent to the output file if the corresponding entry is well-formed. If an entry is non-wellformed or if it is marked as being so, it is sent to the error output file.

The main routine counts all well-formed and non-wellformed entries and runs a timer to measure the time elapsed for parsing. At the end of the parsing process, the program generates a status report to inform the user about the process.

5.2 Module GRAMMAR

This module receives the grammar from the configuration file, converts each rule and stores all rules in the object *grammar*. The object *grammar* consists of an object 'starting symbol' and a set of objects 'non-terminals'. Each object 'non-terminal' consists of a list of objects 'alternate rules'. Each object 'alternate rule' consists of a list of objects 'nodes'. Each object 'node' corresponds to a specific element of the language as being defined by the grammar.

The following table gives an overview of all node objects which are distinguished by the object *grammar* and which must be treated by the parsing engine according to their semantics.

Table of all Grammar Nodes

Node	Function
NodeError	Error directive (`@`)
NodeXCode	XCode
NodeCategory	Category
NodeLitSingle	A single literal (`$`-operator)
NodeLitMultiple	A sequence of literals (`$$`-operator)
NodeLitMultExcl	A sequence of literals (`$\`-operator)
NodeLitNumerical	Numerical literal (`#`-operator)
NodeLitExpect	A specific literal (string constant)
NodeLitRep	A specific literal (regular expression)
NodeSkipIncl	Inclusive skip operator (`*`)
NodeSkipExcl	Exclusive skip operator (`*\`)
NodeExcept	Except operator (`\`)
NodeTFstate	Typeface state
NodeTFstateSet	Set a typeface state
NodeOptional	Optional rule
NodeOptionalEnd	End of an optional rule
NodeMulOptional	Multiple optional rule
NodeMulOptionalEnd	End of a multiple optional rule
NodeSet	Set of terminals
NodeSetEnd	End of a set of terminals
NodeCounter	Counter
NodeResetCounter	Reset Counter

The object *grammar* does not rearrange grammar rules, i.e., *grammar* does not transform an optional rule as specified under `(1)` into a pure context-free rule as specified under `(2)` by generating an additional category `AUX`.

```
(1)  A    ->   [ X ].
```

```
(2) A    ->  AUX.
    AUX  ->  X
         |   .
```

This forces the parsing engine to handle all specified grammar nodes, especially the nodes `NodeOptional` and `NodeMulOptional`. This procedure is taken up with a view to greater clarity: The parser does not automatically generate new categories which might confuse a user on reading the logfiles during the development (or debugging) of a grammar. The grammar may be read in the logfiles as it was specified by the user.

The object *grammar* offers several interface methods such as `add_rule/2`, `add_label/2` and `add_start/1` with which rules, labels or a starting symbol may be added to the grammar. The object *grammar* may be passed to other modules as a constant object, i.e., a 'read-only' object which cannot be modified.

5.3 Module EXPORT

This module implements the object *Export* for displaying the LexParse using other grammar formalisms. Currently the formalism EBNF (*Extended Backus Naur Form*[36]) and the formalism SGML DTD (*Document Type Declaration*[37]) are supported. The object *Export* may be sent to an output stream which can either be the console or a file on which it writes the complete LexParse grammar as being stored by object *grammar* using the specified formalism.

5.4 Module PPROCESS

This module implements the generic object *pprocess* for replacing string patterns. The preprocessor and the postprocessor are derived from this object. The object supports six different actions to be applied on a string:

- Delete a specific pattern within a string.
- Delete a specific pattern within a string in which another specific pattern could be found.
- Delete the whole string in which a specific pattern could be found.
- Replace a specific pattern by another pattern within a string.
- Replace a specific pattern by another pattern within a string in which a third specific pattern could be found.
- Delete a specific range of columns from a string.

[36] For details about the EBNF formalism refer to [15].
[37] For details about the SGML DTD formalism refer to [6].

The patterns specified for an action may be either string constants or regular expressions.

pprocess offers the interface method `add_action/2` with which complete actions may be specified.

5.5 Module PREP

This module contains the preprocessor (object *prep*) for the parsing engine. The preprocessor reads the input file (*input stream*) on a line-by-line basis and is able to apply seven different kinds of actions (*directives*) to the input stream:

Delete: Delete a specific pattern in the input line.

DeleteRep: Delete a specific regular expression in the input line.

DelLine: Delete the whole input line in which the specified pattern could be found.

DelLineRep: Delete the whole input line in which the specified regular expression could be found.

Change: Replace a specific pattern by another pattern within the input line.

ChangeRep: Replace a specific regular expression by a replace pattern within the input line.

DeleteCol: Delete a specific range of columns from the line.

Note: The order of all directives as specified in the configuration file is preserved during execution.

The `Change` directive offers the possibility of inserting XCode tokens into the input stream. For doing so, the desired XCode must be enclosed by two '`#`' characters.

```
Change = "&12&34" -> "#XFLBE# #XFTxb#"
```

prep offers the interface method `add_action/2` with which complete directives may be specified.

The interface method `get_next/1` returns the next line as being read from the input file and being worked on by the preprocessor.

5.6 Module SCAN

This module contains the scanner (or tokenizer) of the parser. The scanner object *scan* receives lines of characters from the preprocessor and splits them into separate tokens. The scanner distinguishes two classes of tokens: XCode tokens and literal tokens.[38]

By configuring the scanner, each XCode may be defined by specifying a pattern which is replaced by an XCode token if it occurs in the input. The object *scan* supports a large table of XCodes which all can be connected with a pattern.[39]

scan offers the interface method `add_xcode/2` to define an XCode together with its pattern.

The interface method `lookahead/1` tells the caller the next token, that is available in the input. The token is not removed from the stream so that subsequent calls to `lookahead/1` all result in the same token.

`get/1` reads the next token from the input stream. The token is removed from the stream.

`unget/1` puts the specified token back onto the input stream, so that the next call to `get/1` results in this token. There are no limits concerning the number of calls to `unget/1`.

The object *scan* also supports several interface methods to expand abbreviations: `expand_define/2` allows the definition of a string which is returned as a literal token whenever a sequence of the initial letter of this string followed by a dot occurs in the input stream. This expansion must be enabled using `expand_enable/0` before any calls to `get/1` and may be disabled using method `expand_disable/0` after such a call.

5.7 Module PARSE

This module contains the object *parse* representing the parsing engine. The parsing engine consults the grammar representation stored in the object *grammar* and reads tokens from the scanner. All categories which can be expanded are sent to the object *output* in the module OUTPUT.

The parsing engine uses the algorithm of recursive descent parsing. This is a *top-down, depth-first* algorithm which heavily uses backtracking, i.e., reading and consulting the tokens repeatedly.[40]

parse offers the interface methods `accept/1` and `accept_eof/0`. `accept/1` is a polymorphic function and may be executed with the following different arguments: grammar, category, rule and grammar node. Both methods have a

[38] For a complete discussion of these different tokens refer to section 3.2 (p. 12).

[39] A list of all supported XCodes can be found in section A (p. 59) in the appendix.

[40] For a detailed discussion of this refer to Aho, Sethi & Ullman 1988 [1] p. 181 ff.

boolean return value indicating whether the specified argument could be accepted or not. This enables the following main loop for parsing:

```
WHILE ( parser.accept_eof() = false ) DO
      result := parser.accept( Grammar )
      ...
      display parse tree
      ...
      END
```

5.8 Module OUTPUT

This module contains the object *output* which is capable of storing and displaying the generated parse trees. The object *output* supports many switches and options to control the style of display not to be mentioned here in more detail.

output offers the interface methods `cat_announce/3` (call whenever a category is predicted by the parsing engine) and `cat_complete/1` (call whenever a category could be completed by the parsing engine) to construct the parse tree. Each time a category has to be removed from the parse tree (e.g. because of backtracking) the method `cat_undo/1` may be called to remove subtrees from the whole parse tree. The complete parse tree may be displayed by sending the object *output* to an output stream (e.g. logfile or console).

The object *output* is partly derived from the object *pprocess* which represents the postprocessor of the output.

5.9 Module ERROR

This module contains the error handler of the LexParse program. The object *error* distinguishes between six different error classes.

Panic: Serious error — the program terminates immediately. Output files and logfiles are unusable.

Critical: Serious error — the program terminates immediately. Output files and logfiles may be incomplete.

Internal: Internal program error — the program terminates immediately. Output files and logfiles may be evaluated.

Fatal: Program error — the program terminates immediately. Output files and logfiles may be evaluated.

Normal: Program error — the program terminates later on because of this error condition. Output files and logfiles may be evaluated.

Warning: Warning message only — the program does not terminate because of this message. Output files and logfiles may be evaluated completely.

error offers the interface method `add_error/6` with which an error message and an error condition may be raised. `exec/0` displays all messages accumulated so far and executes the corresponding action, e.g., the program termination if a normal error was generated.

Each error message consists of an error class (see above), the name of the module in which the error occured, an error code, an error description and an error location.

The error code contains four digits whereby the first two digits identify the module in which the error occured:

Error Code Classes	
Error codes	Error occured in module:
00xx	Could not identify module or main
01xx	Error in module LEXPARSE
02xx	Error in module PREP
03xx	Error in module SCAN
04xx	Error in module PARSE
05xx	Error in module GRAMMAR
06xx	Error in module OUTPUT
07xx	Error in module INI
08xx	Error in module PPROCESS
09xx	Error in module GREP
10xx	Error in module STATES

5.10 Module GREP

This module contains the object *grep*, a generic object to search for regular expressions in strings.

5.11 Module INI

This module contains the object *inifile* handling the `.INI` configuration files. The object offers several interface methods to access the sections and options of the configuration file.

5.12 Module TOOLS

This module contains additional generic functions, e.g., string functions and (simple) user interface functions.

6 Conclusion and Outlook

The LexParse program is a powerful tool for dictionary entry parsing and the developing of dictionary entry grammars. The grammar formalism offers various features to meet the specific needs of dictionary entry parsing, e.g., the handling of structure indicators, font codes and counters. LexParse promotes a quick development of dictionary entry grammars providing a rich set of facilities for identifying inadequate grammars as well as detecting non-wellformed dictionary entries. The parse trees being generated by the LexParse program may be displayed in different, user-definable styles, offering both easy-to-read and comprehensive graphical representations and tagged structures for a computer evaluation.

Depending on the complexity and the size of the grammar and the entries, the parser runs at a speed of up to 10 entries per second on a PC with '486/33 CPU under the OS/2™ operating system.

The program was tested on several dictionaries: a grammar was developed for the Duden-Stilwörterbuch at the Department of Linguistics at the University of Tübingen. This grammar covers over 98% of all entries.[41] A grammar for the Duden-Bedeutungswörterbuch is currently being worked out at the same institute. Another LexParse grammar is being developed within the project "Deutsches Rechtwörterbuch" carried out at the Academy of Science in Heidelberg. In the German Department of the University of Heidelberg work has begun in elaborating a grammar for the "Frühneuhochdeutsches Wörterbuch".[42]

We have several plans for enhancing the LexParse system in the future:

- We are aiming at an integration with the *ELWIS*-TOOLBOX — a collection of different programs for intellectually analyzing dictionary entry structures.
- An elegant and efficient handling of nesting and niching dictionaries will be integrated into the system.
- An interactive debugger facilitating an easier development of dictionary grammars will soon be added to the program.

We hope that the experiences gathered in future applications of the system on varied types of dictionaries will assist us in further developing LexParse into a flexible, efficient, multi-task tool in dictionary entry parsing.

[41] About 10% of the entries were non-wellformed, i.e., structure indicators were absent or counters were invalid. Cf. Engelke 1994 [8].

[42] As far as can be seen right now, both grammars will cover distinctly more than 90% of all entries.

7 References

References

[1] **Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman (1988):** *Compilers, Principles, Techniques and Tools.* Reading, Massachusetts: Addison Wesley.

[2] **Brigitte Bläser & Matthias Wermke (1990):** *Projekt 'Elektronische Wörterbücher / Lexika' Abschlußbericht der Definitionsphase.* IWBS Report 145. Wissenschaftliches Zentrum / Institut für Wissensbasierte Systeme der IBM Deutschland GmbH.

[3] **Christoph Bläsi & Heinz-Detlev Koch (1991):** *Maschinelle Strukturerschließung von Wörterbuchartikeln mit Standardmethoden.* In: *Lexicographica.* Vol. 7, (1991) 1992, pp. 182–201. Tübingen: Niemeyer.

[4] **Bran Boguraev (1991):** *Building a Lexicon: The Contribution of Computers.* In: *International Journal of Lexicography* Vol. 4, (1991), pp. 227-260

[5] **Bran Boguraev & Ted Briscoe (Eds.) (1989):** *Computational Lexicography for Natural Language Processing.* London, New York: Longman.

[6] **Martin Bryan (1988):** *SGML — An Author's Guide to the Standard Generalized Markup Language.* Waltham, Massachusetts: Addison Wesley.

[7] **Günther Drosdowski (Ed.) (1988):** *Duden Stilwörterbuch der deutschen Sprache.* Mannheim: Duden.

[8] **Sabine Engelke (1994, forthcoming):** *Grammatikentwicklung für Wörterbücher mit dem* LEXPARSE *Formalismus.* SfS Report, Universität Tübingen.

[9] **Ralf Hauser (1993):** LEXPARSE *User's Manual.* SfS Report 10-93, Seminar für Sprachwissenschaft der Universität Tübingen.

[10] **Ralf Hauser (1993):** *Das Parsen von Wörterbüchern. Problematik und Lösungsansätze.* Magisterarbeit am Seminar für Sprachwissenschaft der Universität Tübingen.

[11] **Ralf Hauser & Angelika Storrer (1993):** LEXPARSE — *Ein Parser zur maschinellen Analyse von Wörterbuchstrukturen.* SfS Report 9-93, Seminar für Sprachwissenschaft der Universität Tübingen.

[12] **Ralf Hauser & Angelika Storrer (1994):** *Dictionary Entry Parsing Using the* LEXPARSE *System.* In: *Lexicographica.* Vol. 9, (1993) 1994, pp. 174–219. Tübingen: Niemeyer.

[13] **Franz Josef Hausmann, Oskar Reichmann, Herbert Ernst Wiegand & Ladislav Zgusta (Eds.) (1989):** *Wörterbücher. Ein internationales Handbuch zur Lexikographie.* Berlin: de Gruyter.

[14] **Franz Josef Hausmann & Herbert Ernst Wiegand (1989):** *Component Parts and Structures of General Monolingual Dictionaries: A Survey.* In: *Wörterbücher. Ein internationales Handbuch zur Lexikographie.* Edited by Franz J. Hausmann, Oskar Reichmann, Herbert E. Wiegand & Ladislav Zgusta. Berlin: de Gruyter, Vol. 1, pp. 328–360.

[15] **K. Jensen & N. Wirth (1985):** *Pascal User Manual and Report.* Berlin, New York: Springer.

[16] **Mary S. Neff & Branimir K. Boguraev (1990):** *From Machine-Readable Dictionaries to Lexical Data Bases.* Research Report RC #16080 (71353) 8/31/90, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

[17] **Mary S. Neff, R. J. Byrd & O. A. Rizk (1988):** *Creating and querying lexical data bases.* In: *Proceedings of the Second ACL Conference on Applied Natural Language Processing* Austin, Texas; pp. 84–92.

[18] **Angelika Storrer (1994, forthcoming):** *Methodisches Vorgehen beim Entwickeln von Artikelstrukturgrammatiken.* SfS Report, Universität Tübingen.

[19] **Angelika Storrer, Helmut Feldweg & Erhard Hinrichs (1993):** *Korpusunterstützte Entwicklung lexikalischer Wissensbasen. Sprache und Datenverarbeitung 1-2, 1993* Vol. 17, pp. 59–72.

[20] **Herbert Ernst Wiegand (1989a):** *Aspekte der Makrostruktur im allgemeinen einsprachigen Wörterbuch: alphabetische Anordnungsformen und ihre Probleme.* In: *Wörterbücher. Ein internationales Handbuch zur Lexikographie.* Edited by Franz J. Hausmann, Oskar Reichmann, Herbert E. Wiegand & Ladislav Zgusta. Berlin: de Gruyter, Vol. 1, pp. 371–409.

[21] **Herbert Ernst Wiegand (1989b):** *Der Begriff der Mikrostruktur: Geschichte, Probleme, Perspektiven.* In: *Wörterbücher. Ein internationales Handbuch zur Lexikographie.* Edited by Franz J. Hausmann, Oskar Reichmann, Herbert E. Wiegand & Ladislav Zgusta. Berlin: de Gruyter, Vol. 1, pp. 409–462.

[22] **Herbert Ernst Wiegand (1989c):** *Formen von Mikrostrukturen im allgemeinen einsprachigen Wörterbuch.* In: *Wörterbücher. Ein internationales Handbuch zur Lexikographie.* Edited by Franz J. Hausmann, Oskar Reichmann, Herbert E. Wiegand & Ladislav Zgusta. Berlin: de Gruyter, Vol. 1, pp. 462–501.

[23] **Herbert Ernst Wiegand (1989d):** *Die deutsche Lexikographie der Gegenwart.* In: *Wörterbücher. Ein internationales Handbuch zur Lexikographie.* Edited by Franz J. Hausmann, Oskar Reichmann, Herbert E. Wiegand & Ladislav Zgusta. Berlin: de Gruyter, Vol. 2, pp. 2100–2246

[24] **Herbert Ernst Wiegand (1991):** *Printed Dictionaries and their Parts as Text. An Overview of More Recent Research as an Introduction* In: *Lexicographica.* Vol. 6, (1990) 1991, pp. 1–124. Tübingen: Niemeyer.

Appendixes

A List of All Available XCodes

The following table contains a list of all XCodes available in the LexParse system together with a description of their semantics.[43]

Note: All pre- and userdefined XCodes are case sensitive.

List of All Available XCodes		
XCode	Character	Description
XBRPO	(	BracketParenthesisOpen
XBRPC	)	BracketParenthesisClose
XBRCB	{	BracketCurlyOpen
XBRCE	}	BracketCurlyClose
XBRAO	<	BracketAngleOpen
XBRAC	>	BracketAngleClose
XBRSO	[	BracketSquareOpen
XBRSC	]	BracketSquareClose
XPDOT	.	PunctuationDot
XPCOM	,	PunctuationComma
XPSEM	;	PunctuationSemicolon
XPCOL	:	PunctuationColon
XPEXC	!	PunctuationExclamationMark
XPQUE	?	PunctuationQuestionMark
XPHYP	-	PunctuationHyphen
XPRDT	•	PunctuationRoundedDot
XPCDT	·	PunctuationCenteredDot
XPAST	*	PunctuationAsterisk
XCSLA	/	CharacterSlash
XCBSL	\	CharacterBackslash
XCAMP	&	CharacterAmpersand
XCPAR	§	CharacterParagraph
XCSEP	|	CharacterSeparator
XCARR	^	CharacterArrow
XCQUO	``	CharacterQuoteOpen
XCQUC	''	CharacterQuoteClose
		continued on next page ...

[43] The list of all available XCodes may be extended in future versions of LexParse.

. . . continued from previous page

XCSCO	_	CharacterUnderscore
XCTIL	~	CharacterTilde
XCACC	'	CharacterAccent
XCATS	@	CharacterATSign
XFLBE		FormatLemmaBegin
XFLEN		FormatLemmaEnd
XFPBE		FormatParagraphBegin
XFPEN		FormatParagraphEnd
XFTst		FormatTypefaceStandard
XFTit		FormatTypefaceItalic
XFTsl		FormatTypefaceSlanted
XFTti		FormatTypefaceTiny
XFTca		FormatTypefaceCapital
XFTtt		FormatTypefaceTypewriter
XFTsp		FormatTypefaceSuperscript
XFTsb		FormatTypefaceSubscript
XFTbo		FormatTypefaceBold
XFTxb		FormatTypefaceExtraBold
XFTwi		FormatTypefaceWide
XFTf0		FormatTypefaceFont0
XFTf1		FormatTypefaceFont1
XFTf2		FormatTypefaceFont2
XFTf3		FormatTypefaceFont3
XFTf4		FormatTypefaceFont4
XFTf5		FormatTypefaceFont5
XFTf6		FormatTypefaceFont6
XFTf7		FormatTypefaceFont7
XFTf8		FormatTypefaceFont8
XFTf9		FormatTypefaceFont9
X_TAB		File: Tabulator
X__NL		File: Newline (LF/CR)
X_EOF		File: EndOfFile

Note: The semantics of XCodes of classes `XF***` (Format) and `XU***` (User-definable) cannot be expressed by a character.

B A Sample .INI Configuration File for LexParse

```
;
;   General INI file for LEXPARSE
;
;   sections PREPROCESS, SCANNER, PARSER
;   and GRAMMAR are exported to include file

include     = STILWORT.19    ; grammar

[Settings]
; Debug      = A S P I G O
TypefaceStates = On
PostProcess = On
PreProcess = On
SkipNLChar = Yes
; StepEntry  = Yes
; StepSingle = Yes
FirstEntry = 3
LastEntry  = 9

[EBNF]
file        = ebnf.asc
Header      = "\tExtended Backus Naur Format:\n\n"
;Terminals  = Yes
;Operators  = Yes
Errorhandlers = No
Labels      = Yes
Comment     = "*** "
Indent      = 4

[SGML-DTD]
file        = sgml-dtd.asc
Header      = "\t<!ELEMENT   Dictionary  (WA+)>\n\n"
Footer      = "\n"
VirtualCategories = No
Indent      = 4

[Logfiles]
Comment     = '#'
preprocess = c:\tmp\prep.log
scan        = c:\tmp\scan.log
parse       = c:\tmp\parse.log
output      = parsed\output.log
error       = parsed\error.log
ErrorsOnly = Yes
;LogGrammar = Yes
;LogConfig  = Yes
OutputNoHeader = Yes

[Display]  ; for a display in TREE style
Mode        = T
```

```
Title      = Yes
Unlabeled  = Yes
EmptyCat   = Yes
ToParseLog = Yes
Parselevel = Yes
;AddSpaces  = Yes
Header     = "--- ParseTree %# ---\n"
;Footer     = "\n\n"
Tree       = " | "
TreeS      = " +--> "
TreeLD     = " +--> "
RMargin    = 79   ; for 80 column display
EntryNum   = No
Format     = "%c[ - %l][ \"%$\"]\n"
Progress   = Off
```

C A Sample LexParse Grammar for the Duden-Stilwörterbuch

The following sample grammar describes a very small fragment of the Duden-Stilwörterbuch.

Please note that due to restrictions in the TEX word processing system some of the escape sequences — especially German Umlaute — may not appear in the listing. Refer to the STILWORT.INI file of the installation disk for a complete listing with all escape sequences.

```
[PreProcess]
ChangeRep = "}}$" -> " "  ; end-of-line marker
; begin of a lemma and boldface
Change    = "T25" -> "#XFLBE# #XFTxb#"
DeleteRep = "}-}$"        ; hyphen
Delete    = "vr10"      ; italic correction

[Scanner]
AddLetters = "ÄÖÜäöüß"
XPSEM   = ";"
XPCOL   = ":"
XFLEN   = ""
; control codes for typfaces ('Settings:TypefaceStates')
XFTst   = "1"
XFTit   = "2"
XFTbo   = "3"
XFTxb   = "5"

[Parser]
Start   = WA              ; start symbol for grammar
IndentLog = On
RecoverCounterError = Yes
ExpandAbbrev = LZGA       ; category containing LZGA
```

```
SkipCriticalIsError = Yes

[Grammar]
;--- Wörterbuchartikel
WA      -> XFLBE, FK, SK, XFLEN
        |  WA_Err.                ; error handler

WA_Err  -> * XFLEN @.             ; error handler

;--- Formkommentar
FK      -> LZGA [ GrA ] XPCOL
        |  FK_Err.                ; error handler

FK_Err  -> * XPCOL @.             ; error handler

LZGA    -> XFTxb: $ [ ","^ ].     ; Lemmazeichengestaltangabe

GrA     -> GA.                    ; Grammatische Angabe

GA      -> XFTst: { "der", "die", "das" }.

;--- Semantischer Kommentar
SK      -> < PAA, PragSemA, BeiGA >.

;--- Typen von Polysemieangaben
PAA     -> XFTbo: %1, ".", %Ra.   ; Polysemieangabe (arabisch)

;--- Pragmatisch Semantische Angabe
PragSemA-> BPA.

;--- Bedeutungsangaben
BPA     -> XFTit: * XPCOL.

;--- Beispielgruppenangabe
BeiGA   -> BeiA < XPSEM, BeiA > [ "." ].

;--- Beispielangabe
BeiA    -> -XPCOL:                ; no XCodes XPCOL herein!
           XFTst: $$.

[Labels]
WA       = "Wörterbuchartikel"
WA_Err   = "Fehler in Wörterbuchartikel"
FK       = "Formkommentar"
FK_Err   = "Fehler in Formkommentar"
LZGA     = "Lemmazeichengestaltangabe"
GA       = "Genusangabe"
GrA      = "Grammatische Angabe"
SK       = "Semantischer Kommentar"
PAA      = "Polysemieangabe (arabisch)"
PragSemA = "Pragmatisch-Semantische Angabe"
BPA      = "Bedeutungsparaphrasenangabe"
BeiGA    = "Beispielgruppenangabe"
BeiA     = "Beispielangabe"
```

D DUDEN-Stilwörterbuch Grammar in EBNF Style

The following describes the grammar for the DUDEN-Stilwörterbuch in EBNF (*Extended Backus Naur Form*) notation as being generated by the LEXPARSE program.[44]

```
*** Wörterbuchartikel
WA     = FKw [ SKw ]
       | FK SK [ PKP ]  .

*** Formkommentar zur Wendung
FKw    = LZGA [ PragA ] WendA WBA  .

*** Semantischer Kommentar zur Wendung
SKw    = BPAw [ BeiGA ]  .

*** Formkommentar
FK     = LZGA [ VerwA ] [ PragA ] [ LZGA ] { GrA } [ PragA ] [ VerwA ]  .

*** Semantischer Kommentar
SK     = SSK0 { SSK0 }
       | SSK1 { SSK1 }
       | SSK2 { SSK2 }
       | [ PragSemA ] [ BeiGA ] { KSV [ BeiGA ] }  .

*** Postkommentar zur Phraseologie
PKP    = SKP { SKP }  .

*** Wendungsbildungsangabe
WBA    = [ PragA ] [ GrA ]
       | [ GZB ] [ BeiGA ] { [ GZB ] [ BeiGA ] }  .

*** Beispielgruppenangabe
BeiGA  = [ UGrA ] BeiA { [ UGrA ] BeiA }  .

*** Grammatische Angabe
GrA    = GA
       | WAA
       | SmA
       | UGrA  .

*** Semantischer Subkommentar 0. Stufe
SSK0   = PAR [ PragSemA ] [ BeiGA ] { KSV [ BeiGA ] } { SSK1 }  .

*** Semantischer Subkommentar 1. Stufe
SSK1   = PAA [ PragSemA ] [ BeiGA ] { KSV [ BeiGA ] } { SSK2 }  .

*** Semantischer Subkommentar 2. Stufe
SSK2   = PAB [ PragSemA ] [ BeiGA ] { KSV [ BeiGA ] }  .
```

[44] The command line option `/ebnf` does generate such a display.

```
*** Pragmatisch-Semantische Angabe
PragSemA = [ PragA ] [ GrA ] [ PragA ] [ AzF ] [ BA ]  .

*** Kommentar zur speziellen Verwendung
KSV    = SKSV { SKSV }  .

*** Subkommentar zur Phraseologie
SKP    = [ PragA ] PhrasA { KPB }  .

*** Glossat zur Bedeutung
GZB    = [ GZP ] [ GZP ]  .

*** Beispielangabe
BeiA   = [ BWA ] [ SAA ] [ PragA ] { PragA } { GZB } [ SAA ]  .

*** Bedeutungsangabe
BA     = BPA
       | HypA  .

*** Subkommentar zur speziellen Verwendung
SKSV   = VSA [ PragA ] [ BPA ] { BeiGA }  .

*** Phrasemangabe
PhrasA = [ PragA ]  .

*** Kommentar zur Phrasembedeutung
KPB    = [ PAB ] PBA [ BeiGA ]  .

*** Sprechaktangabe
SAA    = [ GZP ]
       | [ GZP ]  .

*** Verwendungsspezifizierende Angabe
VSA    = MA
       | RWA
       | WKA [ WKA ]
       | FGA  .

*** Phrasembedeutungsangabe
PBA    = [ GZP ]  .
```

E Sample Sessions

In the following we present several sample sessions of running the LexParse program to parse dictionary entries being part of a typesetting tape for the Duden-Stilwörterbuch.

E.1 'Treffen', DUDEN-Stilwörterbuch, Page 703

The following dictionary entry was parsed by LEXPARSE successfully. The parse tree for the entry Treffen is complete. Abbreviated lemma signs were expanded automatically by the program.

```
[D:\FOR\SFS\LEXPARSE]lexparse /entry -1 /nowarn lexparse.ini wbs\9-entries.dat

 LEXPARSE V1.208 ß-Code   [Jul  1 1994]   <DEBUG-SRC>   A Dictionary Parser

 (C) 1992-1994 Seminar fuer Sprachwissenschaft at the University of Tuebingen
 (C) 1992-1994 Ralf Hauser
     Created by 'Affie' (k.f. c|o)

 LEXPARSE: Reading configuration: lexparse.ini
 LEXPARSE: Grammar checked and OK.

 Inputfile..........: wbs/9-entries.dat
 Parsing............: Enabled
 Grammar nodes......: 514
 First entry........: 1
 Last entry.........: 1
 Time of day........: Fri Jul  1 10:54:07 1994
----------------------------------------------------------------------
----- ParseTree 1 -----
WA (Wörterbuchartikel)
 +--> FK (Formkommentar)
 |  +--> LZGA (Lemmazeichengestaltangabe): "Treffen"
 |  +--> GrA (Grammatische Angabe)
 |     +--> GA (Genusangabe): "das"
 +--> SK (Semantischer Kommentar)
 |  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
 |  |  +--> PAA (Polysemieangabe (arabisch)): "1."
 |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
 |  |  |  +--> BA (Bedeutungsangabe)
 |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "Zusammenkunft, Begegnung"
 |  |  +--> BeiGA (Beispielgruppenangabe)
 |  |     +--> BeiA (Beispielangabe): "regelmäßige, seltene Treffen"
 |  |     +--> BeiA (Beispielangabe): "ein Treffen der Abiturienten"
 |  |     +--> BeiA (Beispielangabe): "ein Treffen der Außenminister"
 |  |     +--> BeiA (Beispielangabe): "ein Treffen verabreden, veranstalten"
 |  |     +--> BeiA (Beispielangabe): "an einem Treffen teilnehmen"
 |  |     +--> BeiA (Beispielangabe): "zu einem Treffen kommen."
 |  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
 |  |  +--> PAA (Polysemieangabe (arabisch)): "2."
 |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
 |  |  |  +--> PragA (Pragmatische Angabe): "militär. veraltet"
 |  |  |  +--> BA (Bedeutungsangabe)
 |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "Gefecht"
 |  |  +--> BeiGA (Beispielgruppenangabe)
 |  |     +--> BeiA (Beispielangabe): "frische Truppen ins Treffen führen."
 |  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
 |     +--> PAA (Polysemieangabe (arabisch)): "3."
 |     +--> PragSemA (Pragmatisch-Semantische Angabe)
 |     |  +--> PragA (Pragmatische Angabe): "Sport"
 |     |  +--> BA (Bedeutungsangabe)
 |     |     +--> BPA (Bedeutungsparaphrasenangabe): "Wettkampf"
 |     +--> BeiGA (Beispielgruppenangabe)
 |        +--> BeiA (Beispielangabe): "ein faires, spannendes Treffen"
 |        +--> BeiA (Beispielangabe): "das Treffen endete unentschieden"
 |        +--> BeiA (Beispielangabe): "sie konnte das Treffen für sich entscheiden."
 +--> PKP (Postkommentar zur Phraseologie)
    +--> SKP (Subkommentar zur Phraseologie)
       +--> PragA (Pragmatische Angabe): "geh."
       +--> PhrasA (Phrasemangabe): "etwas ins Treffen führen"
       +--> KPB (Kommentar zur Phrasembedeutung)
```

```
        +--> PBA (Phrasembedeutungsangabe): "etwas als Argument vorbringen"

 Entry #1: Accepted!
----------------------------------------------------------------------
 Inputfile..........: wbs/9-entries.dat
 First entry........: 1
 Last entry.........: 1
 Entries parsed.....: 1
 Errors.............: 0
 Time of day........: Fri Jul  1 10:54:07 1994
 Time of start......: Fri Jul  1 10:54:07 1994
 Elapsed time.......: 00h 00m 00s
 Average parse time.: < 1 sec.

 LEXPARSE V1.208 ß-Code   [Jul  1 1994]   <DEBUG-SRC>
       Program terminated successfully

 LEXPARSE: (W-105) Warning occurred in module 'LEXPARSE':
       Parsing terminated due to 'LastEntry' specification
```

E.2 'trennen', DUDEN-Stilwörterbuch, Page 704

The following dictionary entry was parsed by LEXPARSE successfully. The parse tree for the entry trennen is *not* complete. Abbreviated lemma signs were expanded automatically by the program.

The entry is not well–formed since the item *Beispielangabe* (BeiA) 'ein Gebirgszug trennt das Land in zwei Regionen.' does not contain a trailing semicolon although there are more categories BeiA.

LEXPARSE generated a maximum parse tree and all tokens which could be consumed by the parser following the error were collected in the accumulator of category *Fehler in Wörterbuchartikel* (WA_Err).

```
[D:\FOR\SFS\LEXPARSE]lexparse /entry 7-7 /nowarn lexparse.ini wbs\9-entries.dat

 LEXPARSE V1.208 ß-Code   [Jul  1 1994]   <DEBUG-SRC>   A Dictionary Parser

 (C) 1992-1994 Seminar fuer Sprachwissenschaft at the University of Tuebingen
 (C) 1992-1994 Ralf Hauser
     Created by 'Affie' (k.f. c|o)

 LEXPARSE: Reading configuration: lexparse.ini
 LEXPARSE: Grammar checked and OK.

 Inputfile..........: wbs/9-entries.dat
 Parsing............: Enabled
 Grammar nodes......: 514
 First entry........: 7
 Last entry.........: 7
 Time of day........: Fri Jul  1 11:10:34 1994

 6 Entries skipped!

----------------------------------------------------------------------
 N O T E: Entry #7 could not be accepted!
----- ParseTree 7 -----
WA (Wörterbuchartikel)
 +--> FK (Formkommentar)
 |  +--> LZGA (Lemmazeichengestaltangabe): "trennen"
 +--> SK (Semantischer Kommentar)
```

```
|  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
|  |  +--> PAA (Polysemieangabe (arabisch)): "1."
|  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |  |  +--> PAB (Polysemieangabe (Buchstabe)): "a"
|  |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  |  |  +--> GrA (Grammatische Angabe)
|  |  |  |  |  +--> SmA (Satzmusterangabe): "jmdn., sich, etwas von jmdm., von etwas/ aus etwa\
                    s trennen"
|  |  |  |  +--> BA (Bedeutungsangabe)
|  |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "lösen, entfernen, abtrennen"
|  |  |  +--> BeiGA (Beispielgruppenangabe)
|  |  |     +--> BeiA (Beispielangabe): "eine Borte vom Kleid, das Futter aus dem Mantel trenn\
                 en"
|  |  |     +--> BeiA (Beispielangabe): "den Kopf vom Rumpf trennen"
|  |  |     |  +--> GZB (Glossat zur Bedeutung): "abschlagen"
|  |  |     +--> BeiA (Beispielangabe): "ein Tier von der Herde trennen"
|  |  |     +--> BeiA (Beispielangabe): "das Kind von seiner Mutter, von seiner Familie trenne\
                 n"
|  |  |     +--> BeiA (Beispielangabe): "das Erz vom Gestein trennen"
|  |  |     +--> BeiA (Beispielangabe): "das Eigelb vom Eiweiß trennen"
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |  |  +--> PAB (Polysemieangabe (Buchstabe)): "b"
|  |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  |  |  +--> GrA (Grammatische Angabe)
|  |  |  |  |  +--> SmA (Satzmusterangabe): "jmdn., etwas trennen"
|  |  |  |  +--> BA (Bedeutungsangabe)
|  |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "auseinanderbringen"
|  |  |  +--> BeiGA (Beispielgruppenangabe)
|  |  |     +--> BeiA (Beispielangabe): "Eigelb und Eiweiß trennen"
|  |  |     +--> BeiA (Beispielangabe): "Sauerstoff und Wasserstoff trennen"
|  |  |     +--> BeiA (Beispielangabe): "die Bestandteile einer Mischung sorgfältig trennen"
|  |  |     +--> BeiA (Beispielangabe): "die Nähte trennen"
|  |  |     +--> BeiA (Beispielangabe): "die Streitenden mußten getrennt werden."
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |  |  +--> PAB (Polysemieangabe (Buchstabe)): "c"
|  |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  |  |  +--> GrA (Grammatische Angabe)
|  |  |  |  |  +--> SmA (Satzmusterangabe): "etwas trennen"
|  |  |  |  +--> BA (Bedeutungsangabe)
|  |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "in seine Bestandteile zerlegen"
|  |  |  +--> BeiGA (Beispielgruppenangabe)
|  |  |     +--> BeiA (Beispielangabe): "ein Kleid trennen"
|  |  |     +--> BeiA (Beispielangabe): "ein Stoffgemisch chemisch, durch Kondensation trennen\
                 "
|  |  |     +--> BeiA (Beispielangabe): "ein Wort nach Silben trennen"
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |     +--> PAB (Polysemieangabe (Buchstabe)): "d"
|  |     +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |     |  +--> GrA (Grammatische Angabe)
|  |     |  |  +--> SmA (Satzmusterangabe): "jmdn. trennen"
|  |     |  +--> BA (Bedeutungsangabe)
|  |     |     +--> BPA (Bedeutungsparaphrasenangabe): "auseinanderreißen"
|  |     +--> BeiGA (Beispielgruppenangabe)
|  |        +--> BeiA (Beispielangabe): "die beiden Geschwister sollten nicht getrennt werden"
|  |        +--> BeiA (Beispielangabe): "der Krieg hat die Familie getrennt"
|  |        +--> BeiA (Beispielangabe): "nichts konnte die Liebenden trennen"
|  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
|  |  +--> PAA (Polysemieangabe (arabisch)): "2."
|  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  |  +--> BA (Bedeutungsangabe)
|  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "klar unterscheiden, auseinanderhalten"
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |  |  +--> PAB (Polysemieangabe (Buchstabe)): "a"
|  |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  |  |  +--> GrA (Grammatische Angabe)
|  |  |  |     +--> SmA (Satzmusterangabe): "jmdn., etwas trennen"
|  |  |  +--> BeiGA (Beispielgruppenangabe)
```

```
|  |  |     +--> BeiA (Beispielangabe): "die Begriffe klar, sauber trennen"
|  |  |     +--> BeiA (Beispielangabe): "wir müssen Person und Sache strikt trennen"
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |     +--> PAB (Polysemieangabe (Buchstabe)): "b"
|  |     +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |     |  +--> GrA (Grammatische Angabe)
|  |     |     +--> SmA (Satzmusterangabe): "jmdn., etwas von etwas trennen"
|  |     +--> BeiGA (Beispielgruppenangabe)
|  |        +--> BeiA (Beispielangabe): "wir müssen die Person streng von der Sache trennen"
|  |        +--> BeiA (Beispielangabe): "mein Beruf kann von meiner Freizeit nicht streng getr\
                 ennt werden."
|  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
|  |  +--> PAA (Polysemieangabe (arabisch)): "3."
|  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |  |  +--> PAB (Polysemieangabe (Buchstabe)): "a"
|  |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  |  |  +--> GrA (Grammatische Angabe)
|  |  |  |  |  +--> SmA (Satzmusterangabe): "sich trennen"
|  |  |  |  +--> BA (Bedeutungsangabe)
|  |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "auseinandergehen"
|  |  |  +--> BeiGA (Beispielgruppenangabe)
|  |  |  |  +--> BeiA (Beispielangabe): "wir trennten uns am Bahnhof"
|  |  |  |  +--> BeiA (Beispielangabe): "unsere Wege trennen sich hier"
|  |  |  |  |  +--> GZB (Glossat zur Bedeutung): "jeder nimmt einen anderen Weg"
|  |  |  |  +--> BeiA (Beispielangabe): "nach drei Stunden Diskussion trennte man sich"
|  |  |  +--> KSV (Kommentar zur speziellen Verwendung)
|  |  |     +--> SKSV (Subkommentar zur speziellen Verwendung)
|  |  |        +--> VSA (Verwendungsspezifizierende Angabe)
|  |  |        |  +--> FGA (Fachgebietsangabe): "Sport."
|  |  |        +--> BeiGA (Beispielgruppenangabe)
|  |  |           +--> BeiA (Beispielangabe): "die beiden Mannschaften trennten sich 0: 0."
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |     +--> PAB (Polysemieangabe (Buchstabe)): "b"
|  |     +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |     |  +--> GrA (Grammatische Angabe)
|  |     |  |  +--> SmA (Satzmusterangabe): "sich von jmdm. trennen"
|  |     |  +--> BA (Bedeutungsangabe)
|  |     |     +--> BPA (Bedeutungsparaphrasenangabe): "weggehen"
|  |     +--> BeiGA (Beispielgruppenangabe)
|  |        +--> BeiA (Beispielangabe): "vor der Haustür trennte er sich von mir."
|  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
|  |  +--> PAA (Polysemieangabe (arabisch)): "4."
|  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |  |  +--> PAB (Polysemieangabe (Buchstabe)): "a"
|  |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  |  |  +--> GrA (Grammatische Angabe)
|  |  |  |  |  +--> SmA (Satzmusterangabe): "sich trennen"
|  |  |  |  +--> BA (Bedeutungsangabe)
|  |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "eine Partnerschaft, Gemeinschaft auflö\
                    sen"
|  |  |  +--> BeiGA (Beispielgruppenangabe)
|  |  |  |  +--> BeiA (Beispielangabe): "wir haben uns endlich, nach zwei Jahren, freundschaft\
                 lich, im guten getrennt"
|  |  |  |  +--> BeiA (Beispielangabe): "die beiden Teilhaber haben sich getrennt"
|  |  |  +--> KSV (Kommentar zur speziellen Verwendung)
|  |  |     +--> SKSV (Subkommentar zur speziellen Verwendung)
|  |  |        +--> VSA (Verwendungsspezifizierende Angabe)
|  |  |        |  +--> WKA (Wortartenkonversionsangabe): "adj."
|  |  |        |  +--> WKA (Wortartenkonversionsangabe): "Part."
|  |  |        +--> BeiGA (Beispielgruppenangabe)
|  |  |           +--> BeiA (Beispielangabe): "die Eheleute leben getrennt."
|  |  +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|  |     +--> PAB (Polysemieangabe (Buchstabe)): "b"
|  |     +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |     |  +--> GrA (Grammatische Angabe)
|  |     |  |  +--> SmA (Satzmusterangabe): "sich von jmdm. trennen"
```

```
|  |     |  +--> BA (Bedeutungsangabe)
|  |     |     +--> BPA (Bedeutungsparaphrasenangabe): "sich loslösen"
|  |     +--> BeiGA (Beispielgruppenangabe)
|  |     |  +--> BeiA (Beispielangabe): "siehat sich von ihrem Mann getrennt"
|  |     |  +--> BeiA (Beispielangabe): "von meinem Gesangspartner habe ich mich getrennt"
|  |     +--> KSV (Kommentar zur speziellen Verwendung)
|  |        +--> SKSV (Subkommentar zur speziellen Verwendung)
|  |           +--> VSA (Verwendungsspezifizierende Angabe)
|  |           |  +--> MA (Metaphernangabe): "übertr."
|  |           +--> PragA (Pragmatische Angabe): "verhüll."
|  |           +--> BeiGA (Beispielgruppenangabe)
|  |              +--> BeiA (Beispielangabe): "die Firma hat sich von diesem Mitarbeiter getre\
                       nnt."
|  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
|  |  +--> PAA (Polysemieangabe (arabisch)): "5."
|  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|  |  |  +--> GrA (Grammatische Angabe)
|  |  |  |  +--> SmA (Satzmusterangabe): "sich von etwas trennen"
|  |  |  +--> BA (Bedeutungsangabe)
|  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "etwas hergeben"
|  |  +--> BeiGA (Beispielgruppenangabe)
|  |  |  +--> BeiA (Beispielangabe): "sich von Erinnerungsstücken nur ungern trennen, nicht tr\
              ennen können"
|  |  |  +--> BeiA (Beispielangabe): "sich von jeglichem Besitz trennen"
|  |  +--> KSV (Kommentar zur speziellen Verwendung)
|  |     +--> SKSV (Subkommentar zur speziellen Verwendung)
|  |        +--> VSA (Verwendungsspezifizierende Angabe)
|  |        |  +--> MA (Metaphernangabe): "übertr."
|  |        +--> BeiGA (Beispielgruppenangabe)
|  |           +--> BeiA (Beispielangabe): "sich von einem Gedanken, einem Wunsch, einer Vorst\
                    ellung trennen müssen"
|  |           +--> BeiA (Beispielangabe): "sich von einem Anblick nicht trennen können."
|  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
|     +--> PAA (Polysemieangabe (arabisch)): "6."
|     +--> PragSemA (Pragmatisch-Semantische Angabe)
|     +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|     |  +--> PAB (Polysemieangabe (Buchstabe)): "a"
|     |  +--> PragSemA (Pragmatisch-Semantische Angabe)
|     |  |  +--> GrA (Grammatische Angabe)
|     |  |  |  +--> SmA (Satzmusterangabe): "etwas trennt etwas"
|     |  |  +--> BA (Bedeutungsangabe)
|     |  |     +--> BPA (Bedeutungsparaphrasenangabe): "etwas bildet eine Grenze, ein Hinderni\
                    s zwischen etwas"
|     |  +--> BeiGA (Beispielgruppenangabe)
|     |  |  +--> BeiA (Beispielangabe): "ein Stacheldraht trennt die Grundstücke"
|     |  |  +--> BeiA (Beispielangabe): "ein Zaun trennt die Gärten"
|     |  +--> KSV (Kommentar zur speziellen Verwendung)
|     |     +--> SKSV (Subkommentar zur speziellen Verwendung)
|     |        +--> VSA (Verwendungsspezifizierende Angabe)
|     |        |  +--> MA (Metaphernangabe): "übertr."
|     |        +--> BeiGA (Beispielgruppenangabe)
|     |           +--> BeiA (Beispielangabe): "uns trennen Welten"
|     |           |  +--> GZB (Glossat zur Bedeutung): "wir sind äußerst verschieden"
|     |           +--> BeiA (Beispielangabe): "die verschiedene Herkunft trennte sie."
|     +--> SSK2 (Semantischer Subkommentar 2. Stufe)
|        +--> PAB (Polysemieangabe (Buchstabe)): "b"
|        +--> PragSemA (Pragmatisch-Semantische Angabe)
|        |  +--> GrA (Grammatische Angabe)
|        |  |  +--> SmA (Satzmusterangabe): "etwas trennt jmdn., etwas von jmdm., von etwas"
|        |  +--> BA (Bedeutungsangabe)
|        |     +--> BPA (Bedeutungsparaphrasenangabe): "etwas grenzt jmdn., etwas gegen jmdn.,\
                     etwas ab"
|        +--> BeiGA (Beispielgruppenangabe)
|           +--> BeiA (Beispielangabe): "der Kanal trennt England vom Kontinent"
|           +--> BeiA (Beispielangabe): "nur ein Graben trennt die Zoobesucher von den Elefant\
                 en"
|           +--> BeiA (Beispielangabe): "eine Glaswand trennt ihn von seinem Verteidiger"
|           +--> UGrA (Unspezifizierte grammatische Angabe): "auch ohne Präp.- Obj."
```

```
|          +--> BeiA (Beispielangabe): "ein Gebirgszug trennt das Land in zwei Regionen."
+--> WA_Err (Fehler in Wörterbuchartikel): "übertr.: nur noch wenige Tage trennen uns von den \
     Wahlen. XFTbo 7. XFTst (Rundf.) <etwas trennt etwas; mit Artangabe> XFTit etwas besitzt e\
     ine bestimmte Trennschärfe: XFTst das Radio trennt die Sender gut, scharf, nicht richtig,\
      genügend. XFTbo 8. XFTst <jmdn., etwas trennen> XFTit (eine telefonische Verbindung unte\
     rbrechen: XFTst die Verbindung wurde getrennt; man hat uns getrennt."

Entry #7: Error caused by directive ('@' operator)

----------------------------------------------------------------------
Inputfile..........: wbs/9-entries.dat
First entry........: 7
Last entry.........: 7
Entries parsed.....: 1
Errors.............: 1
Time of day........: Fri Jul  1 11:10:39 1994
Time of start......: Fri Jul  1 11:10:34 1994
Elapsed time.......: 00h 00m 05s
Average parse time.: 5 sec.

LEXPARSE V1.208 ß-Code   [Jul  1 1994]   <DEBUG-SRC>
       Program terminated successfully

LEXPARSE: (W-105) Warning occurred in module 'LEXPARSE':
       Parsing terminated due to 'LastEntry' specification
```

E.3 'überhaupt', DUDEN-Stilwörterbuch, Page 716

The following dictionary entry was parsed by LEXPARSE successfully. The parse tree for the entry überhaupt is complete. Abbreviated lemma signs were expanded automatically by the program.

However, the entry is *not* well–formed since the enumeration in the item *Polysemieangabe* (PAA) is invalid. The parser did find an invalid symbol 5 instead of the expected symbol 4 in the scope of the forth instance of item *Semantischer Subkommentar* (SSK).

Nevertheless, the parser generated a complete parse tree and issued an error message (warning #403) indicating the invalid enumeration.[45]

```
[D:\FOR\SFS\LEXPARSE]lexparse lexparse.ini /nowarn wbs\counter_error.dat

LEXPARSE V1.208 ß-Code   [Jul  1 1994]   <DEBUG-SRC>   A Dictionary Parser

(C) 1992-1994 Seminar fuer Sprachwissenschaft at the University of Tuebingen
(C) 1992-1994 Ralf Hauser
    Created by 'Affie' (k.f. c|o)

LEXPARSE: Reading configuration: lexparse.ini
LEXPARSE: Grammar checked and OK.

Inputfile..........: wbs/counter_error.dat
Parsing............: Enabled
Grammar nodes......: 514
First entry........: 1
Last entry.........: no spec.
Time of day........: Fri Jul  1 11:22:41 1994

----------------------------------------------------------------------
```

[45]The switch `[Parser] RecoverCounterError = Yes` does achieve such a behaviour.

```
 N O T E: Entry #1 could not be accepted!
----- ParseTree 1 -----
WA (Wörterbuchartikel)
 +--> FK (Formkommentar)
 |  +--> LZGA (Lemmazeichengestaltangabe): "überhaupt"
 +--> SK (Semantischer Kommentar)
    +--> SSK0 (Semantischer Subkommentar 0. Stufe)
    |  +--> PAR (Polysemieangabe (roemisch)): "I."
    |  +--> PragSemA (Pragmatisch-Semantische Angabe)
    |  |  +--> GrA (Grammatische Angabe)
    |  |     +--> WAA (Wortartangabe): "Adverb"
    |  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
    |  |  +--> PAA (Polysemieangabe (arabisch)): "1."
    |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
    |  |  |  +--> BA (Bedeutungsangabe)
    |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "insgesamt, aufs Ganze gesehen"
    |  |  +--> BeiGA (Beispielgruppenangabe)
    |  |     +--> BeiA (Beispielangabe): "ich habe ihn gestern nicht angetroffen, er ist überha\
                    upt selten zu Hause"
    |  |     +--> BeiA (Beispielangabe): "mir gefällt es in Madrid, überhaupt in Spanien."
    |  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
    |  |  +--> PAA (Polysemieangabe (arabisch)): "2."
    |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
    |  |  |  +--> GrA (Grammatische Angabe)
    |  |  |  |  +--> UGrA (Unspezifizierte grammatische Angabe): "verstärkend bei Verneinungen"
    |  |  |  +--> BA (Bedeutungsangabe)
    |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "ganz und gar"
    |  |  +--> BeiGA (Beispielgruppenangabe)
    |  |     +--> BeiA (Beispielangabe): "das ist überhaupt nicht möglich, nicht wahr"
    |  |     +--> BeiA (Beispielangabe): "davon kann überhaupt keine Rede sein"
    |  |     +--> BeiA (Beispielangabe): "er hat heute überhaupt noch nichts gegessen"
    |  |     +--> BeiA (Beispielangabe): "das geht ihn überhaupt nichts an."
    |  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
    |  |  +--> PAA (Polysemieangabe (arabisch)): "3."
    |  |  +--> PragSemA (Pragmatisch-Semantische Angabe)
    |  |  |  +--> BA (Bedeutungsangabe)
    |  |  |     +--> BPA (Bedeutungsparaphrasenangabe): "abgesehen davon, überdies"
    |  |  +--> BeiGA (Beispielgruppenangabe)
    |  |     +--> BeiA (Beispielangabe): "du kannst einmal nachfragen, und überhaupt solltest d\
                    u dich/ und überhaupt, du solltest dich mehr darum kümmern."
    |  +--> SSK1 (Semantischer Subkommentar 1. Stufe)
    |     +--> PAA (Polysemieangabe (arabisch)): "5."
    |     +--> PragSemA (Pragmatisch-Semantische Angabe)
    |     |  +--> BA (Bedeutungsangabe)
    |     |     +--> BPA (Bedeutungsparaphrasenangabe): "gerade, besonders"
    |     +--> BeiGA (Beispielgruppenangabe)
    |        +--> BeiA (Beispielangabe): "wir gehen gerne im Wald spazieren, überhaupt im Herbs\
                    t"
    |        +--> BeiA (Beispielangabe): "man wird, überhaupt im Alter, nachlässiger."
    +--> SSK0 (Semantischer Subkommentar 0. Stufe)
       +--> PAR (Polysemieangabe (roemisch)): "II."
       +--> PragSemA (Pragmatisch-Semantische Angabe)
       |  +--> GrA (Grammatische Angabe)
       |  |  +--> UGrA (Unspezifizierte grammatische Angabe): "Partikel"
       |  +--> BA (Bedeutungsangabe)
       |     +--> BPA (Bedeutungsparaphrasenangabe): "eigentlich"
       +--> BeiGA (Beispielgruppenangabe)
          +--> BeiA (Beispielangabe): "was willst du überhaupt hier?"
          +--> BeiA (Beispielangabe): "wie ist das überhaupt passiert?"
          +--> BeiA (Beispielangabe): "du könntest überhaupt etwas freundlicher sein."
             +--> GZB (Glossat zur Bedeutung): "ruhig"

 Entry #1: Error caused by an invalid counter value


 LEXPARSE: (W-403) Warning occurred in module 'Parser':
        Probably incorrect value for counter detected:
        "5":
```

```
       '4'
----------------------------------------------------------------------
 Inputfile..........: wbs/counter_error.dat
 First entry........: 1
 Last entry.........: 1
 Entries parsed.....: 1
 Errors.............: 1
 Time of day........: Fri Jul  1 11:22:41 1994
 Time of start......: Fri Jul  1 11:22:41 1994
 Elapsed time.......: 00h 00m 01s
 Average parse time.: < 1 sec.

 LEXPARSE V1.208 ß-Code   [Jul  1 1994]   <DEBUG-SRC>
        Program terminated successfully
```