

# An Open Source Tool for Partial Parsing and Morphosyntactic Disambiguation<sup>\*</sup>

Adam Przepiórkowski and Aleksander Buczyński

Institute of Computer Science  
Polish Academy of Sciences,  
Warsaw, Poland  
adamp@ipipan.waw.pl, olekb@ipipan.waw.pl  
<http://nlp.ipipan.waw.pl/>

**Abstract.** This article presents a formalism and an open source implementation of a new tool for simultaneous partial parsing and morphosyntactic disambiguation and correction. We argue that, contrary to the common pipeline approach, where morphosyntactic tagging is fully accomplished before shallow or partial parsing, both tasks are best approached in parallel. This has been suggested before, and formalisms which allow for the interweaving of partial parsing and morphosyntactic disambiguation have been proposed. Our approach is novel in that a fully uniform formalism is presented, and a single grammar rule may contain structure-building operations, as well as morphosyntactic correction and disambiguation operations. The formalism has been implemented in Java and is now available under the GNU General Public License.

## 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, called *Shallow Parsing and Disambiguation Engine*, initially abbreviated to “SPADE”, but — because of the existence of an earlier parsing system called SPADE<sup>1</sup> — now abbreviated to “♠” (Unicode character 0x2660) and to the more international acronym “Spejd”<sup>2</sup>, pronounced the same way as *spade*. The formalism is essentially a cascade of regular grammars, where (currently) each regular grammar is

---

<sup>\*</sup> This article is an improved and extended version of [1] and it is up to date as of 7th November 2007.

<sup>1</sup> *Sentence-level PArsing for DiscoursE*, <http://www.isi.edu/licensed-sw/spade/>.

<sup>2</sup> Polish: *Składniowy Parser (Ewidentnie Jednocześnie Dezambiguator)*, German: *Syntaktisches Parsing Entwicklungssystem Jedoch mit Disambiguierung*, French: *Super Parseur Et Jolie Désambiguisation*.

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 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., [2], [3], [4], [5]) allow for the interweaving of disambiguation and parsing.

[6] 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 [7, 8] and [9], where INTEX local grammars [10], 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

Each rule consists of up to 5 parts marked as **Rule**, **Left**, **Match**, **Right** and **Eval**:

```
Rule:  "some rule id here"
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<sup>3</sup> such that the first token is an unambiguous preposition ([pos~~prep]), and the second token is a form

---

<sup>3</sup> 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 its morphosyntactic interpretations (lemma, grammatical

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 tokens which do not agree in case (cf. `unify(case,1,2)`); **3**) 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. The other fields, i.e., `Rule`, `Match` and `Eval` are obligatory.

Note that, apart from `Rule`, all fields end in a semicolon, and also particular actions in `Eval` are separated by semicolons. Comments may be added to rules, starting with the hash character “#”, and fields may be split across lines, so a rule fully equivalent to the rule above may look as follows:

```
# a trivial rule for the purpose of this article only
Rule: "some rule id here"
Match: [pos~~"prep"]      # a sure preposition
      [base~"co|kto"];    # a form of CO or KTO
Eval:  unify(case,1,2);   # must agree in case
      group(PG,1,2);     # Prepositional Group
```

Although the `Rule` part, specifying the identifier of the rule, is obligatory, we will omit it below in the interest of brevity.

### 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 adhere to segment specifications of the Poliqarp [11] corpus search engine as defined in [12];<sup>4</sup> in particular, a specification like `[pos~~"subst"]` says

---

class, grammatical categories); *syntactic word* is a non-empty sequence of segments and/or syntactic words marked as an entity 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.

<sup>4</sup> One difference between the query language of Poliqarp and the language of token specifications assumed here is that here, 1) multiple conditions within a single token specification may be combined only with the conjunction operator, while Poliqarp also allows disjunction, 2) conjunction is expressed by the operator `&&`, as in `[case~nom && number~sg]`, as opposed to Poliqarp’s `&`, which reflects a subtle difference in the semantics of these operators (the specification just given says that there must be an interpretation which is *simultaneously* nominative and singular), while the corresponding Poliqarp query `[case~nom & number~sg]` would find tokens which have a nominative interpretation and a — possibly different — singular interpretation), 3) negation may only be used in negated versions of the operators `~` and `~~`, i.e., `!~` and `!~~`, while in Poliqarp negation may outscope conjunction and disjunction.

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 [13], e.g., `[semh=[pos~~"subst"]]` specifies a syntactic group whose semantic head is a token whose all interpretations are substantive (i.e., 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)`. Additionally, **5) 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 or add 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);
```

Currently the following predicates 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>, ...*) — an action which leaves only those interpretations of tokens *<tok>, ...* which agree in *<cat> ...* in case such agreement is possible (i.e., in case **agree**(*<cat> ... , <tok>, ...*) evaluates to true), and does nothing otherwise.

**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>*. The *<tag>* specification may be a specific tag, e.g., **subst:sg:nom:f**, or it may contain abbreviations of the form *<cat>\**, which expand to all possible values of the given category. For example, given the tagset of Polish described in [12], the action **add**(**subst:number\*:case\*:n**, "emu", 1) adds 14 interpretations to the token referred to by 1, all with the base form EMU, corresponding to the 2 grammatical numbers and 7 grammatical cases in Polish.

**set**(*<tag>, <base>, <tok>*) — delete all interpretations of the specified token and set the interpretation(s) of that token as *<tag>* with the base form *<base>*. Equivalent to the sequence: **delete**(, *<tok>*); **add**(*<tag>, <base>, <tok>*).

**word**(*<tag>, <base>*) — create a new syntactic word comprising all tokens matched by the **Match** specification, and assign it the given tag and base form. The sequence *<tag>, <base>* may be repeated any number of times (separated by semicolons), 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>* may be a specific string, as "frank" or "francuski" in the rules above, or — more generally — it may be a concatenation of strings, where each string is given directly or is specified as *<n>.orth* (the orthographic form of the token corresponding to the *n*th specification) or *<n>.base* (a base form of such a token).<sup>5</sup> For example, the following rule, finding a syntactic word of class *liczba* 'number',

<sup>5</sup> In case of *<n>.base*, the assumption is that all interpretations of the *n*th token have the same base form, so it does not matter which one is taken. If the token has

creates the base form of this syntactic word by concatenating 4 strings: the orthographic form of the token matched by `[orth~"[0-9]+"]`, the space (cf. " "), and the orthographic forms matching the specifications `[orth~"mln|mld"]` and `(ns? [orth~"."])?`, respectively.

```
Match: [orth~"[0-9]+" ] [orth~"mln|mld"] (ns [orth~"."])?;
Eval:  word(liczba, 1.orth " " 2.orth 3.orth);
```

Since such operations are frequent, the special base form specification `0.orth` takes all orthographic forms of all tokens in the match and concatenates them, taking into account information about no spaces between particular tokens, so a rule equivalent to the one above would be:

```
Match: [orth~"[0-9]+" ] [orth~"mln|mld"] (ns [orth~"."])?;
Eval:  word(liczba, 0.orth);
```

`word(<tok>,<tag_modification>,<base_modification>)` — create a new syntactic word comprising all tokens matched by the `Match` specification, by taking all interpretations of the token `<tok>` and modifying their tags and bases according to `<tag_modification>` and `<base_modification>`, respectively. In the simplest case, a `<..._modification>` may be empty and the corresponding tag or base form is copied to the new interpretation. If not empty, `<base_modification>` may be a concatenation of specific strings and the special specification `base`; for example, `word(1,"nie " base)` creates base forms by adding the prefix “*nie* ” to the base forms of the first token matched. On the other hand, `<tag_modification>` may contain the value of a grammatical category; for any interpretation of the token specified by `<tok>` whose grammatical class allows for the grammatical category with value specified by `<tag_modification>`, this value is inserted into that interpretation. For example, the following rule, specifying that the left context is not the negative marker *nie* (or *Nie*), may be used to construct simple syntactic words consisting of single verbs, where the verbal interpretations are modified by adding the *aff* (affirmative) value of the (optional) *negation* grammatical class:

```
Left: [orth!~"[Nn]ie"];
Match: [pos~~"praet|fin|impt|imps|inf"];
Eval: word(2, aff, );
```

For both versions of `word`, 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

---

different base forms, this operation is indeterminate, as any of those base forms may be taken.

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:<sup>6</sup>

```
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., to `([pos~~"subst"] | [synh=[pos~~"subst"]])` but not, say, to `[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.

### 4.1 Input and Output

The parser implementing the specification above currently takes as input the version of the XML Corpus Encoding Standard [14] assumed in the IPI PAN Corpus of Polish (<http://korpus.p1/>; [12]). 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

---

<sup>6</sup> A rationale for distinguishing these two kinds of heads is given in [13].

`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.<sup>7</sup> The representation describes the top level of the current state of the sentence tree, including only the 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, tags are converted to strings of fixed length. In such a string, each character corresponds to one morphological category from the tagset (first part of speech, then number, case,

---

<sup>7</sup> 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.

gender, etc.) as, for example, in the Czech positional tag system [15]. 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 1.

IPI PAN tag	fixed length tag
adj:pl:acc:f:sup	UBDD0C00000000
conj	B0000000000000
fin:pl:sec:imperf	bB00B0A0000000
subst:pl:nom:m1	NBAA0000000000

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

**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 expression. 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. Therefore, `orth` and `base` requirements are not converted. Requirements with negated `orth` and `base` have to use the zero-width negative lookahead construct.

**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 fixed length tag representation is preceded by <. Assuming that nominal **subst** tags 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 the update is linear with respect to the match length or the number of interpretations of tokens involved, depending on the action type), 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 described above has been implemented in Java. When given a set of 167 rules of varying complexity, ♠ processed a 34MB XML file containing over 174 thousand segments (almost 16 thousand sentences) in about 4 minutes, which gives the average of about 700 words per second (as measured on an Intel Core2Duo T7200 laptop). In the process, over 21 thousand syntactic words and over 22 thousand syntactic groups were marked. While parsing times increase with the size of the grammar, they are still acceptable, given the intended use of the system for the off-line shallow parsing of a corpus.

### 4.4 Availability

The tool described here has just become available under the GNU General Public License (version 2) and the current paper is the official article announcing the public availability of this tool. In case it is accepted to CICLing 2008, this section will briefly describe the release and it will give the URL from which ♠ may be downloaded (now withheld for the reasons of anonymity).

## 5 Conclusion

The system presented above, ♠, 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 a rule set is currently prepared for the parsing of the IPI PAN Corpus of Polish, ♠ is fully language-independent and we hope it will also be useful in the processing of other languages.

## References

1. Przepiórkowski, A., Buczyński, A.: ♠: Shallow Parsing and Disambiguation Engine. In Vetulani, Z., ed.: *Proceedings of the 3rd Language & Technology Conference*, Poznań, Poland (2007)
2. Neumann, G., Braun, C., Piskorski, J.: A divide-and-conquer strategy for shallow parsing of German free texts. In: *Proceedings of the 6th Applied Natural Language Processing Conference*, Seattle, WA, ACL (2000) 239–246
3. Marimon, M., Porta, J.: PoS disambiguation and partial parsing bidirectional interaction. [16]
4. Aït-Mokhtar, S., Chanod, J.P., Roux, C.: Robustness beyond shallowness: incremental deep parsing. *Natural Language Engineering* **8** (2002) 121–144
5. Schiehlen, M.: Experiments in German noun chunking. In: *Proceedings of the 19th International Conference on Computational Linguistics (COLING 2002)*, Taipei (2002)
6. Karlsson, F., Voutilainen, A., Heikkilä, J., Anttila, A., eds.: *Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text*. Mouton de Gruyter, Berlin (1995)
7. Nenadić, G., Vitas, D.: Formal model of noun phrases in Serbo-Croatian. *BULAG* **23** (1998) Presses de l'Université de Franche-Comté, Besançon, France.
8. Nenadić, G., Vitas, D.: Using local grammars for agreement modeling in highly inflective languages. In: *Proceedings of Text, Speech and Dialogue (TSD) 1998*. (1998) 91–96
9. Nenadić, G.: Local grammars and parsing coordination of nouns in Serbo-Croatian. In Sojka, P., Kopeček, I., Pala, K., eds.: *Text, Speech and Dialogue: Third International Workshop, TSD 2000*, Brno, Czech Republic, September 2000. Volume 1902 of *Lecture Notes in Artificial Intelligence*., Berlin, Springer-Verlag (2000) 57–62
10. Silberztein, M.: INTEX: a corpus processing system. In: *Fifteenth International Conference on Computational Linguistics (COLING '94)*, Kyoto, Japan (1994) 579–583
11. Janus, D., Przepiórkowski, A.: Poliqarp: An open source corpus indexer and search engine with syntactic extensions. In: *Proceedings of ACL 2007 Demo and Poster Sessions*. (2007) 85–88
12. Przepiórkowski, A.: The IPI PAN Corpus: Preliminary version. Institute of Computer Science, Polish Academy of Sciences, Warsaw (2004)

13. 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) 50–61
14. Ide, N., Bonhomme, P., Romary, L.: XCES: An XML-based standard for linguistic corpora. [16] 825–830
15. Hajič, J., Hladká, B.: Probabilistic and rule-based tagger of an inflective language - a comparison. In: Proceedings of the 5th Applied Natural Language Processing Conference, Washington, DC, ACL (1997) 111–118
16. ELRA: Proceedings of the Third International Conference on Language Resources and Evaluation, LREC 2000. In: Proceedings of the Third International Conference on Language Resources and Evaluation, LREC 2000, Athens, ELRA (2000)