Spejd: A Shallow Processing and Morphological Disambiguation Tool

Adam Przepiórkowski and Aleksander Buczyński

Polish Academy of Sciences, Institute of Computer Science
ul. Ordona 21, 01-237 Warsaw, Poland
{adamp,olekb}@ipipan.waw.pl
http://www.ipipan.waw.pl

Abstract. This article presents a formalism and a beta version of a new tool for simultaneous morphosyntactic disambiguation and shallow parsing. Unlike in the case of other shallow parsing formalisms, the rules of the grammar allow for explicit morphosyntactic disambiguation statements, independently of structure-building statements, which facilitates the task of the shallow parsing of morphosyntactically ambiguous or erroneously disambiguated input.

Key words: morphosyntactic disambiguation, partial parsing, shallow parsing, constituency parsing, syntactic words, syntactic groups, spejd, poliqarp

1 Introduction

Two observations motivate the work described here. First, morphosyntactic disambiguation and shallow parsing inform each other and should be performed in parallel, rather than in sequence. Second, morphosyntactic disambiguation and shallow parsing rules often implicitly encode the same linguistic intuitions, so a formalism is needed which would allow to encode disambiguation and structure-building instructions in a single rule.

The aim of this paper is to present a new formalism and tool, Spejd,[1] abbreviated to "♠" (the Unicode character 0x2660). The formalism is essentially a cascade of regular grammars, where (currently) each regular grammar is expressed by a — perhaps very complex — single rule. The rules specify, both morphosyntactic disambiguation/correction operations and structure-building operations, but, unlike in pure unification-based formalisms, these two types of operations

[1] *Spejd* stands for the Polish *Składniowy Parser (Ewidentnie Jednocześnie Dezambiguator)*, the English *Shallow Parsing and Eminently Judicious Disambiguation*, the German *Syntaktisches Parsing Entwicklungsystem Jedoch mit Disambiguierung*, and the French *Super Parseur Et Jolie Désambiguïsation*. Originally the system was called *Shallow Parsing And Disambiguation Engine* and abbreviated to *SPADE*. However, to avoid confusion with the other SPADE parsing system (*Sentence-level PArsing for DiscoursE*, http://www.isi.edu/licensed-sw/spade/), it was renamed to Spejd.

are decoupled, i.e., a rule may be adorned with instructions to the effect that a structure is built even when the relevant unification fails.

After a brief presentation of some related work in §2, we present the formalism in §3 and its implementation in §4, with §5 concluding the paper.

2 Background and Related Work

Syntactic parsers differ in whether they assume morphosyntactically disambiguated or non-disambiguated input: deep parsing systems based on unification usually allow for ambiguous input, while shallow (or partial) parsers usually expect fully disambiguated input. Some partial parsing systems (e.g., [10], [6], [1], [14]) allow for the interweaving of disambiguation and parsing.

[5] present a unified formalism for disambiguation and *dependency* parsing. Since dependency parsing in that approach is fully *reductionistic*, i.e., it assumes that all words have all their possible syntactic roles assigned in the lexicon and it simply rejects some of these roles, that formalism is basically a pure disambiguation formalism. In contrast, the formalism described below is *constructive*: it groups constituents into larger constituents.

Previous work that comes closest to our aims is reported in [8, 9] and [7], where INTEX local grammars [15], normally used for disambiguation, are the basis for a system that recognises various kinds of noun phrases and handles agreement within them. However, it is not clear whether these extensions lead to a lean formalism comparable to the formalism presented below.

3 Formalism

3.1 The Basic Format

In the simplest case, each rule consists of up to 4 parts marked as `Left`, `Match`, `Right` and `Eval`:

```
Left:  ;
Match: [pos~~"prep"] [base~"co|kto"];
Right: ;
Eval:  unify(case,1,2); group(PG,1,2);
```

The rule means:

1. find a sequence of two tokens[2] such that the first token is an unambiguous preposition (`[pos~~prep]`), and the second token is a form of the lexeme CO 'what' or KTO 'who' (`[base~"co|kto"]`);
2. if there exist interpretations of these two tokens with the same value of case, reject all interpretations of these two tokens which do not agree in case (cf. `unify(case,1,2)`);
3. if the above unification did not fail, mark thus identified sequence as a syntactic group (`group`) of type `PG` (prepositional group), whose syntactic head is the first token (`1`) and whose semantic head is the second token (`2`; cf. `group(PG,1,2)`).

`Left` and `Right` parts of a rule, specifying the context of the match, may be empty; in such a case they may be omitted.

Note that, unlike in typical unification-based formalisms, unification and grouping are decoupled here. In particular, it is possible to reverse the order of `group` and `unify` in the rule above: in this case the rule will always mark the match as a group and only subsequently unify case values, if possible. This feature of the formalism is useful, e.g., for dealing with systematic deficiencies of the morphological analyser used.

3.2 Matching (Left, Match, Right)

The contents of parts `Left`, `Match` and `Right` have the same syntax and semantics. Each of them may contain a sequence of the following specifications:

1. **token specification**, e.g., `[pos~~"prep"]` or `[base~"co|kto"]`; these specifications are compatible with segment specifications of the Poliqarp [4] corpus search engine as specified in [11]; in particular, a specification like `[pos~~"subst"]` says that *all* morphosyntactic interpretations of a given token are nominal (substantive), while `[pos~"subst"]` means that there *exists* a nominal interpretation of a given token;
2. **group specification**, extending the Poliqarp query language as proposed in [12], e.g., `[semh=[pos~~"subst"]]` specifies a syntactic group whose semantic head is a token whose all interpretations are nominal;
3. one of the **following specifications**: `ns`: no space; `sb`: sentence beginning; `se`: sentence end;
4. an **alternative** of such sequences in parentheses, e.g., `([pos~~"subst"] | [synh=[pos~~"subst"]] se)`.

[2] A terminological note is in order, although its full meaning will become clear only later: by *segment* we understand the smallest interpreted unit, i.e., a sequence of characters together with their morphosyntactic interpretations (lemma, grammatical class, grammatical categories); *syntactic word* is a non-empty sequence of segments and/or syntactic words marked by the action `word`; *token* is a segment or a syntactic word; *syntactic group* (in short: *group*) is a non-empty sequence of tokens and/or syntactic groups, marked as an entity by the action `group`; *syntactic entity* is a token or a syntactic group.

Additionally, each such specification may be modified with one of the three **regular expression quantifiers**: `?`, `*` and `+`.

An example of a possible value of `Left`, `Match` or `Right` might be:

```
[pos~~"adv"] (
     [pos~~"prep"] [pos~"subst"] ns? [pos~"interp"]? se |
     [synh=[pos~~"prep"]]
)
```

The meaning of this specification is: find an adverb followed by a prepositional group, where the prepositional group is specified as either a sequence of an unambiguous preposition and a possible noun at the end of a sentence, or an already recognised prepositional group.

3.3 Conditions and Actions (Eval)

The `Eval` part contains a sequence of Prolog-like predicates evaluating to true or false; if a predicate evaluates to false, further predicates are not evaluated and the rule is aborted. Almost all predicates have side effects, or actions. In fact, many of them always evaluate to true, and they are 'evaluated' solely for their side effects. In the following, we will refer to those predicates which may have side effects as *actions*, and to those which may evaluate to false as *conditions*.

There are two types of actions: morphosyntactic and syntactic. While morphosyntactic actions delete some interpretations of specified tokens, syntactic actions group entities into syntactic words (consecutive segments which syntactically behave like single words, e.g., multi-segment named entities, etc.) or syntactic groups.

Natural numbers in predicates refer to tokens matched by the specifications in `Left`, `Match` and `Right`. These specifications are numbered from 1, counting from the first specification in `Left` to the last specification in `Right`. For example, in the following rule, there should be case agreement between the adjective specified in the left context and the adjective and the noun specified in the right context (cf. `unify(case,1,4,5)`), as well as case agreement (possibly of a different case) between the adjective and noun in the match (cf. `unify(case,2,3)`).

```
Left:  [pos~~"adj"];
Match: [pos~~"adj"][pos~~"subst"];
Right: [pos~~"adj"][pos~~"subst"];
Eval:  unify(case,2,3); unify(case,1,4,5);
```

The exact repertoire of predicates still evolves, but currently the following are defined:

`agree(<cat> ...,<tok>,...)` — a condition checking if the grammatical categories (`<cat> ...`) of tokens specified by subsequent numbers (`<tok>,...`) agree. It takes a variable number of arguments: the initial arguments, such as `case` or `gender`, specify the grammatical categories that should *simultaneously* agree, so the condition `agree(case gender,1,2)` is stronger than the sequence of conditions: `agree(case,1,2), agree(gender,1,2)`. Subsequent arguments of `agree` are natural numbers referring to entity specifications that should be taken into account when checking agreement.

`unify(<cat> ...,<tok>,...)` — a condition (and, simultaneously, an action) which checks agreement, just as `agree`, but also deletes interpretations that do not agree.

`delete(<cond>,<tok>,...)` — delete all interpretations of specified tokens matching the specified condition (for example `delete(case~"gen|acc",1)`).

`leave(<cond>,<tok>,...)` — leave only the interpretations matching the specified condition.

`add(<tag>,<base>,<tok>)` — add to the specified token the interpretation `<tag>` with the base form `<base>`.

`word(<tag>,<base>)` — create a new syntactic word comprising of all tokens matched by the `Match` specification, and assign it the given tag and base form.

In both cases, `<tag>` may be a simple complete tag, e.g., `conj` for a conjunction or `adj:pl:acc:f:sup` for a superlative degree feminine accusative plural form of an adjective, but it may also be a specification of a number of tags. For example, `add(subst:number*:gen:m3, procent, 1)` will add 2 (one for each number) nominal genitive inanimate masculine interpretations to the token referred by `1`, in both cases with the base form PROCENT 'per cent'. Moreover, the sequence `<tag>,<base>` may be repeated any number of times, so, e.g., the abbreviation *fr.* may be turned into a syntactic word representing any of the 2×7 number/case values of the noun FRANK 'franc' (the currency), or any of the 2×7×5 number/case/gender values of the (positive degree) adjective FRANCUSKI 'French':

```
Match: [orth~"fr"] ns [orth~"\."];
Eval:  word(subst:number*:case*:m3,frank;
            adj:number*:case*:gender*:pos,francuski);
```

`<base>` is a sequence of static strings and references to tokens' base or orthographic forms. The base form of a new syntactic word is created by evaluating and concatenating all elements of the sequence, for example the action `word(qub, "po " 2.orth)` creates a new base form by concatenating PO, space and the orthographic form of the second token.

`word(<tok>, <tag_fragment>,<base>)` — create a new syntactic word comprising of all tokens matched by the `Match` specification, by copying all interpretations of the token `<tok>`, adding or replacing `<tag_fragment>` (for example a negation marker) in each interpretation of that token, and possibly modyfying the respective base forms. The original interpretations are not modified, if both `<base>` and `<tag_fragment>` are empty, as in the following rule, which turns the three tokens of *„Rzeczpospolita”* (i.e., *„*, *Rzeczpospolita* and *”*) into a single word with exactly the same interpretations (and base form) as *Rzeczpospolita* (the name of a Polish newspaper):

```
Match: [orth~"„"] ns? [] ns? [orth~"”"];
```

```
Eval:  word(3,,);
```

The orthographic form of the newly created syntactic word is always a simple concatenation of all orthographic forms of all tokens immediately contained in that syntactic word, taking into account information about space or its lack between consecutive tokens.

`group(<type>,<synh>,<semh>)` — create a new syntactic group with syntactic head and semantic head specified by numbers. The `<type>` is the categorial type of the group (e.g., `PG`), while `<synh>` and `<semh>` are references to appropriate token specifications in the `Match` part. For example, the following rule may be used to create a numeral group, syntactically headed by the numeral and semantically headed by the noun:[3]

```
Left:  [pos~~"prep"];
Match: [pos~~"num"] [pos~~"adj"]*
       [pos~~"subst"];
Eval:  group(NumG,2,4);
```

Of course, rules should be constructed in such a way that references `<synh>` and `<semh>` refer to specifications of single entities, e.g., `[case~~"nom"]` or `([pos~~"subst"] | [synh=[pos~~"subst"]])` but not, say, `[case~~"nom"]+`

In all these predicates, a reference to a token specification takes into account all tokens matched by that specification, so, e.g., in case `1` refers to the specification `[pos~~adj]*`, the action `unify(case,1)` means that all the adjectives matched must be rid of all interpretations whose case is not shared by all of them.

Moreover, the numbers in all predicates are interpreted as referring to tokens; when a reference is made to a syntactic group, the action is performed on the syntactic head of that group. For example, assuming that the following rule finds a sequence of a nominal segment, a multi-segment syntactic word and a nominal group, the action `unify(case,1)` will result in the unification of case values of the first segment, the syntactic word as a whole and the syntactic head of the group.

```
Match: ([pos~~"subst"]|[synh=[pos~~"subst"]])+;
Eval:  unify(case,1);
```

The only exception to this rule is the semantic head parameter in the `group` action; when it references a syntactic group, the semantic, not syntactic, head is inherited.

4 Implementation

Since the formalism described above is novel and to some extent still evolving, its implementation had to be not only reasonably fast, but also easy to modify and maintain. This section briefly presents such an implementation.

[3] A rationale for distinguishing these two kinds of heads is given in [12].

The implementation has been released under the GNU General Public License (version 3). The release address is `http://nlp.ipipan.waw.pl/Spejd/`.

4.1 Input and Output

The parser implementing the specification above currently takes as input the version of the XML Corpus Encoding Standard assumed in the IPI PAN Corpus of Polish (`http://korpus.pl/`; [11]). Rules may modify the input in two possible ways. First, morphosyntactic actions may reject certain interpretations of certain tokens; such rejected interpretations are marked by the attribute `disamb_sh="0"` added to `<lex>` elements representing these interpretations. Second, syntactic actions modify the input by adding `<syntok>` and `<group>` elements, marking syntactic words and groups.

For example, the rule given at the top of §3.1 above may be applied to the following input sequence (slightly simplified in irrelevant aspects; e.g., the token *co* actually has 3 more interpretations, apart from the two given below) of two tokens *Po co* 'why, what for', lit. 'for what', where *Po* is a preposition which either combines with an accusative argument or with a locative argument, while *co* is ambiguous between, *inter alia*, a nominative/accusative noun:

```
<tok id="tA5">
 <orth>Po</orth>
 <lex><base>po</base>
      <ctag>prep:acc</ctag></lex>
 <lex><base>po</base>
      <ctag>prep:loc</ctag></lex>
</tok>
<tok id="tA6">
 <lex><base>co</base>
      <ctag>subst:sg:nom:n</ctag></lex>
 <lex><base>co</base>
      <ctag>subst:sg:acc:n</ctag></lex>
</tok>
```

The result should have the following effect (bits added by the rule are *emphasised*):

```
<group type="PG"
       synh="tA5" semh="tA6">
<tok id="tA5">
 <orth>Po</orth>
 <lex><base>po</base>
      <ctag>prep:acc</ctag></lex>
 <lex disamb_sh="0"><base>po</base>
      <ctag>prep:loc</ctag></lex>
</tok>
<tok id="tA6">
```

```
 <lex disamb_sh="0"><base>co</base>
      <ctag>subst:sg:nom:n</ctag>
 </lex>
 <lex><base>co</base>
      <ctag>subst:sg:acc:n</ctag></lex>
</tok>
</group>
```

4.2 Algorithm Overview

During the initialisation phase, the parser loads the external tagset specification and the rules, and converts the latter to a set of compiled regular expressions and actions/conditions. Then, input files are parsed one by one (for each input file a corresponding output file containing parsing results is created).

To reduce memory usage, parsing is done by chunks defined in the input files, such as sentences or paragraphs. In the remainder of the paper we assume the chunks are sentences.

The parser concurrently maintains two representations for each sentence: **1)** an object-oriented syntactic entity tree, used for easy manipulation of entities (for example, for disabling certain interpretations or creating new syntactic words) and preserving all necessary information to generate the final output; **2)** a compact string for quick regexp matching, containing only the information important for the rules which have not been applied yet.

Tree Representation The entity tree is initialised as a flat (one level deep) tree with all leaves (segments and possibly special entities, like no space, sentence beginning, sentence end) connected directly to the root. Application of a syntactic action means inserting a new node (syntacting word or group) to the tree, between the root and some of the existing nodes. As the parsing proceeds, the tree changes its shape: it becomes deeper and narrower.

Morphosyntactic actions do not change the shape of the tree, but also reduce the string representation length by deleting from that string certain interpretations. The interpretations are preserved in the tree to produce the final output, but are not relevant to further stages of parsing.

String Representation The string representation is a compromise between XML and binary representation, designed for easy, fast and precise matching, with the use of existing regular expression libraries.[4] The representation describes the top level of the current state of the sentence tree, including only the

[4] Two alternatives to this approach were considered: **1)** building a custom finite state automata on binary representation: our previous experience shows that while this may lead to an extremely fast search engine, it is at the same time costly to maintain; **2)** operating directly on XML files: the strings to search would be longer, and matching would be more complex (especially for requirements including negation); a prototype of this kind was written in Perl and parsing times were not acceptable.

information that may be used by rule matching. For each child of the tree root, the following information is preserved in the string: type (token / group / special) and identifier (for finding the entity in the tree in case an action should be applied to it). The ensuing part of the string depends on the type of the child: for a token, it is orthographic forms and a list of interpretations; for a group — number of heads of the group and lists of interpretations for the syntactic and semantic head.

Because the tagset used in the IPI PAN Corpus is intended to be human-readable, the morphosyntactic tags are fairly descriptive and, as a result, they are rather long. To facilitate and speed up pattern matching, each tag is converted to a relatively short string of fixed width. In the string, each character corresponds to one morphological category from the tagset (first part of speech, then number, case, gender, etc.) as, for example, in the Czech positional tag system [3]. The characters — upper- and lowercase letters, or 0 (zero) for categories non-applicable to a given part of speech — are assigned automatically, on the basis of the external tagset definition read at initialisation. A few possible correspondences are presented in Table 4.2.

Table 1. Examples of tag conversion between human-readable and inner positional tagset.

IPI PAN tag	fixed length tag
`adj:pl:acc:f:sup`	`UBDD0C0000000`
`conj`	`B000000000000`
`fin:pl:sec:imperf`	`bB00B0A000000`
`subst:pl:nom:m1`	`NBAA000000000`

Matching (Left, Match, Right) The conversion from the `Left`, `Match` and `Right` parts of the rule to a regular expression over the string representation is fairly straightforward. Two exceptions — regular expressions as morphosyntactic category values and the distinction between existential and universal quantification over interpretations — are described in more detail below.

First, the rule might be looking for a token whose grammatical category is described by a regular expresion. For example, `[gender~~"m."]` should match personal masculine (also called virile; `m1`), animal masculine (`m2`), and inanimate masculine (`m3`) tokens; `[pos~~"ppron[123]+|siebie"]` should match all pronouns (`ppron12`, i.e., first or second person personal pronouns, `ppron3`, i.e., third person personal pronouns, or forms of the reflexive/reciprocal pronoun SIEBIE, which happens to have a separate grammatical class in the IPI PAN Corpus, called `siebie`); `[pos!~~"adj.*"]` should match all segments except for (various classes of) adjectives; etc. Because morphosyntactic tags are converted to fixed length representations, the regular expressions also have to be converted before compilation.

To this end, the regular expression is matched against all possible values of the given category. Since, after conversion, every value is represented as a single character, the resulting regexp can use square bracket notation for character classes to represent the range of possible values.

The conversion can be done only for attributes with values from a well-defined, finite set. Since we do not want to assume that we know all the text to parse before the compilation of the rules, we assume that the dictionary is infinite. The assumption makes it difficult to convert requirements with negated `orth` or `base` (for example `[orth!~"[Nn]ie"]`). As for now, such requirements are not included in the compiled regular expression, but instead handled by special predicates in the `Eval` part.

Second, a segment may have many interpretations and sometimes a rule may apply only when all the interpretations meet the specified condition (for example `[pos~~"subst"]`), while in other cases one matching interpretation should be enough to trigger the rule (`[pos~"subst"]`).

In the string interpretation, `<` and `>` were chosen as convenient separators of interpretations and entities, because they should not appear in the input data (they have to be escaped in XML). In particular, each representation of a fixed length tag is preceded by `<`. Assuming that tags representing a nominal (`subst`) are translated into fixed length string starting with an `N`, the universal specification `[pos~~"subst"]` will be translated into the regular expression `(<N[^<>]+)+`, while the existential specification `[pos~"subst"]` will be translated into `(<[^<>]+)*(<N[^<>]+)(<[^<>]+)*`.

Of course, a combination of existential and universal requirements is a valid requirement as well, for example: `[pos~~"subst" case~"gen|acc"]` (all interpretations noun, at least one of them in genitive or accusative case) should translate to: `(<N[^<>]+)*(<N.[BD][^<>]+)(<N[^<>]+)` (if genitive and accusative translate to `B` and `D`).

Conditions and Actions (Eval) As described in §3.3, when a match is found, the parser evaluates a sequence of predicates connected to the given rule. Each predicate may be a condition with no side effects involved, a morphosyntactic action or a syntactic action. The parser executes the sequence until it encounters a predicate which evaluates to false (for example, unification of cases fails).

The actions affect both the tree and the string representation of the parsed sentence. The tree is updated instantly (the cost of update is constant or linear with respect to match length), but the string update (cost linear to sentence length) is delayed until it is really needed (at most once per rule).

4.3 Efficiency

The system has been implemented in Java. So far, it has been tested in two practical applications: valence acquisition from the morphosyntactically annotated IPI PAN Corpus of Polish [13] and sentiment analysis of product reviews [2].

When given a set of over 90 rules of varying complexity, ♠ processed a 12MB XML file containing over 56 thousand words in about 42 seconds, which gives the average of about 1340 words per second (as measured on a contemporary Intel Core2Duo T7200 laptop). In the process, almost 6800 syntactic words and over 5600 syntactic groups were marked. While parsing times increase with the size of the grammar, they are still acceptable.

5 Conclusion

The system presented, ♠, is perhaps unique in allowing the grammar developer to encode morphosyntactic disambiguation and shallow parsing instructions in the same unified formalism, possibly in the same rule. The formalism is more flexible than either the usual shallow parsing formalisms, which assume disambiguated input, or the usual unification-based formalisms, which couple disambiguation (via unification) with structure building. While the rule sets so far have been prepared for parsing of Polish, ♠ is fully language-independent and we hope it will also be useful in the processing of other languages.

References

1. Aït-Mokhtar, S., Chanod, J.-P., Roux C.: Robustness beyond shallowness: incremental deep parsing. Natural Language Engineering, 8:121–144 (2002)
2. Buczyński, A., Wawer, A.: Automated classification of product review sentiments using bag of words and Sentipejd. In: Kłopotek, M.A., Przepiórkowski, A., Wierzchoń, S.T., Trojanowski, K. (eds.) Intelligent Information Systems. Warsaw: Institute of Computer Science, Polish Academy of Sciences (2008)
3. Hajič, J., Hladká, B.: Probabilistic and rule-based tagger of an inflective language - a comparison. In: Proceedings of the ANLP'97. Washington, DC (1997)
4. Janus, D., Przepiórkowski, A.: Poliqarp: An open source corpus indexer and search engine with syntactic extensions. In: Proceedings of ACL 2007 Demo Session (2007)
5. Karlsson, F., Voutilainen, A., Heikkilä, J., Anttila, A. (eds.): Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text. Berlin: Mouton de Gruyter (1995)
6. Marimon, M., Porta, J.: PoS disambiguation and partial parsing bidirectional interaction. In: Proceedings of the Third International Conference on Language Resources and Evaluation, LREC 2000. Athens, Greece: ELRA (2000)
7. Nenadić, G.: Local grammars and parsing coordination of nouns in Serbo-Croatian. In Proceedings of Text, Dialogue and Speech (TSD) 2000. Springer-Verlag (2000)
8. Nenadić, G., Vitas, D.: Formal model of noun phrases in Serbo-Croatian. BULAG, 23. Presses de l'Université de Franche-Comté, Besançon, France (1998)
9. Nenadić, G., Vitas, D.: Using local grammars for agreement modeling in highly inflective languages. In: Proceedings of Text, Dialogue and Speech (TSD) (1998)
10. Neumann, G., Braun, C., Piskorski, J.: A divide-and-conquer strategy for shallow parsing of German free texts. In Proceedings of ANLP-2000. Seattle, Washington (2000)
11. Przepiórkowski, A.: The IPI PAN Corpus: Preliminary version. Warsaw: Institute of Computer Science, Polish Academy of Sciences (2004)
12. Przepiórkowski, A.: On heads and coordination in valence acquisition. In: Gelbukh, A. (ed.), Computational Linguistics and Intelligent Text Processing (CICLing 2007), Lecture Notes in Computer Science. Berlin: Springer-Verlag (2007)

13. Przepiórkowski, A.: Powierzchniowe przetwarzanie języka polskiego. Akademicka Oficyna Wydawnicza EXIT. Warsaw (2008)
14. Schiehlen, M.: Experiments in German noun chunking. In: Proceedings of the 19th International Conference on Computational Linguistics (COLING 2002). Taipei (2002)
15. Silberztein, M.: INTEX: a corpus processing system. In: Fifteenth International Conference on Computational Linguistics (COLING '94). Kyoto, Japan (1994)