WCCL: A Morpho-syntactic Feature Toolkit*

Adam Radziszewski, Adam Wardyński, and Tomasz Śniatowski

Institute of Informatics
Wrocław University of Technology
Wybrzeże Wyspiańskiego 27,
Wrocław, Poland

Abstract. The paper presents WCCL, a new formalism and toolkit for constructing morpho-syntactic features, a crucial task for many natural language processing algorithms. One existing solution, JOSKIPI, is analysed from two perspectives: features of the formalism as well as software engineering-related issues. Then we propose its successor. A short case study follows, exemplifying the improvement enabled by using rich features expressed with WCCL. The formalism is targeted at Polish, although it seems well suited for any inflectional language.

1 Background

Many NLP tasks may be performed using Machine Learning (ML) methods. The bulk of knowledge extraction can be offloaded to an underlying model (e.g. a classifier, clusterer). Even so, some amount of knowledge engineering is still required, e.g., it is necessary to provide a set of *features*, such as properties of consecutive word forms or simple syntactic dependencies between word forms.

The construction of features is especially challenging for Slavic languages, where syntactic dependencies are often marked by morphological features rather than a particular ordering of word forms [9]. The importance of morpho-syntactic features is confirmed by experiments. For instance, a successful adaptation of Brill's tagging algorithm to inflectional languages includes a decomposition of tags into parts corresponding to sets of selected morpho-syntactic categories [1]. Similarly, explicit tests for adjective–noun morphological agreement improved semantic similarity measures extracted from large corpora [7].

The ability to provide good morpho-syntactic features is an important factor in the construction of NLP systems. Furthermore, different tasks, such as morpho-syntactic tagging and lexico-semantic relation extraction, refer to similar types of information: parts-of-speech, values of morpho-syntactic categories, simple syntactic dependencies such as grammatical agreement. These observations suggest that a common reusable framework for expressing such features could contribute to a productivity increase in the NLP community.

There exists one formalism (with two implementations) that fits into the described application scenarios, namely the JOSKIPI language [6,8]. In the next section we argue that although JOSKIPI is conceptually well-founded, the design and implementation

* This work is financed by Innovative Economy Programme project POIG.01.01.02-14-013/09.

I. Habernal and V. Matoušek (Eds.): TSD 2011, LNAI 6836, pp. 434–441, 2011.

flaws render the software not applicable as a general feature extraction toolkit. Then we propose our solution: an extended formalism for expressing morpho-syntactic features, constraints and tagging rules, as well as its implementation — a toolkit designed to be used as a reusable component in language engineering.

2 JOSKIPI

JOSKIPI originates from TaKIPI, a morpho-syntactic tagger for Polish [6]. Detailed description of the formalism can be found in [8].

The formalism was devised for writing rules for the tagger, as well as a means to define basic functional expressions that would be building blocks for automatically extracted rules. The functional expressions (called *operators*) operate on sentences, understood as sequences of morphologically analysed (but possibly not fully disambiguated) tokens, with one token marked as the *current position*. Operators refer to tokens using position relative to the analysed centre: 0 is the current position, -1 is the first token positioned to the left, 1 is the first token to the right etc.; e.g. the operator `cas[-1]` returns the value of grammatical case for the token preceding the current position. The current position is moved sequentially through the sentence and the operators are evaluated every time.

Although such operators may be used to identify simple syntactic dependencies, JOSKIPI is not a shallow parser. The assumptions are different: a shallow parser captures and labels sequences of tokens, while JOSKIPI enables to evaluate functional expressions against given context. The values returned are to support decision making, whether the task is segmentation, disambiguation or relation finding.

2.1 Implementations and Applications

There exist two implementations of JOSKIPI (both available under GNU GPL):

1. The original implementation, a part of the TaKIPI tagger [6]. The implementation is provided as a C++ shared library with limited support for command-line processing.
2. An experimental Python implementation, a part of the Disaster system [12]. The original language is extended with means of referring to shallow syntactic annotation and features such as explicit variable assignment, regular expressions and boolean literals.

The language proved useful for several different applications. JOSKIPI predicates were used to filter a list of candidate Multi-Word Expressions acquired automatically [8]. Furthermore, JOSKIPI operators were used to generate features for extraction of lexico-semantic relations as performed in the SuperMatrix system [2]. The Python version was used for shallow parsing of Polish text [12].

2.2 Limitations and Design Flaws

Despite the successful applications, we argue that neither of the implementations is applicable as a general toolkit for feature generation due to design and implementation

flaws. The situation is worsened by the lack of technical documentation and the poor design of the available APIs.

The C++ implementation uses a hard-coded tagset definition, citing computational efficiency reasons. Unfortunately this causes even slight modifications to be labour-intensive, since one tagset attribute or value is defined in several places in the source code. This also makes the project not likely to survive for a longer time in the face of the changing standards, e.g. when the announced National Corpus of Polish with its own tagset [10] becomes available.

The language specification lacks strict data type definition. Even if an operator only makes sense when supplied with strings, it can be applied to a symbol set or even a boolean value. It is not possible to infer the type of an operator, or the type of a returned value. Strings are kept as indices to a global string table, which hinders distributed computation and is prone to memory leaks.

The Python implementation on the other hand is far too slow to be useful for processing large corpora. For instance, applying the expression `cas[0]` (i.e. retrieving the value of the grammatical case) to a 800 000-token corpus takes about 3.5 minutes on a 2.2 GHz Athlon 64 PC (around 4000 tokens per second).

3 Proposed Successor to JOSKIPI

We wanted to provide a toolkit that would capitalize on strengths of JOSKIPI formalism and fix its shortcomings. The new, redesigned and extended version of the feature description language is what we have called WCCL (for Wrocław Corpus Constraint Language). The implementation is written in C++ as a shared library, using and extending the Corpus2 library from the Maca system [13] for basic data structures and I/O.

3.1 Toolkit Features

The language supports configurable tagset read from definitions similar to those in [1]. Thus, it is not bound to any particular language. Configuring WCCL with various tagset definition works seamlessly with parts of the language that use tagset symbols. For example, if the employed tagset contains an attribute `tense` with three values: `pres`, `past`, `fut`, then `equal(tense[0], {pres})` is a valid predicate that checks whether the current token's `tense` is `pres`.

Variables in a parsed expression are available at run-time for inspection and assignment. This is especially useful in the case of string variables: it is possible to create only one operator instance and then manipulate the variables; in JOSKIPI it was necessary to wastefully create thousands of operators.

As JOSKIPI lacked a convenient means of referring to lexical information, we extended the formalism with possibility to refer to external lexicon files. Lexicons are essentialy lists of key–value pairs. They may be used by special `lex` operator to translate sets of strings into the corresponding values. Items not present in the lexicon are omitted in the output, so this mechanism can be used both for classification and for filtering of infrequent forms.

While not suitable for feature selection, disambiguation rules that were available in JOSKIPI are also present in WCCL — making it effectively a rule-based disambiguation engine with configurable tagset.

We have created a clear specification of WCCL file syntax. Whole file parsing enables users to put operators in named groups and subsequently access them in a convenient fashion. This directly supports creation of feature sets for classification. The syntax also allows a header for lexicon imports and a section for tagging rules.

Support for parallel processing is provided. The operators have no state apart from a well-separated container for variables, and as such a complex operator can be shared between threads, potentially reducing memory use.

WCCL also provides usable command-line utilities, e.g., for evaluation of multiple operators against a corpus (producing feature values). The utilities may be used in tandem with Maca analysis and conversion utilities through input-output piping. This way we achieve the architectural model of maximal component decoupling that is recommended for NLP toolkits [5]. In addition, the API is available both as a native C++ code and simple Python wrappers that enable rapid NLP application development and fast prototyping.

3.2 WCCL Operators

WCCL is strongly-typed: all expressions have a well-defined type. The range of available types currently consists of positions (integers), string sets, boolean values and tagset symbol sets. Variables of all the types are supported, in contrast to JOSKIPI, which supported only variables of the position type. Each type has a defined syntax for variable definition and literals, which enables automatic type inference in functions such as `equal`. The subsequent enhancements include an explicit functional-style *if* statement with inferable type.

WCCL operators work in similar fashion to the aforementioned JOSKIPI operators. To an extent, they are a superset of JOSKIPI functionality as we did want existing rules to be easily applicable to the new, extended formalism. Below we enumerate types of operators in WCCL.

1. Constants for all types.
   (a) Configurale tagset symbols: symbols of grammatical classes (roughly, parts-of-speech) and values of grammatical categories as defined in given tagset (e.g. the tagset of the IPI PAN Corpus (IPIC) [11]). Examples of such operators include: `{}` (empty set); `{nmb}` (*number* category, equivalent to `{sg, pl}` - *singular* and *plural*); `{f, n}` (values of *feminine* and *neuter* from the *gender* category, excluding masculine values)
   (b) Set of strings, e.g. `[]` (empty set); `"water"` (single string; equivalent to `["water"]`); `["ice", "water"]` (constant string set with two values)
   (c) Position, e.g. `begin` (begining of a sentence); `end` (end of a sentence); `0` (current position); `-2` (position second to the left from current position)
   (d) Boolean, `True` or `False`
2. Retrieving value of variables for all types, with names prefixed according to the type, e.g. `$Pos` (position variable named *Pos* - no prefix for this type); `$s:S`

(string set variable named *S*); `$t:T` (variable *T* for a set of tagset symbols); `$b:F` (boolean variable *F*).

3. Retrieving values of morpho-syntactic categories. Such operators are defined for the grammatical class and all the tagset attributes with names depending on the tagset given. They take a position as an argument and return set of values for given category from all lexemes of a token pointed by the position. E.g.: `cas[0]` (returns value of *case* category); `class[$V]` (returns word class of token pointed by position variable named *V*).
4. Retrieving string values: lemmas or orthographic forms. Again, as lemmas may be ambiguous, sets are returned. E.g: `orth[0]` (orthographic form of token at current position); `base[2]` (lemmas for the second token to the right from the current one).
5. Simple predicate constraints that allow testing whether values returned by various operators satisfy a relation. E.g. `equal(orth[0], "ice")` (is the orthographic form of the current token equal to *ice*?); `inter(gnd[-1], {n, f})` (does the set of values for *gender* category taken from the token preceding the current one intersect with constant set `{n, f}`?).
6. Constraints that test for morpho-syntactic agreement on a given set of grammatical categories between two specified tokens or a range of tokens.
7. Search operators that try to find a token satisfying a constraint in an iteration-like fashion using variables. The operators mimic regular predicates and can be used in conjunction with functional expressions on the token found.
8. Logical predicate operators (`and`, `not`, `or`) and conditional operator (`if`) that allow composition of simpler operators to create more complex ones.
9. Aforementioned `lex` operator that translates a set of strings according to a lexicon read from an external file.

An example operator utilising the language extensions is presented below.

```
if(
   rlook(0, end, $Pos, inter($s:Lemma,base[$Pos])),
   class[$Pos], // return the grammmatical class if found
   {ign} // else clause
)
```

The operator examines the sentence, starting from the central position (`0`), proceeding left-to-right (the iteration is realised by increasing the value of the `$Pos` variable). The first token whose possible lemma set intersects (`inter` predicate) with the string set given via an external variable (`$s:Lemma`) is taken, and its grammatical class returned. If no token satisfying the condition is found, the value of the *else* clause is taken — in this case, `ign` grammatical class (unknown form in the IPIC tagset).

4 Case Study: Creating a Memory-Based Chunker

Chunking is the task of identifying phrase boundaries in text. *NP chunking* is limited to recognising noun phrases (NPs). When using ML techniques, the usual practice is

to treat the task as a sequence labelling problem, i.e., special tags that denote whether a token is inside, outside or begins an NP are attached to tokens [14]. The task was performed for Polish using decision trees and hand-tailored features, yielding 85.7% precision and 84.9% recall [12].

Memory-Based Learning (MBL) is an ML technique where instead of the usual generalisation during training, all the training examples are memorised. The actual classification is based on finding a number of most similar examples in the memory and retrieving their class label. MBL has been successfully applied to various NLP tasks including chunking [3]. We can attempt to create a chunker for Polish using MBT, a software toolkit for memory-based tagging [4], to examine the impact of additional features generated by WCCL in a real NLP task.

The baseline was to train and test MBT on input consisting of the word forms, morpho-syntactic description (MSD) tags and the corresponding chunk tags (class labels). MBT treats the feature values atomically, hence tags are not decomposed into parts in this setting.

More sophisticated features were introduced by writing WCCL expressions. The first improvement was to provide features for grammatical class, number and gender in a fixed window $(-3, \ldots, 2)$ (similar features are used in TaKIPI [6]). As the NP chunks that we try to recognise are based on the agreement on number, gender and case, we introduced explicit tests for such agreements as the second improvement by providing the following WCCL predicates:

1. checking for agreement on two- and three-token ranges crossing the current position (5 predicates),
2. checking for "weak" agreement[1] on a token range that stretches between an adjective and a noun with those boundaries possibly being several tokens away from the current position (for both possible word orders).

The third step involved enriching the lexical information. In this setting we added 5 additional features accounting for the wordforms of all the tokens in the fixed window. To avoid inclusion of infrequent forms into the domain, we created a frequency list from an extra part of the employed corpus[2] and created a lexicon of 800 most frequent wordforms. The lexicon was then used by WCCL expressions that kept the encountered forms intact if present in the lexicon, while mapping the rest to empty symbol (e.g. `lex(lower(orth[0]), "freq")`).

The features for the third set-up were generated using the following code (and the `wccl-run` utility):

```
import("800.txt", "freq") // import lexicon file as "freq"
@"simple" (
   class[-3]; nmb[-3]; cas[-3]; gnd[-3]; // repeated for -2..2
)
```

[1] Weak agreement is not violated when the range contains additional indeclinable forms.

[2] The chunk corpus [12] consists of a small chunk-annotated part (used for chunker evaluation) and a much larger part with morpho-syntactic annotation only. The latter was used to gather the frequency list.

```
@"lex" (
   lex(lower(orth[-3]), "freq"); // repeated for -2..2
)
@"agrs" (
   agrpp(-1,0,{nmb,gnd,cas}); agrpp(0,1,{nmb,gnd,cas});
   wagr(-2,0,{nmb,gnd,cas}); wagr(-1,1,{nmb,gnd,cas});
   wagr(0,2,{nmb,gnd,cas});
   if(and(
      llook(0,-2,$L,agrpp($L, $L, {nmb,gnd,cas})),
      rlook(0,4,$R,inter(class[$R], {subst,ger,depr})),
      wagr($L,$R, {nmb,gnd,cas})
   ), $R);
   if(and(
      rlook(0,2,$R,agrpp($R, $R, {nmb,gnd,cas})),
      llook(0,-4,$L,inter(class[$L], {subst,ger,depr})),
      wagr($L,$R, {nmb,gnd,cas})
   ),$L)
)
```

The results on the data set from [12] are reported in Tab. 1. The most impressive improvement is observerd after introduction of simple symbols extracted from tags. Adding of agreement tests as well as lexical information also helps. Our best results are still significantly worse than those achieved by [12], nevertheless it is instructive to see that an acceptable performance may be quickly achieved by using readily available components, and that the extra features improved the result significantly.

Table 1. The perfomance of MBL chunkers with different feature sets

Chunker	Precision	Recall	F
MBL baseline: form, whole MSD tag	0.4782	0.6267	0.5424
MBL baseline + class, nmb, gnd, cas	0.6708	0.7655	0.7150
MBL baseline + class, nmb, gnd, cas, agreement	0.7133	0.8008	0.7544
MBL baseline + class, nmb, gnd, cas, agreement, lex	0.7238	0.8039	0.7617
Decision trees [12]	0.8574	0.8490	0.8530

5 Conclusion

Providing features required by ML approaches in NLP can be simplified accross many NLP tasks if a proper toolkit is made available. We analysed JOSKIPI formalism along with its implementations and found it to be inadequate as a general purpose toolkit for processing Polish corpora. Taking it as a great starting point, we addressed the identified problems and added important missing features, such as support for different tagsets, broader type system with variables of every type, ability to assign values to variables at run time, support for parallel processing, and more.

We proposed a successor to JOSKIPI: the WCCL formalism and its implementation, a shared library and useful command-line utilites. We aim to additionally develop a web API — this way the formalism will be used as a corpus query language.

References

1. Acedański, S., Gołuchowski, K.: A morphosyntactic rule-based brill tagger for polish. In: Proceedings of Intelligent Information Systems, pp. 67–76 (2009)
2. Broda, B., Piasecki, M.: SuperMatrix: a general tool for lexical semantic knowledge acquisition. In: Speech and Language Technology, vol. 11, pp. 239–254. Polish Phonetics Assocation (2008)
3. Daelemans, W., van den Bosch, A.: Memory-Based Language Processing. Cambridge University Press, Cambridge (2005)
4. Daelemans, W., Zavrel, J., van den Bosch, A., van der Sloot, K.: MBT: Memory-Based Tagger, version 3.2. Tech. Rep. 10-04, ILK (2010)
5. Leidner, J.: Current Issues in Software Engineering for Natural Language Processing. In: Patrick, J., Cunningham, H. (eds.) Proceedings of the HLT-NAACL 2003 Workshop (SEALTS), pp. 45–50 (2003)
6. Piasecki, M.: Polish tagger TaKIPI: Rule based construction and optimisation. Task Quarterly 11(1–2), 151–167 (2007)
7. Piasecki, M., Broda, B.: Semantic similarity measure of polish nouns based on linguistic features. In: Abramowicz, W. (ed.) BIS 2007. LNCS, vol. 4439, pp. 381–390. Springer, Heidelberg (2007)
8. Piasecki, M., Radziszewski, A.: Morphosyntactic constraints in acquisition of linguistic knowledge for polish. In: Marciniak, M., Mykowiecka, A. (eds.) Bolc Festschrift, vol. 5070, pp. 163–190. Springer, Heidelberg (2009)
9. Przepiórkowski, A.: Slavic Information Extraction and Partial Parsing. In: Proceedings of the Workshop on Balto-Slavonic Natural Language Processing, pp. 1–10. ACL, Prague (2007)
10. Przepiórkowski, A.: A comparison of two morphosyntactic tagsets of Polish. In: Koseska-Toszewa, V., Dimitrova, L., Roszko, R. (eds.) Representing Semantics in Digital Lexicography: Proceedings of MONDILEX Fourth Open Workshop, Warszawa, pp. 138–144 (2009)
11. Przepiórkowski, A., Woliński, M.: A flexemic tagset for Polish. In: Proceedings of Morphological Processing of Slavic Languages, EACL 2003 (2003)
12. Radziszewski, A., Piasecki, M.: A preliminary noun phrase chunker for Polish. In: Proceedings of the Intelligent Information Systems (2010)
13. Radziszewski, A., Śniatowski, T.: Maca — a configurable tool to integrate Polish morphological data. In: Proceedings of FreeRBMT11 (2011)
14. Ramshaw, L.A., Marcus, M.P.: Text chunking using transformation-based learning. In: Proceedings of the Third ACL Workshop on Very Large Corpora, Cambridge, MA, USA, pp. 82–94 (1995)