# Declarative Parsing and Annotation of Electronic Dictionaries

Christian Schneiker[1], Dietmar Seipel[1], Werner Wegstein[2], and Klaus Prätor[3]

[1] Department of Computer Science
{schneiker|seipel}@informatik.uni-wuerzburg.de
[2] Competence Center for Computational Philology
werner.wegstein@uni-wuerzburg.de
University of Würzburg, Am Hubland, D – 97074 Würzburg, Germany

[3] Berlin–Brandenburg Academy of Sciences and the Humanities
Jägerstr. 22–23, D – 10117 Berlin, Germany
praetor@bbaw.de

**Abstract.** We present a declarative annotation toolkit based on XML and PROLOG technologies, and we apply it for annotating the Campe Dictionary to obtain an electronic version in XML (TEI).
For parsing flat structures, we use a very compact grammar formalism called extended definite clause grammars (EDCG's), which is an extended version of the DCG's that are well–known from the logic programming language PROLOG.
For accessing and transforming XML structures, we use the XML query and transformation language FNQUERY.
It turned out, that the declarative approach in PROLOG is much more readable, reliable, flexible, and faster than an alternative implementation which we had made in JAVA and XSLT for the TEXTGRID community project.

## 1 Introduction

Dictionaries traditionally offer information on words and their meaning in highly structured and condensed form, making use of a wide variety of abbreviations and an elaborate typography. Retro–digitizing printed dictionaries, especially German dictionaries from the 18th and early 19th century, therefore requires sophisticated new tools and encoding techniques in order to convert typographical detail into a fine–grain *markup* of dictionary structures on one hand and to allow at the same time for variation in orthography, morphology and usage on the other hand, since in the decades around 1800 the German language was still on its way to standardization.

This is one of the reasons why TEXTGRID [17], the first grid project in German e*Humanities*, funded by the Federal Ministry of Education and Research, chose the Campe Dictionary [1]: 6 volumes with altogether about 6.000 pages and about 140.000 entries, published between 1807 and 1813 as one testbed for their TEXTGRID Lab, a Virtual Research Library. It entails a grid–enabled workbench, that will process, analyse, annotate, edit and publish text data for academic research, and TEXTGRIDRep,

a grid repository for long–term storage. This paper reflects some of the research on annotation in this context.

Using PROLOG technology for *parsing* and *annotating* is common in natural language processing. PROLOG has been used within the Jean Paul project at the Berlin–Brandenburg Academy of Sciences [15], where XML transformations based on FNQUERY turned out to be easier to write than XSLT transformations. The task is to find the proper level of abstraction and write suitable macros for frequently occurring patterns in the code; PROLOG even allows to design dedicated special–purpose languages [11]. E.g., definite clause grammars have been developed as an abstraction for parsing; they have been used for parsing controlled natural languages [4, 5, 13]. Since PROLOG code is declarative and very compact, the variations of natural language can be handled nicely.

Figure 1 shows the typographical layout of the Campe Dictionary. The basic text has been double–keyed in China. The encoding is based on the TEI P5 Guidelines [16], using a Relax NG Schema. The encoding structure uses elements to markup the dictionary entry, the form block with inflectional and morphological information, the sense block handling semantic description and references, quotations, related entries, usage as well as notes. In the future, this encoding will help us to structure the digital world according to semantic criteria and thus provide an essential basis for constructing reliable ontologies.

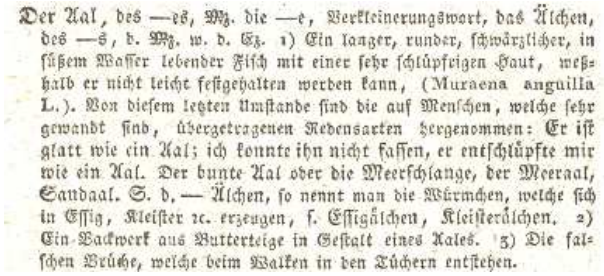
Der Aal, des —es, Mz. die —e, Verkleinerungswort, das Älchen, des —s, d. Mz. w. d. Ez. 1) Ein langer, runder, ſchwärzlicher, in ſüßem Waſſer lebender Fiſch mit einer ſehr ſchlüpfrigen Haut, weßhalb er nicht leicht feſtgehalten werden kann, (Muraena anguilla L.). Von dieſem letzten Umſtande ſind die auf Menſchen, welche ſehr gewandt ſind, übergetragenen Redensarten hergenommen: Er iſt glatt wie ein Aal; ich konnte ihn nicht faſſen, er entſchlüpfte mir wie ein Aal. Der bunte Aal oder die Meerſchlange, der Meeraal, Sandaal. S. d. — Älchen, ſo nennt man die Würmchen, welche ſich in Eſſig, Kleiſter ꝛc. erzeugen, ſ. Eſſigälchen, Kleiſterälchen. 2) Ein Backwerk aus Butterteige in Geſtalt eines Aales. 3) Die falſchen Brüche, welche beim Walken in den Tüchern entſtehen.

**Figure 1.** Excerpt from the Campe Dictionary

The rest of this paper is organized as follows: In Section 2 we will show how to extract entries of a dictionary using our XML query and transformation language FNQUERY. The process for parsing and annotating elements for creating a well–formed and valid TEI structure is illustrated for the Campe Dictionary in Section 3: we can access and create elements with PROLOG using DCG's and EDCG's in an easy and efficient way. Section 4 describes the use of transformation rules for the annotation and the further processing of XML elements. Section 5 gives an overview of our declarative annotation toolkit and compares the declarative techniques with a JAVA and XSLT approach, which we had implemented earlier.

## 2 Basic XML Handling in PROLOG and FNQUERY

Currently, the Campe Dictionary, which was written in the early years of the 19th century, is converted into a machine readable structure. Within this process, the only annotations available so far, were the declaration of the different font sizes Joachim Heinrich Campe uses for displaying the structure of his act, the numbering of the line and page breaks in the dictionary, and paragraphs; thus, we found a very limited XML structure in the source file which we used for the first basic transformaions.

In this paper, we present how to annotate documents with PROLOG and the XML query and transformation language FNQUERY [14], which is implemented in SWI–PROLOG. To exemplify these annotations, we use the Campe Dictionary as a low–structured base for obtaining a well–formed XML document according to TEI.

**The PROLOG Data Structure for XML.** The field notation developed for FNQUERY represents an XML element `<T` $a_1 =$ "$v_1$"$\ldots a_n =$ "$v_n$"`>...</T>` as a PROLOG term `T:As:C`, called FN triple, with the tag "`T`" and an association list `As` $= [a_1 : v_1, \ldots, a_n : v_n]$ of attributes $a_i$ and their corresponding values $v_i$ (with $1 \leq i \leq n$), which are PROLOG terms. The content `C` can be either text or nested sub–elements represented as FN triples. If `As` is empty, then the FN triple can be abbreviated as a pair `T:C`.

In most available dictionaries, each entry is encapsulated in its own paragraph, and thus, it could be easily detected. In the following example, an entry is annotated with `paragraph` and is followed by an element `W_2`, which shows the lemma of the entry in a larger font; recognizing both elements is necessary, because there could exist other `paragraph` elements, which do not represent entries.

```
<paragraph>
  <W_2>Der Aal</W_2>, <W_1>des -- es, Mz. die -- e</W_1>, ...
</paragraph>
```

An XML document can be loaded into an FN triple using the predicate `dread`. For the `paragraph` element above we get FN triples with empty attribute lists:

```
paragraph:[ 'W_2':['Der Aal'], ', ',
   'W_1':['des -- es, Mz. die -- e'], ', ...' ]
```

**Extraction of Entries Using FNQUERY.** The query language FNQUERY allows for accessing a component of an XML document by its attribute or tag name. Furthermore, complex *path* or *tree expressions* can be formulated in a way quite similar to XPATH.

The XML element from above could now be parsed with the following predicate `campe_find_entry/2`. The path expression `Campe/descendant::paragraph` selects a descendant element of `Campe` with the tag `paragraph`. For avoiding the recognition of a new paragraph without a following `W_2` tag, we use another path expression `Entry/nth_child::1/tag::'*'` for computing the tag of the first child of the considered entry.

```
campe_find_entry(Campe, Entry) :-
   Entry := Campe/descendant::paragraph,
   'W_2' := Entry/nth_child::1/tag::'*'.
```

Finally, with PROLOG's backtracking mechanism it is possible to find all entries in the source file.

## 3 Annotation with Extended Grammar Rules

In the past, PROLOG has been frequently used for implementing natural language applications, in which parsing text is one of the main concerns. Most PROLOG systems include a preprocessor for defining grammar rules for parsing text. These *definite clause grammars* hide arguments which are not relevant for the semantics of the parsing problems; thus, they are more readable and reliable than standard PROLOG rules.

In this section, we want to discuss this benefit for parsing electronic dictionaries. We give an example for parsing a lemma of an entry to generate TEI elements. In a further step, we introduce an extended version of DCG's (EDCG's), which give the user the ability to create compact grammar rules for generating generic FN terms for the parsed tokens. A comparison to DCG's and standard PROLOG rules will be elaborated in addition to the possibility of integrating EDCG's in the other formalisms.

### 3.1 Parsing with Definite Clause Grammars

Firstly, we will use DCG's for parsing the headwords of a single entry and for detecting punctuation in natural text. Using grammar rules is more reliable than using standard PROLOG rules, since the code becomes much more compact and readable. Moreover, we introduce the `sequence` predicate for parsing a list of XML elements.

**Headwords.** An entry of a dictionary normally consists of a lemma which is highlighted with a larger font; in our case, it is annotated with `W_2`–tags. Often, such a lemma only consists of one word – in the case of verbs – or a noun and its determiner:

The DCG predicate `campe_headword` given below parses a headword XML element `<W_2>Der Aal</W_2>` and annotates it to derive the following `form`–element:

```
<form>
  <form type="lemma">
    <form type="determiner"> <orth>Der</orth> </form>
    <form type="headword"> <orth>Aal</orth> </form>
  </form>
</form>
```

There also exist some cases with more than one headword or additional XML tags depending on the current stage of the process, such as line breaks or abbreviations, e.g., the following collective reference to related entries:

```
<W_2>Der Blitzstoffmesser, der Blitzstoffsammler, <lb n="0569.49" />
der Blitzstoffsauger</W_2>
```

The additional elements have to be passed through and should not be annotated; the different headwords have to be annotated, and a `form`–element for each lemma has to be created.

The following DCG rule for `campe_headword` can handle the described variations; it parses the different types of `W_2` elements to create FN triples `X`:

```
campe_headword(X) -->
   ( [X], { X = T:As:Es }
   ; [X], { atomic(X), campe_is_unicode_char(X) }
   ; [A, B], { atomic(A), atomic(B),
       campe_is_determiner(A),
       X = form:[type:lemma]:[
              form:[type:determiner]:[orth:[A]],
              form:[type:headword]:[orth:[B]] ] }
   ; [A], { atomic(A),
       X = form:[type:lemma]:[
              form:[type:headword]:[orth:[A] ] ] } ).
```

With standard DCG technology, this predicate has to be called recursively for parsing a (possibly empty) sequence of headwords. This is done by the recursive predicate `campe_headwords`, which terminates when no more headwords are found:

```
campe_headwords([X|Xs]) --> campe_headword(X), campe_headwords(Xs).
campe_headwords([]) --> { true }.
```

To simplify this, we have developed the meta-predicate `sequence`; like in regular expressions, the first argument `'*'` indicates that we look for an arbitrary number of headwords (other possible values are, e.g., `'+'` or `'?'`):

```
campe_headwords(Xs) --> sequence('*', campe_headword, Xs).
```

We can apply `campe_headwords` to produce a sequence of headwords, which are afterwards enclosed in a `form` tag and written to the screen:

```
?- campe_headwords(Xs, ['Der', 'Aal'], []), dwrite(xml, form:Xs).
```

**Punctuation.** For annotating punctuation in a lemma, which can appear between single headwords, the DCG predicate `campe_punctuation` is used for checking each token if it is a punctuation mark, and – if so – annotating it with a `c`–tag.

```
campe_punctuations(Xs) --> sequence('*', campe_punctuation, Xs).

campe_punctuation(X) -->
   ( [A], { is_punctuation(A), X = c:[A] } ; [X] ).
```

The meta-predicate `sequence` used in the DCG predicate `campe_punctuations` parses a list of elements.

### 3.2 Parsing with Extended Definite Clause Grammars

In the following, we show how nouns can be parsed using EDCG's. For complex applications, standard DCG's can have a complex structure, and understanding and debugging them can be tedious. Thus, we have developed a new, more compact syntax for writing DCG rules in PROLOG, which we call Extended DCG's (EDCG). For the representation of XML elements created by EDCG's we use a generic field notation.

Assume that a paragraph of a dictionary has to be parsed and annotated for labeling the inflected forms of a noun. In the Campe Dictionary, lemma variations (such as plural or genitive forms) are found in nearly all of the regular substantives; thus, an easy to read structure has to be developed to give the programmer the potential of writing complex parsers in a user friendly way.

The following line shows such an extract of the Campe Dictionary.

```
des -- es, Mz. die -- e
```

These tokens have to be annotated in TEI, and different `form` tags with sub–elements defining the corresponding grammatical structure have to be created. The following XML code, which should be produced, shows the complexity of such annotations, which indicates that the corresponding PROLOG code will also be complex:

```
<form type="inflected">
   <gramGrp>
      <gram type="number"> <abbr>Mz.</abbr> </gram>
      <case value="nominative"/>
      <number value="plural"/> </gramGrp>
   <form type="determiner"> <orth>die</orth> </form>
   <form type="headword">
      <orth> <oVar> <oRef>-- e</oRef> </oVar> </orth> </form>
</form>
```

**Extended Definite Clause Grammars.** For solving a parsing problem using regular PROLOG rules, the input tokens as well as the field notation for the created XML have to be processed.

Thus, we have developed a new notation for parsing language where the output is regular XML. Instead of using the functor `-->` of DCG's, we are using the new functor `==>` for writing EDCG's. The output arguments can be hidden, since the output is constructed in a *generic* way: the output of an EDCG rule with the head `T` is a list `[T:Xs]` containing exactly one FN triple, where `Xs` is the list of FN triples produced by the body of the rule.

The following EDCG rules parse inflected forms – like indicated above – into FN triples:

```
form ==> grammar_determiner, form_headword.
grammar_determiner ==> ( gram, !, determiner ; determiner ).
gram ==> ['Mz.'].
determiner ==> [X], { campe_is_determiner(X) }.
form_headword ==> orth.
orth ==> ['--', _].
```

Below, we call the predicate `form` for parsing a list of tokens (the second argument) into a list `Xs` (the first argument) of `form` elements; the last argument contains the tokens that could not be parsed – i.e., it should be empty:

```
?- form(Xs, ['Mz.', die, '--e'], []), dwrite(xml, form:Xs).
```

The output of the predicate `form` is enclosed in a further `form` tag and written to the screen:

```
<form>
  <grammar_determiner>
    <gram>Mz.</gram> <determiner>die</determiner>
  </grammar_determiner>
  <form_headword> <orth>-- e</orth> </form_headword>
</form>
```

From this, the exact, desired XML structure can be derived using some simple transformations, which we will describe in Section 4. The code is much better understandable than for DCG's, because this notation suppresses irrelevant arguments.

If we define `grammar_determiner` with the following DCG's and mix them with the EDCG's for the other predicates, then we can get even closer to the desired XML structure in one step – at the expense of a less compact code:

```
grammar_determiner([G, F]) -->
   gram([gram:[C]]), !, determiner([determiner:[D]]),
   { G = gramGrp:[ gram:[type:number]:[abbr:[C]],
        case:[value:nominative]:[],
        number:[value:plural]:[] ],
     F = form:[type:determiner]:[orth:[D]] }.
grammar_determiner([G, F]) -->
   determiner([determiner:[D]]),
   { G = gramGrp:[
        case:[value:genitive]:[],
        number:[value:singular]:[] ],
     F = form:[type:determiner]:[orth:[D]] }.
```

Instead of the generic element `grammar_determiner` produced by the EDCG rule, the DCG rules can produce the two elements (`G` and `F`) of the desired XML structure. Now, we can derive the complete desired XML with very simple transformations.

Finally, the different cases like genitive or dative and the plural forms of a dictionary entry could be parsed using similar DCG or EDCG rules.

**Comparison with DCG's and Standard PROLOG.** In contrast, for our example the corresponding standard DCG rules of PROLOG (we show only half of them) are more complex than the EDCG rules:

```
form([form:Es]) -->
   grammar_determiner(Xs), form_headword(Ys),
   { append(Xs, Ys, Es) }.
gram([gram:['Mz.']]) --> ['Mz.'].
determiner([determiner:[X]]) --> [X], { campe_is_determiner(X) }.
```

In many applications – like the annotation of electronic dictionaries or other programs producing XML – these DCG rules are quite complex and simplifying them is necessary. Finally, the implementation in pure, standard PROLOG would look even more complicated (again, we show only half of the rules):

```
form([form:Es], As, Bs) :-
   grammar_determiner(Xs, As, Cs), form_headword(Ys, Cs, Bs),
   append(Xs, Ys, Es).
gram([gram:['Mz.']], As, Bs) :-
   As = ['Mz.'|Bs].
determiner([determiner:[X]], As, Bs) :-
   campe_is_determiner(X), As = [X|Bs].
```

Besides the output in the first argument of the DCG predicates here, which is constructed explicitely rather than generic, there are two more arguments for passing the list of input tokens. DCG's only use the first argument, and EDCG's hide all three arguments.

**Sequences of Form Elements.** If we add the following DCG rule for `form` at the beginning of the DCG program and assume that commas have already been annotated as FN triples `c:[',']`, then we can annotate sequences of inflected `form` elements:

```
form([X]) --> [X], { X = T:As:Es, ! }.
```

With the predicate `sequence` it is possible to parse a sequence `Ts` of tokens to inflected form elements, even when more than one genitive or plural form occurs. Since the output `Fs` is a list of lists of elements, we have to flatten it to an ordinary list `Xs` before we can write it to the screen:

```
?- Ts = [des, '--', es, c:[','], 'Mz.', die, '--', e],
   sequence('*', form, Fs, Ts, []),
   flatten(Fs, Xs), dwrite(xmls, Xs).
```

These PROLOG rules are efficient and easly readable. The derived FN triples can be ouput in XML using the predicate `dwrite/2`.

## 4 Annotation with Transformation Rules

In this section, we transform XML elements from the Campe Dictionary with FNQUERY in a way quite similar to XSLT, but with a more powerful backengine in PROLOG.

A rule with the head `--->(Predicate, T1, T2)` transforms an FN triple `T1` to another FN triple `T2`. Arbitrary PROLOG calls could be integrated within the rules; thus, FNQUERY is a Turing complete transformation language. FN transformation rules are called by the predicate `fn_item_transform`. The transformation is recursive; it starts in the leaves of the XML tree and ends in the root element.

For example, the following rules transform all `A` elements in an FN triple to an `hi` element with an attribute `rend="roman"` and all `W_1` elements to an `hi` element with an attribute `rend="large"` for labeling the font size. Other elements are left unchanged, because of rule 3:

```
--->(antiqua, 'A':_:Es, hi:[rend:roman]:Es).
--->(large_font, 'W_1':_:Es, hi:[rend:large]:Es).
--->(_, X, X).
```

Another example is the transformation of the XML elements created by the EDCG's of Section 3.

```
--->(inflected, form:_:[T1, T2], form:[type:inflected]:[G, F1, F2]) :-
   D := T1/determiner/content::'*', O := T2/orth,
   ( C := T1/gram/content::'*', C = ['Mz.'] ->
     G = ...
   ; G = ... ),
   F1 = form:[type:determiner]:[orth:D],
   F2 = form:[type:headword]:[oVar:[oRev:[O]]].
```

This rule transforms an FN triple analogously to the DCG rule for `grammar_determiner` defined in Section 3; an attribute `type` is added to each of the `form` elements. The element `grammar_determiner` is separated into two different elements (namely `gramGrp` including two additional elements `case` and `number`) and an optional `gram` element; all of them depend on the content of the `case` element. The `orth` sub–element of the `form` element with attribute `type="headword"` is enclosed in two more elements for obtaining the desired structure.

## 5 The Declarative Annotation Toolkit

The annotation techniques presented in the previous sections are part of an integrated declarative annotation toolkit. In this section, we sketch some further annotations which we have implemented in the field of electronic dictionaries.

### 5.1 Annotations of Electronic Dictionaries

With FNQUERY, DCG's and EDCG's we have developed several applications for parsing natural text in electronic dictionaries such as Campe and Adelung. These techniques give us the possibility to identify and annotate lemmas, their inflected forms, as well as punctuations and hyphenations in parsed entries.

Furthermore, it is possible to match slightly modified variants of a lemma in the text and to parse certain German relative clauses. Moreover, we have developed an application for annotating sub–grouped senses in an entry labeled with list markers such as `I`, `1)`, `2)`, `a.`, `b)`, where PROLOG's backtracking mechanism is very useful for obtaining the proper structure. Figure 2 shows the rendering of an entry, which we had annotated with PROLOG before.

### 5.2 Comparison to JAVA and XSLT

In an earlier step of the TEXTGRID–project, we had designed and implemented a tool for parsing and annotating the Campe Dictionary in JAVA and XSLT. According to the guidelines of the TEXTGRIDLAB, this implementation was necessary for the community project.

For Volume 1 of the Campe Dictionary, which consists of 26.940 entries, the JAVA approach needs approximately 44 minutes for parsing and annotating all entries on a

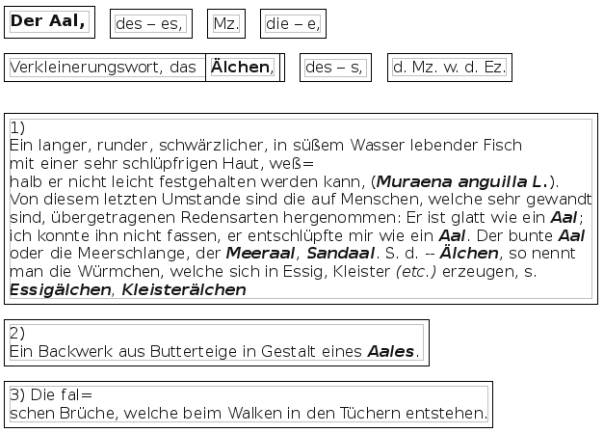


**Figure 2.** Rendering of an Annotated Entry

dual core CPU system using two threads. In our new approach in PROLOG, we can reduce this runtime to about 4 minutes while using only one core of the system. Since we are now using declarative technology, the code length could be reduced to only 5% of the JAVA implementation.

## 6 Conclusions

In the BMBF research project on *variations in language*, we want to build a meta–lemma list by analyzing a huge collection of dictionaries from different epochs of the German language. Both here and in the TEXTGRID community project, we need a fast, reliable, easy to read and modular toolkit for parsing, annotating and querying data sets. With the development of FNQUERY and EDCG's, we have the possibility to fulfil these requirements; our declarative annotation toolkit is even faster than modern applications written in JAVA and XSLT.

The usability of the introduced technologies is not limited to the annotation and parsing of natural language; in another project we are using EDCG's and transformation rules for analyzing log messages – or even data buses – in order to find root causes in network systems.

The aspect of integrating *text mining* to extend our toolkit will be a subject of future research.

## References

1. CAMPE, Joachim Heinrich: *Wörterbuch der deutschen Sprache*. 5 Volumes, Braunschweig, 1807–1811.
2. COVINGTON, M.A.: *GULP 3.1: An Extension of Prolog for Unification–Based Grammar*. *Research Report AI-1994-06, Artificial Intelligence Center, University of Georgia, 1994*

3. DEREKO: *The German Reference Corpus Project*. *http://www.sfs.nphil.uni-tuebingen.de/dereko/, 2009*
4. FUCHS, N.E.; FROMHERZ, M.P.J.: *Transformational Development of Logic Programs from Executable Specifications – Schema Based Visual and Textual Composition of Logic Programs*. *C. Beckstein, U. Geske (eds.), Entwicklung, Test und Wartung deklarativer KI–Programme, GMD Studien Nr. 238, Gesellschaft für Informatik und Datenverarbeitung, 1994*
5. FUCHS, N.E.; SCHWITTER, R.: *Specifying Logic Programs in Controlled Natural Language*. *Proc. Workshop on Computational Logic for Natural Language Processing (CLNP) 1995*
6. GAZDAR, G.; MELLISH, C. *Natural Language Processing in Prolog*. An Introduction to Computational Linguistics. *Addison–Wesley, 1989*
7. HAUSMANN, F.J.; REICHMANN, O.; WIEGAND, H.E.; ZGUSTA, L.; eds.: *Wörterbücher / Dictionaries / Dictionnaires – Ein internationales Handbuch zur Lexikographie / An International Encyclopedia of Lexicography / Encyclopédie internationale de lexicographie*. Berlin/New York, 1989 (I), 1990 (II)
8. HIRAKAWA, H.; ONO, K.; YOSHIMURA, Y.: *Automatic Refinement of a POS Tagger Using a Reliable Parser and Plain Text Corpora*. *Proc. 18th International Conference on Computational Linguistics (COLING) 2000*
9. LANDAU, S.: *Dictionaries. The Art and Craft of Lexicography*. 2nd Edition, Cambridge, 2001
10. LLOYD, J.: *Practical Advantages of Declarative Programming*. *CSLI Lecture Notes, Number 10, 1987*
11. O'KEEFE, R.A.: *The Craft of Prolog*. *MIT Press, 1990*
12. PEREIRA, F.C.N.; SHIEBER, S.M: *Prolog and Natural–Language Analysis*. *CSLI Lecture Notes, Number 10, 1987*
13. SCHWITTER, R.: *Working for Two: a Bidirectional Grammer for a Controlled Natural Language*. *Proc. 21st Australasian Joint Conference on Artificial Intelligence (AI) 2008*, pp. 168-179
14. SEIPEL, D.: *Processing XML Documents in Prolog*. *Proc. 17th Workshop on Logic Programmierung (WLP) 2002*
15. SEIPEL, D.; PRÄTOR, K.: XML *Transformations Based on Logic Programming*. *Proc. 18th Workshop on Logic Programming (WLP) 2005*, pp. 5-16
16. TEI CONSORTIUM, eds.: TEI *P5: Guidelines for Electronic Text Encoding and Interchange*. *http://www.tei-c.org/Guidelines/P5/*
17. TEXTGRID: *Modular platform for collaborative textual editing – a community grid for the humanities*. *http://www.textgrid.de, 2009*