PSI-Toolkit: A Natural Language Processing Pipeline

Filip Graliński, Krzysztof Jassem, Marcin Junczys-Dowmunt

Abstract The paper presents the main ideas and the architecture of the open source PSI-Toolkit, a set of linguistic tools being developed within a project financed by the Polish Ministry of Science and Higher Education. The toolkit is intended for experienced language engineers as well as casual users not having any technological background. The former group of users is delivered a set of libraries that may be included in their Perl, Python or Java applications. The needs of the latter group should be satisfied by a user friendly web interface. The main feature of the toolkit is its data structure, the so-called PSI-lattice that assembles annotations delivered by all PSI tools. This cohesive architecture allows the user to invoke a series of processes with one command. The command has the form of a pipeline of instructions resembling shell command pipelines known from Linux-based systems.

1 Introduction

1.1 PSI-Toolkit – General Outline

PSI-Toolkit consists of a set of tools (called processors) that aim at processing natural language texts. There are three types of processors: readers, annotators, and writers. A reader creates the main data structure, the so-called PSI-lattice (see Section 2.), from an external source of information, e.g. from a file (a text file, HTML file, PDF file) or keyboard. An annotator (e.g. a tokenizer, a lemmatizer, a parser) adds new annotations in the form of new edges to the PSI-lattice. Finally, a writer writes back a PSI-lattice to an output device (e.g a file or screen).

Faculty of Mathematics and Computer Science
Adam Mickiewicz University
ul. Umultowska 87, 61-614 Poznań, Poland
e-mail: {filipg, jassem, junczys}@amu.edu.pl

1.2 An Overview of NLP Toolkits

The first widely used NLP toolkit was GATE (General Architecture for Text Engineering) [1] designed in the mid-nineties and still being developed. Currently, dozens of open-source toolkits are available; Wikipedia lists over 30 of them.[1] Most toolkits deal with the English language, but toolkits designed for other languages (like TESLA – German [2], Nooj – French [3], Apertium – Spanish [4]) gain popularity. The situation is a bit different for Slavonic languages. In spite of the rapid progress in the development of linguistic tools and resources for Slavonic languages in the 21st century, attempts at organizing them in the format of cohesive toolkits are still sparse and rare.

1.3 PSI-Toolkit Assumptions

1.3.1 Free Licence

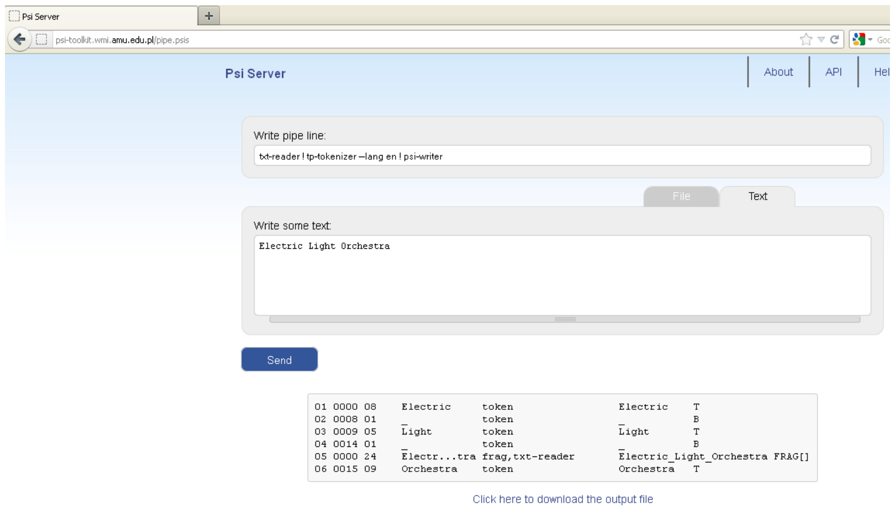
PSI-Toolkit is released under Lesser General Public License (LGPL). This is a free license, which, in addition to GPL, allows for linking with proprietary software and further distributing the result under any terms (also for business purposes).

1.3.2 Easy Access via a Web Browser

All PSI-Toolkit tools may be accessed and tested on-line via a web browser interface at `psi-toolkit.wmi.amu.edu.pl`. Figure 1. shows an exemplary usage of the toolkit in a web window. The user inputs a text into an edit box (e.g. *Electric Light Orchestra*) and specifies a command as a sequence of processors that should be executed on the text (e.g. `txt-reader ! tp-tokenizer --lang en ! psi-writer`). The processors are run in the order specified in the command (here: read a raw text, tokenize the text according to rules of the English language, write the output in the dedicated PSI format). The PSI output lists every edge of the PSI-lattice (see: 2.1) in a separate line. In Figure 1: line 01: corresponds to the edge spanning over the first 8 characters of the input (start position is equal to 0000, offset is equal to 0008). The type of the edge is "token", the value of the token is *Electric*, the lemma the token belongs to is *Electric*. Line 02 describes a space between the first and the second token of the input. Line 05 corresponds to the edge spanning over the whole input (the edge has been constructed by the `txt-reader`).

The format of the output may be simplified by replacing the `psi-writer` with `simple-writer` in the command line. Combining the simple output format with

[1] `http://en.wikipedia.org/wiki/List_of_natural_language_processing_toolkits`.



txt-reader ! tp-tokenizer --lang en ! psi-writer

Electric Light Orchestra

```
01 0000 08    Electric     token                  Electric    T
02 0008 01    _            token                  _           B
03 0009 05    Light        token                  Light       T
04 0014 01    _            token                  _           B
05 0000 24    Electr...tra frag,txt-reader        Electric_Light_Orchestra FRAG[]
06 0015 09    Orchestra    token                  Orchestra   T
```

Fig. 1: Web access to PSI-Toolkit.

other PSI-Toolkit processors may result, for instance, in on-line machine translation of the input, yielding a use case similar to that offered by Apertium.

1.3.3 Linux Distribution

The PSI-Toolkit is also distributed in the form of two Linux binaries: `psi-pipe` and `psi-server`. `psi-pipe` may be installed and used as a command-line tool on personal computers, whereas `psi-server` allows for the creation of other PSI-Toolkit web pages. Currently, the PSI-Toolkit binaries are distributed in form of easily installable packages for the Ubuntu Linux distribution.

1.3.4 PSI-Toolkit Pipeline

The PSI-Toolkit command is specified as a pipeline of processors (an example of such a pipeline was given in 1.3.2). If the PSI-Toolkit is used under Ubuntu on a personal computer, the processors should be invoked in a bash-like manner. For example, in order to process the string *Electric Light Orchestra* in a way equivalent to that shown in 1.3.2., the following pipeline should be formed:

```
> echo "Electric Light Orchestra" | psi_pipe \
txt_reader ! tp-tokenizer --lang en ! psi-writer \
| grep "T_" | less -S
```

Note the resemblance of the original linux piping syntax by providing processors chained with `!` as arguments to the `psi-pipe` tool. Moreover, PSI processors may be replaced or supplemented by external tools in the PSI pipeline. The PSI engine will add annotations provided by external tools to the PSI-lattice. It is admissible to use two different processors of the same type in the same pipeline. For example running two different sentence splitters in the same process may result in two different sentence splits. The ambiguity is stored in the PSI-lattice (it may or may not be resolved in further processing).

1.3.5 Java, Perl, Python Libraries

PSI tools are also accessible as libraries of selected programming languages. The user may include the PSI tools in any of the Java, Perl or Python source codes.

1.3.6 Switches

Usage of a PSI-Toolkit processor may be customized by means of switches (options). Figure 2. shows the list of available switches for the `perl-simple-writer` processor, the "Perl-wrapped" version of the PSI `simple-writer`.

```
Allowed options:
 --linear            skips cross-edges
 --no-alts           skips alternative edges
 --with-blank        does not skip edges with whitespace text
 --tag arg (=token)  basic tag
 --spec arg          specification of higher-order tags
 --with-args         if set, returns text with annotation as
a hash element
```

Fig. 2: Switches of the Perl-simple-writer processor

1.3.7 Natural Languages Processed by PSI-Toolkit

There are no restrictions on languages analyzed by the PSI processors (UTF-8 is used for internationalization). One PSI pipeline may consist of processors defined for various languages. However, the tools delivered by the authors are oriented mainly towards Polish (some processors are also defined for English). The authors hope that the PSI-Toolkit can bring together most Polish language processing tools in one framework. Currently these tools are dispersed (see `http://clip.ipipan.waw.pl` for an exhaustive list of NLP tools and resources for Polish). The proof of this concept has been implemented on the morphosyntactic level: the external morphological tagger Morfologik[2] has been incorporated into the PSI-Toolkit and can be run in the PSI pipeline instead of or besides (!) the standard morphological analyzer of the PSI-Toolkit.

2 PSI-lattice

2.1 Definition

All PSI-Toolkit processors operate on a common lattice-based data structure called *PSI-lattice*. The term *lattice* refers to a (word) lattice [5] as used in natural language processing rather than to a more general notion of abstract algebra. A PSI lattice is composed of an input (*substrate*) string and *edges* spanning substrings of the substrate string.

Lattice readers read the substrate string (usually from a file) and add some initial edges – usually edges spanning single *symbols* (a symbol is a character occurring as part of a natural-language text) and edges representing mark-up tags of a given format (e.g. the PSI-Toolkit HTML reader would construct edges encapsulating HTML tags). PSI-Toolkit annotators create new PSI-lattice edges based on existing ones and/or the substrate string, e.g. a tokenizer groups symbol edges into token edges, a lemmatizer creates edges representing lemmas and lexemes for each token edge, a parser produces new parse edges based on the lexeme edges and previously added parse edges. Finally, lattice writers do not add any new edges, they just output all or selected PSI-lattice edges in a required format.

A PSI-lattice edge consists of the following elements:

- source and target *vertices* (PSI-lattice vertices are defined as inter-character points),
- *annotation item*,
- *layer tags*,
- *partitions*,
- *score* (weight).

The annotation item conveys the description of the language unit represented by a given edge. An annotation item is realized as an attribute-value matrix in which two attributes are obligatory: *category* and *text*. The meaning and interpretation of these two attributes varies between the PSI-Toolkit processors, e.g. the category of a token edge is its type (blank, word, punctuation mark etc.) and its text attribute is just the token itself (as a string), whereas for a lexeme edge the part of speech of the given lexeme is used as the category attribute and its identifier (e.g. `long+adj` for the adjective *long*) – as the text attribute. Other attributes are used to describe

[2] Available at `http://morfologik.blogspot.com`. To our knowledge, this tool has not been described in a published scientific work.

particular features of the given language unit, e.g. morphosyntactic features such as case, gender, tense, person etc.

Layer tags are used to express some meta-information associated with a given edge, e.g.:

- edge type – whether an edge represents a token, lemma, lexeme, parse etc.,
- the name of the processor that added the edge,
- tagset used in the annotation item.

The important point is that edges with the same annotation item are collapsed into a single edge, even if they have different layer tags (with the exception of plane tags – more on this later). In other words, if a processor produces an edge with the same annotation item and the same source and target vertices as some edge already present in the PSI-lattice, then no new edge is added to the PSI-lattice, the two edges are merged instead, the sum of their layer tags is assigned to the updated edge.

As it is not always reasonable to collapse two edges with the same annotation item (for instance in the context of machine translation it would not make sense to equate a source-language lexeme and its target-language equivalent when they happen to have the same canonical form and the same attributes), so-called *plane (layer) tags* are introduced. Plane tags divide a PSI-lattice into a set of disjoint *planes*, i.e. edges belonging to different PSI-lattice planes (having different plane tags) will not be collapsed even if they share the same annotation item. By convention, plane tags begin with an exclamation mark. Language-code tags specifying the language of a given language unit (e.g. `!pl`, `!en`, `!de` tags) are typical examples of plane tags. When a processor combines some edges into a new edge, the new edge will inherit the plane tags of the subedges by default, unless a list of plane edges was specified explicitly while creating a new edge.

A *partition* specifies which edges were used to create a given edge. For example for a parse edge the partition is a sequence of lexemes (terminals) and subparses (non-terminals) directly combined into the given edge. An edge may have more than one partition, e.g. an edge spanning the expression *Electric Light Orchestra* may be a parse partitioned into *Electric* + *Light Orchestra* or into *Electric Light* + *Orchestra*, or it could be a lexeme produced by a multi-word unit lexicon (partitioned into tokens in this interpretation). Each partition is assigned the following properties: layer tags, score and (optionally) rule identifier. *Rule identifier* is an arbitrary number the interpretation of which varies between the PSI-Toolkit processors, e.g. for a parser it could be an identifier of a grammar rule (a partition could be linked to a rule of the parser's grammar this way).

The score (of an edge or a partition) is a floating point value for which the following properties hold:

- the score of an edge is the maximum score of its partitions,
- the score of a partition is the sum of scores of its subedges plus some score for the rule that generated the partition.

For instance, a score for a parse edge/partition could be interpreted as the log probability of the parse and the score of a parse partition could be calculated as the sum of log probabilities of its subedges and the log probability of the grammar rule applied. PSI-lattice scores, however, does not have to be interpreted as (log) probabilities. They might be treated as some kind of weights or penalties (the latter for negative scores).

Taking the maximum score of the partitions (rather than the sum of the partition scores) as the score of an edge might seem controversial from a formal point of view. The reason why we decided to use the maximum value is that the partitions do not have to be independent. For example we would like to run two different lemmatizers on the same words.

2.2 Motivation and Design Assumptions

In other toolkits and even more so in the case of independent stand-alone tools the internal data representation differs in general from the input or output data. In consequence, the internal representation is also different between two kinds of tools, e.g. a tokenizer works mostly on raw string data, while a parser works on trees, forests or possibly charts, a statistical machine translation like Moses uses implicit hypothesis graphs etc.

During the conversion of the internal representation to a readable output format usually a lot of information is purposefully discarded. This is the case if only the first-best interpretation for an ambiguous problem is provided, for instance the most probable tokenization or the best-scored translation. Alternative interpretations can often only be retrieved by analysing log data, the format of which is most certainly not standardized between different tools or toolkits. Providing alternative interpretations to a follow-up tool involves then a lot of manual intervention.

The PSI-lattice is designed to tackle these problems. Firstly, the PSI-lattice is supposed to provide a one-size-fits-all data structure that can be used to contain any annotation generated by various language processing tools without losing information provided by previous processing steps. Secondly, a standardized way to pass around alternate interpretations besides the single-best results is provided. That way, one can easily take advantage of delayed disambiguation, i.e. a higher-order tool can choose among alternatives that a lower-order tool was not able to fully disambiguate. Examples might be a parser that chooses between two given part-of-speech tags or a syntax-based machine translation application that translates parse-forests rather than parse trees. Finally the PSI-lattice obsoletes the conversion from an internal data representation to an external output format as it can itself serve for internal data representation, being perfectly capable of representing alternative tokenizations, parse forests or translation ambiguities. This will be illustrated in the next section where a shallow parser directly constructs a PSI-lattice as output data while using it as an intermediate representation of parses obtained so far.

The ease of data interchange between atomic application leads to the second important design guideline of the presented toolkit: Simplified combination of tools. Using available toolkits can be a challenge by itself as their special syntax or construction can result in quite a steep learning curve. For instance, the NLTK requires at least a basic knowledge of the `Python` programming language. For the PSI-Toolkit it has been decided to walk down a path that has already been cut through — in a context much broader than natural language processing.

As far as basic text-processing capabilities are concerned, the popular shells in Linux or other Unix-based operation systems can themselves be considered as capable text-processing toolkits. The rationale behind them is to provide a set of small applications or commands where each tool by itself is limited in functionality in the sense that it can perform only exactly one task. However, each tool is supposed to perform particularly well for the task it has been designed for. Typically, data for a tool is provided on its standard input while the results can be read from its standard output. The power of the command-line comes then from the possibility to combine these tools into pipelines by directing standard output from one tool to the standard input of another tool, thus building up a much more sophisticated application from smaller parts. For instance, one can easily create basic sorted frequency lists of words in a text file combining `sed`, `sort`, `uniq` into a pipeline without the need to write specialized scripts in a more complex programming language like `Perl`. The operations of each tool in a pipeline can be adjusted by switches — for instance the reverse order of sorting for the `sort` command by `sort -r`.

Due to the popularity of Linux-based systems in the natural language processing community it is safe to assume a high degree of familiarity of the average NLP-researcher with one of the available Linux shells. This assumption leads directly to the second important design decision for our toolkit. Concerning usability it lends itself to exploit this familiarity by simulating the look and feel of the command-line shells in Linux. This includes the construction of a growing set of self-contained single-purpose applications that can be chained into complex pipelines by the use of a familiar looking piping syntax. Also, additional options to any such building-block are provided by switches that again mimic their Linux-shell counterparts.

Every pipeline constructed in such a way constitutes a new stand-alone tool, with well-defined input data, output data and functionality. Formally speaking, a pipeline can be seen as self-contained object similar to a functor, the elements of the pipeline being in turn comparable to combinators that make up the functor. While in Linux shells data is typically passed around in text or plain binary format, the exchange format between the PSI-Toolkit components is the described PSI-lattice.

3 Language Processing with PSI-lattice

3.1 Segmentation

For both, tokenization and sentence splitting, the rules created for the Translatica machine translation system[3] were used (the rules had been published under

[3] http://translatica.pl

the GNU Lesser General Public Licence). The tokenizer (the processor is called `tp-tokenizer`) uses a Translatica in-house format for "cutting-off" rules (each rule specifies a regular expression describing a token of a given type, only the first matching rule is applied), whereas the sentence splitter (called `srx-segmenter`) uses the SRX (Segmentation Rules eXchange) standard[4].

It is quite straightforward to store alternative segmentations (on both token and sentence level) in a PSI-lattice and to take them into account in the subsequent processing stages (in lemmatization, parsing etc.). For the time being, both `tp-tokenizer` and `srx-segmenter` produce only one segmentation, as it is not possible to express segmentation non-determinism in either the tokenization or SRX rules.

Since there are hard-to-disambiguate cases for token/sentence segmentation — e.g. in Polish *gen.* is either an abbreviation for *generał* (= *general*, a military rank) or the word *gen* (= *gene*) at the end of a sentence — i.e. cases in which the decision must be postponed to a later processing stage[5], we are considering enhancing the segmentation rules with some non-determinism. In the case of sentence breaking, however, it would involve extending the widely-used SRX standard. For the time being, segmentation ambiguity could be achieved (at least to some extent) with running segmentation processors twice with slightly different set of rules (e.g. once with *gen.* listed as an abbreviation, once – not listed).

Both `tp-tokenizer` and `srx-segmenter` can be run with an option specifying the maximum length of, respectively, a token and a sentence. In fact, there exist two types of length limits: a soft one and a hard one – in case of the soft limit a token/sentence break is forced only on spaces, whereas exceeding the hard limit always triggers segmentation. Such limits were introduced for practical reasons as a safeguard against extremely long tokens/sentences which may occur when "garbage" (e.g. unrecognized binary data) is fed to PSI-Toolkit (very long tokens or sentences might slow down the subsequent processing to an unacceptable degree).

So far, PSI-Toolkit handles tokenization and sentence breaking for English, French, German, Italian, Polish, Russian and Spanish.

3.2 Lemmatization and Lexica

We plan to incorporate as many open source lemmatizers and lexica as possible into PSI-Toolkit. So far, we have created a general framework for adding new lemmatizers into PSI-Toolkit (now it is relatively easy to add a new lemmatizer on condition that a simple function returning all the morphological interpretations is provided by a given lemmatizer) and incorporated the aforementioned Morfologik lemmatizer for Polish. We are in the process of developing our own finite state library (which

[4] http://www.gala-global.org/oscarStandards/srx/srx20.html

[5] E.g. in named entity recognition or in parsing, see [6] for discussion of tokenization ambiguity

will be used not only for lemmatization, but also for syntax-based machine translation) and adding support for the Stuttgart Finite State Transducer Tools (SFST) [7].

3.3 Shallow Parsing

The PSI-Toolkit includes a shallow parser – Puddle – that can be used to work with any language as long as a appropriate grammar is provided. It started out as a C++ adaptation of the Spejd [8] shallow parser which was a pure Java tool a that time. By now, Puddle has evolved into an independent tool that has been used as a parser for French, Spanish, and Italian in the syntax-based statistical machine translation application Bonsai [9].

The latest version of Puddle has been redesigned to work with the PSI-lattice as an input and output data structure. The parse tree itself is also constructed directly on top of the input lattice. Consecutive iterations work on a PSI-lattice that has been extended by exactly one edge in the previous iteration.

The shallow parsing process of Puddle relies on a set of string matching rules constructed as regular expressions over single characters, words, part-of-speech tags, lemmas and grammatical categories etc. Apart from the matching portion of a rule, it is also possible to define matching patterns for left and right contexts of the main match. The parse tree construction process is linear, matching rules are applied iteratively in a deterministic fashion. The first possible match is chosen and a spanning edge is added to the lattice. No actual search is performed which puts a lot of weight onto the careful design of the parsing grammar. The order of the rules in a grammar determines the choice of rules to be applied to a sentence. The parsing process is finished if no rules can be applied during an iteration.

For the parser to work properly on different types of information, the PSI-lattice already needs to contain edges for tokens, lemmas and morphological properties, so it has to be the result of a pipeline that generated this kind of data. The lattice does not need to be previously disambiguated (although that can be helpful and implemented by adding one or more POS-Taggers to the pipeline) since the parser can also work as a disambiguation tool. Matching rules that require morphological agreement between matched symbols (e.g. between a noun and an adjective) can mark edges that contradict this agreement requirement as “discarded”. However, discarded edges are never quite deleted from the PSI-lattice since they can be of importance in later higher-order processing.

The parser adds a special type of edge to the lattice marked with “parse” tag. The partition of the edge contains information which subedges have been used to construct the new edge. If there are several possible interpretations all of them are added to the lattice. Syntactic heads are marked with additional tags in their respective edges.

4 Work-flow: An End-to-end Example

In this section we will illustrate an example pipeline for a short Polish phrase *przykładowa analiza* (eng. *example analysis*). Figure 3 describes the stepwise construction of the PSI-lattice for this phrase. The following commands have been used to create the corresponding lattices:

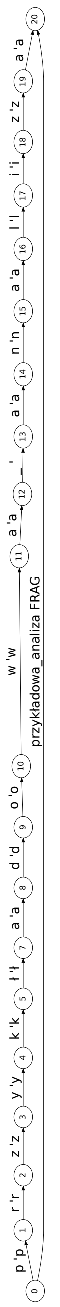
a) `txt-reader ! dot-writer`
b) `txt-reader ! tp-tokenizer --lang pl ! dot-writer`
c) `txt-reader ! tp-tokenizer --lang pl ! morfologik ! dot-writer`
d) `txt-reader ! tp-tokenizer --lang pl ! morfologik ! puddle --lang pl ! dot-writer`

The common “txt-reader” is used to convert a text string into an unannotated lattice. The characters of that string are the smallest units in the PSI-lattice (figure 3a). All figures have themselves been generated by another PSI-Toolkit processor, `dot-writer`, that converts PSI-lattices to graphs described in the DOT-language. This can be interpreted for instance by tools from the GraphViz library [10] available at `http://graphviz.org`. The toolkit contains also a writer named `gv-writer` that uses the GraphViz library to generated `pdf` or `svg` files directly.

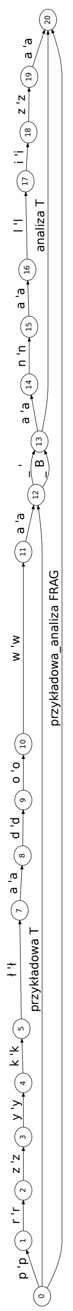
A Polish language tokenizer (figure 3b) is inserted between the reader and writer processors and adds the first level of annotation to the PSI-lattice. Edges that span characters mark tokens (T) and blanks (B), other symbol types could include punctuation information or HTML mark-up data. Alternative tokenization results could be included in the PSI-lattice as well, this would in most cases results in crossing edges.

The third level in the PSI-lattice (figure 3c) is the result of the application of the Morfologik morphological analyzer. Different morphological interpretations are added as individual edges (the morphological features have been omitted in the illustration due to space requirements). The first token *przykładowa* has several interpretations as an adjective (`adj`), similarly *analiza* has been assigned two interpretations as a noun `subst`.

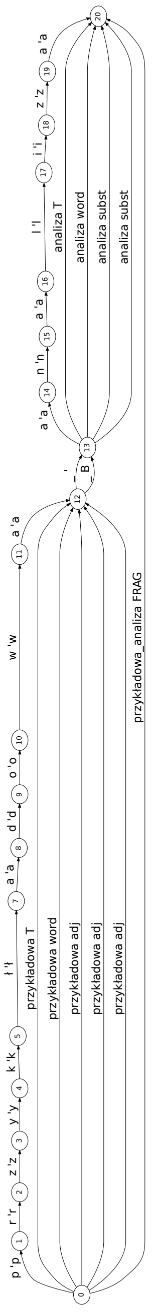
Finally the last level of annotation (figure 3d) is added by the shallow parser described in the previous section. The ambiguity of the morphological annotation results in an ambiguous parsing result with two parallel adjective phrase edges (`AP`). Together with the following noun this adjective phrases forms a noun phrase (`NP`) that spans the entire input string. To facilitate the analysis for the shallow parser a part-of-speech tagger can be added between the morphological analyzer and the parser. Currently, a simple maximum entropy-based tagger processor is being added to the toolkit. For the moment, the shallow parser represents the highest-order processor in the PSI-Toolkit, but deep parsers, machine-translation processors and many other tools will be available in the near future, as well as converters for existing tools like the mentioned Morfologik processor.



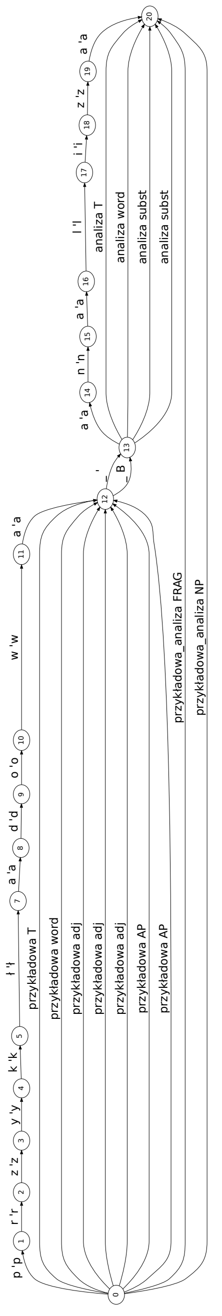
(a) Plain text



(b) Tokenized text



(c) Morphological annotation



(d) Shallow parsing

Fig. 3: Web access to PSI-Toolkit.

5 Conclusions and Future Work

The paper presents the main ideas and the architecture of the open source PSI-Toolkit. The linguistic annotations are stored in the form of the lattice, which allows for keeping annotations of various types in one structure. The PSI-lattice allows for the delayed disambiguation and the usage of multiple processors of the same type (e.g. a few different sentence splitters) in the same run. The PSI tools may be accessed via the web browser as well as run locally. A processing command should be formed as a pipeline similar to the command pipelines known for Linux shells. The PSI wrappers make it possible to use the PSI tools in Java, Perl or Python applications. The project is under rapid development, ending in April 2013. The list of the processors available in PSI-Toolkit is being systematically increased. At the moment this paper is prepared the list consists of 17 processors. The current list of processors, together with their switches and examples of usage is accessible at: `http://psi-toolkit.wmi.amu.edu.pl/help.html`.

Acknowledgements

This paper is based on research funded by the Polish Ministry of Science and Higher Education (Grant No. N N516 480540).

References

1. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V., Aswani, N., Roberts, I., Gorrell, G., Funk, A., Roberts, A., Damljanovic, D., Heitz, T., Greenwood, M.A., Saggion, H., Petrak, J., Li, Y., Peters, W.: Text Processing with GATE (Version 6). (2011)
2. Hermes, J., Schwiebert, S.: Classification of text processing components: The tesla role system. In: GfKl. (2008) 285–294
3. Silberztein, M.: Nooj: a linguistic annotation system for corpus processing. In: Proceedings of HLT/EMNLP on Interactive Demonstrations. HLT-Demo '05, Stroudsburg, PA, USA, Association for Computational Linguistics (2005) 10–11
4. Carme Armentano-Oller, C., Corbí-Bellot, A.M., Forcada, M.L., Ginestí-Rosell, M., Bonev, B., Ortiz-Rojas, S., Pérez-Ortiz, J.A., Ramírez-Sánchez, G., Sánchez-Martínez, F.: An open-source shallow-transfer machine translation toolbox: consequences of its release and availability. In: OSMaTran: Open-Source Machine Translation, A workshop at Machine Translation Summit X. (September 2005) 23–30
5. Dyer, C., Muresan, S., Resnik, P.: Generalizing word lattice translation. In McKeown, K., Moore, J.D., Teufel, S., Allan, J., Furui, S., eds.: ACL, The Association for Computer Linguistics (2008) 1012–1020
6. Chanod, J.P., Tapanainen, P.: A non-deterministic tokeniser for finite-state parsing. In: ECAI '96 workshop on "Extended finite state models of language". (11-12 August 1996)
7. Schmid, H.: A programming language for finite state transducers. In: Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing (FSMNLP 2005), Helsinki, Finland (2005)

8. Przepiórkowski, A., Buczyński, A.: ♠: Shallow parsing and disambiguation engine. In: Proceedings of the 3rd Language & Technology Conference, Poznań (2007)
9. Junczys-Dowmunt, M.: It's all about the trees — towards a hybrid syntax-based MT system. In: Proceedings of IMCSIT. (2009) 219–226
10. Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C., Woodhull, G.: Graphviz and dynagraph static and dynamic graph drawing tools. In: GRAPH DRAWING SOFTWARE, Springer-Verlag (2003) 127–148