Universal Computational Formalisms and Developer Environment for Rule-Based NLP

Svetlana Sheremetyeva(✉)

Department of Linguistics and Translation, South Ural State University,
76, Lenin pr., Chelyabinsk 454080, Russia
sheremetevaso@susu.ru, lanaconsult@mail.dk

Abstract. The paper explores the issues of universal computational formalisms and reusable developer environment as applied to rule-based NLP. It suggests a portable grammar framework and modular NLP architecture that by combining certain modules can be reused for different unilingual and multilingual applications through a universal developer environment. The developer environment includes a lexicon shell with flexible settings to define, among others, tag descriptions, entry structures, depth of knowledge, and a number of compilers with universal rule-writing formalisms. The formalisms and compilers described have been successfully used (in different combinations and with different depth of analysis) in a number of unilingual and multilingual applications that involved English, Danish, French and Russian.

Keywords: Rule-based NLP · Reusability · Computational formalism
Compilers

1 Introduction

After more than a decade of the dominance of the statistical paradigm in NLP, a new wave of R&D has reverted to the primacy of rule-based approaches. This is particularly true for processing tasks where highly inflecting languages are involved, for which, though certain attempts are made to substitute costly rule-based procedures with purely statistical methods, hidden costs are recognized associated with the use of pure statistics [11]. However, high quality NLP demands rich knowledge resources (world models, grammar rules and lexicons), which are often handcrafted from scratch for every new application, language or language pair.

The idea to reduce development and maintenance costs, by sharing and reusing processing methods and knowledge has been in focus of researchers' attention for many years. Certain attempts have been made to develop universal tagsets [2, 7], portable cross-linguistic knowledge [6, 16], and reusable rule-based components [9]. A formalism for simultaneous rule-based morphosyntactic tagging and partial parsing is suggested in [10]. Universal computational formalisms have been already explored in early works on portability [3, 8], which outline the following major principles: (a) universal computational formalisms are to be based on grammars that in a uniform

A. Gelbukh (Ed.): CICLing 2017, LNCS 10761, pp. 67–78, 2018.
https://doi.org/10.1007/978-3-319-77113-7_5

way deal with atomic informational structures and then manipulate these structures by means of a few well-defined operations, which build new more complex structures; (b) within the frame of these formalisms both the atomic and complex structures, are to be application oriented and motivated by processing considerations; (c) a computational formalism is to be well-defined, which means that its semantics (not the semantics of the language described by the formalism) is also well defined.

To be used in practice, computational formalisms, apart from being natural language frameworks, should be available for a developer through developer tools for knowledge acquisition, code validation, navigation, test suite management, etc. Most of such tools (see, e.g., [1]) require programming qualification and are primarily developed for programmers. However, the contribution of linguistic knowledge in developing rule-based applications cannot be cannot be but appreciated, though the needs of linguists, not so experienced in programming, are often neglected.

In this work we attempt to cover this gap and suggest universal computational formalisms and developer environment addressing both, programmers, and, primarily, linguists without extensive programming training. In what follows, we first present the overall framework of a line of rule-based applications, sharing linguistic and programming resources and then describe the main modules of the developer environment. The work in summarized in Conclusions.

2 Overall Framework

2.1 Grammar

In an attempt to introduce robustness into the grammar itself, rather than adjusting a parser algorithm, we combine the formalisms of context free lexicalized Phrase Structure Grammar (PSG) and Dependency Grammar (DG). The PSG component consists of a subset of regular PSG rewriting rules. However, this subset includes neither the basic PSG rule "S = NP + VP", nor any rules for rewriting VP.

The PSG grammar component covers only those sentence constituents that are not clause predicates (be it a main clause or a subordinate/relative clause). It is the basis for a chunking procedure and does not give any description of syntactic dependencies. The PSG component is specified over *a space of supertags* [14] augmented with local information, such as lexical preference and some of rhetorical knowledge, - the knowledge about text segments, anchored to tabulations, commas and periods.

The DG grammar component is a strongly lexicalized case-role grammar specified *over the space of phrases (NP, PP, etc.)* and a residue of tagged "ungrammatical" words, i.e., words that do not satisfy any of the rules of the PSG component. All syntactic and semantic knowledge within this grammar is anchored to one type of lexemes, namely *predicates*.

The grammar assigns a final parse (a universal content representation) to a sentence as shown in Fig. 1, where *label* is a unique identifier of the elementary predicate-argument structure (by convention, marked by the number of its predicate as it appears in the sentence, *predicate-class* is a label of an ontological concept, *predicate* is a string corresponding to a predicate from the lexicon, *status* is a semantic status of a case-role,

such as place, instrument, etc., and *value* is a string which fills a case-role. *Supertag* is a tag, which conveys morphological, syntactic and semantic features as specified in the lexicon. *Word* and *phrase* are a word and phrase (NPs, PPs, etc.) in a standard understanding. This representation is universal in that, in case of multilingual applications, e.g. machine translation, the format of predicate-argument structures stays invariant after transfer to a TL.

```
text::={ template){template}*
template::={label predicate-class predicate ((case-role)(case-role)*}
case-role::= (status value)
value::= phrase{(phrase(word supertag)*)}*
```

Fig. 1. A universal format of content representation, invariant between languages.

2.2 Processing Steps and Applications Architecture

We here present a number of rule-based modules that in different configurations can be used to solve different unilingual and multilingual NLP tasks. This architecture resulted from our multi-year research, in the course of which formalisms and programs first created for authoring of patent claims in English were further ported and updated to develop a family of other multilingual scientific- and technical information-related applications, see, e.g., [5, 14, 15], to name just a few.

The reuse of earlier developed modules was to a great extent possible due to the universal grammar formalism (see Sect. 2.1) and elaborate developer environment for rule and lexicon acquisition.

An umbrella configuration of our processing modules shown in Fig. 2 covers the traditional top level procedures of RBMT (analysis, transfer and generation), while in particular applications only selected modules can be used (e.g., in case of unilingual authoring or summarization the Transfer module is skipped, and the Analyzer is pipelined directly into the Generator). All modules are compatible and can provide different depth of processing, the grammar formalism being the same. Every top level procedure includes a number of application-specific sub procedures. In our computational formalism the basic analysis scenario consists of the following sequence of procedures: *Tokenization, Tagging, Chunking and Shallow semantic analysis. Tokenization* can be tuned to detect generally used and more specific features and to flag them with different types of "border" tags, thus significantly augmenting the feature space for disambiguation rules. *Tagging* includes assigning tags by lexicon look up and tag disambiguation according to disambiguation rules. The specificity of the tagging procedure is that it does not require any lemmatization due to the amount of knowledge stored in the lexicon, where for all lexemes their paradigms are explicitly listed [13]. *Chucking* is performed by a bottom-up heuristic parser with a recursive pattern matching technique. It identifies and classifies text constituents as typed phrases. *Shallow semantic analysis* determines *semantic dependency relations* between the classified text chunks and predicates.

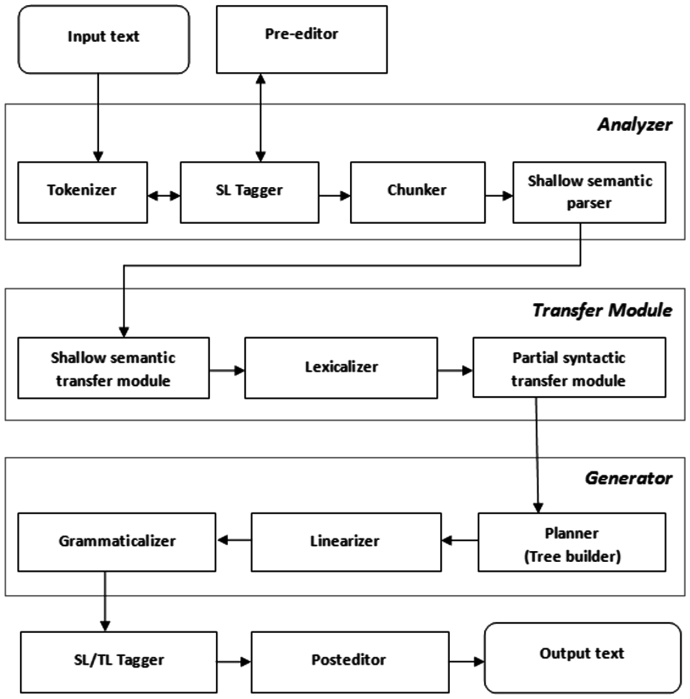

Fig. 2. An umbrella configuration of the rule-based NLP architecture.

For every identified phrase-chunk its governing predicate is detected and, then, the case-role status for every chunked phrase is determined. The final parse is a shallow semantic representation in the form of predicate/argument structures filled with SL text strings (see Fig. 1). Transfer is applied in case of multilingual applications and is a combination of interlingual transfer, lexicalization and syntactic transfer. The interlingual transfer substitutes a SL predicate-argument structure with a TL predicate-argument structure and links the latter with the TL knowledge in the lexicon. Lexicalization, called *Base transfer* substitutes every SL word in a predicate-argument structure with the base form of its cross-language equivalent by lexicon look-up. The predicate-argument format is kept unchanged (invariant). The *Syntactic transfer* is responsible for substituting lexicalized TL fillers resulting from the lexicon look up with well formed TL chunks, which is in fact translation of SL chunks-case-role fillers into a target language. *Generation* module linearizes predicate-argument structures filled with case-role strings into well-formed sentences. In case of a multilingual application a "real" translation procedure is thus reduced to the phrase level which, though not without problems, is still much simpler than machine translation that involves a full syntactic analysis and generation of possibly very complex sentences.

Every procedure relies on the system static knowledge and on the dynamic knowledge collected by the previous processing procedures. The knowledge handling and reuse is maintained by the universal developer environment.

3 Developer Environment

3.1 General Characteristic

The developer environment, we describe here, can be used to acquire and handle knowledge for multiple languages, domains. It includes a lexicon shell with flexible settings, several rule-acquisition compilers, where a linguist can write rules in a very simple formal language, and rule-control interfaces. The compilers provide for a computer environment, which can be used by a linguist not very experienced in programming. The multilingual lexicon developer tool with flexible settings is described in detail in [13]. The program shell of the lexicon allows defining entry structures, tagsets, contains knowledge that is directly used for text generation.

The lexicon program permits porting entry structures, tags and knowledge between languages and applications. The lexicon knowledge is directly pipelined to the rule acquisition compilers. Any changes made in the lexicon, e.g., tagsets, instantaneously propagate to the compilers and are displayed in the compiler interfaces. Developer environment is multilingual and every compiler is linked to a lexicon in a corresponding language.

Rules for different languages are accessed from a single program startup window. It makes it possible for the developer to freely navigate between knowledge bases for different applications and languages and to easily reuse appropriate amounts of linguistic knowledge, which proves to be quite possible, especially, in highly restricted domains, like, e.g., patent claims [14].

All compilers are equipped with front-end interfaces with a lot of effort saving functionalities, - it is possible on a mouse click to automatically get rule templates, transfer tag notations from the displayed list of tags to the curser position and check the consistency of tags and rule syntax. The compilers thus provide for an easy way to experiment with knowledge by simply copying the rules from one compiler to another and then, using the « check » and « control » functionalities update the language-specific rules. Figure 3 shows a startup window to access sets of language-dependent rules for tokenization, tag disambiguation, preediting, chunking, defining shallow semantic dependencies, predicate template tree builder, case-roles syntactic transfer, linearization, postediting.

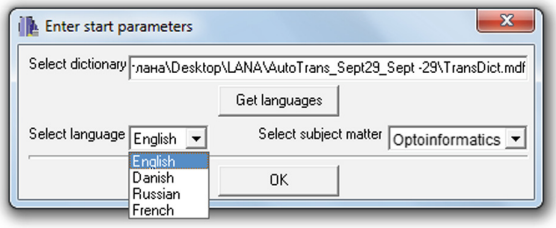

Fig. 3. Compilers' startup window

3.2 Generic Features of Rule Formalisms

All rules are completely or partially formulated over the strings of tag variables and/or word variables. Though particular tagsets for different languages and applications can differ, what only matters in the universal computational formalism is a special top level classification of tags into *single* or *multiple*, *fine* or *coarse* tags. A *single tag* is a one tag symbol as assigned to an unambiguous wordform.

A *single tag* can code any range of linguistic information specified by developers, from simple POS classes [7] to what is included in "supertags" [4, 13–15] or morphosyntactic descriptions MSD [12], etc. Besides, we use the so called border tags to mark the start and end of a certain text segment.

A *multi-tag* is a string of *several single tags* assigned to one ambiguous wordform after, say, lexicon look up.

A *fine tag* is a tag assigned to one wordform after lexicon look up. A *fine tag* can be a *single-tag or multi-tag.*

A *coarse tag* is a tag that codes a *group of single tags* with the same morphosyntactic behavior. For example, if a tagset includes separate tags for different verbforms depending upon voice, person, tense, number etc., then a coarse tag can correspond, for example, to all verb forms or to the groups of verb tags in passive or active voice, correspondingly.

The rules for a very particular application, evidently, instead of generic tags should contain language-specific tags that should either be declared in a particular compiler or pipelined from lexicons. But the rule formalism is still the same. Every compiler program consists of two parts – a declaration part (optional) and a rule part. In the declaration part a developer can set variables, like lists of specific words or new valid tags to be used in the rules. The formalisms for writing rules are language independent, quite simple, though well-defined semantically, and, as said above, are formulated in terms of generic types of tags (see next section).

All compiler descriptions share the declaration part and rule format, - the IF-THEN-ELSE-ENDIF structure, where the IF block (rule conditions) can contain simple or complex conditions. Complex conditions can be formulated with the use of the binary logic operators AND, OR, and the operator NOT. The difference between particular rule formalisms used by every compiler program lies in the rule simple conditions and actions that are specific for every particular processing procedure. Below, we illustrate the developer environment and rule formalisms with the detailed descriptions of the tag disambiguation compiler and syntactic transfer compiler.

3.3 Tag Disambiguation Compiler

The input to the disambiguation compiler is the output of the first tagging procedure that is "welded in" the tagger. The input consists of border tags, as specified in the tokenization compiler, and text strings tagged with single or multiple fine tags assigned by the lexicon look up. The (condition) right-hand side of the tag disambiguating rules uses context information in terms of tags with attributes or words within a 5-word window with the tag/word in question in the middle. Figure 4 shows the compiler interface for writing rules. The left panes display tagset and help to format rules.

Below, for our fellow linguists, we in detail comment the disambiguation rules formalism, where the following notions are used.

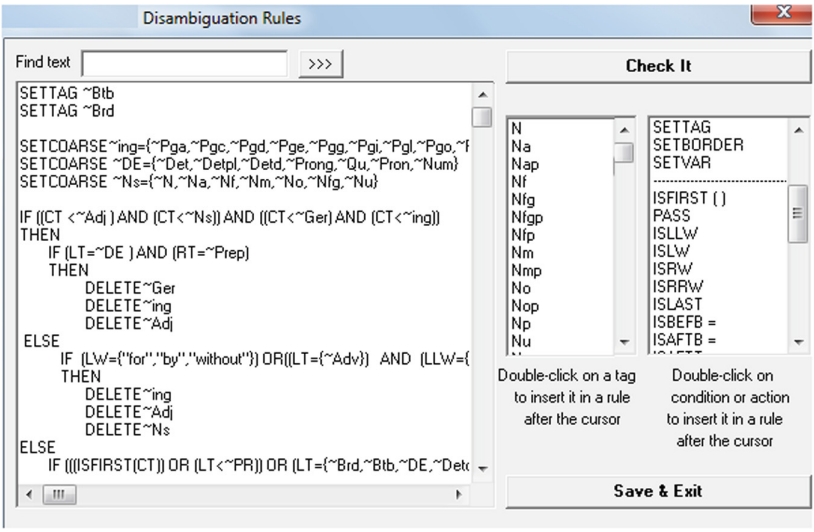

Fig. 4. A screenshot of the compiler interface for tag disambiguation rules (Here and in other exemples language-dependent tags are those specified in the patent-related application lexicon for English, see, e.g., (Sheremetyeva, 2004, 2007)).

CW means a current word, LW is a word to the left of CW, RW is a word to the right of CW, LLW is a word to the left of the left of CW, RRW means a word to the right of the right of CW. Everyone of these variable names can be used either on their own, or specified by some attributes, for example, as follows:

CW = {"means"}, LW = {"comprising", "having", "including"} or RRW = ListA, where listA is declared as SETVAR ListA = {"comprising", "having", "including"}.

ISLW, ISRW, ISLLW, ISRRW are conditions specifying that in the analyzed string there are words to the left, right, left-of-left, right-of-right of the current word;

CT means a current tag, LT means a tag to the left of CT, RT is a tag to the right of CT, LLT is a tag to the left of the left of CT, RRT is a tag to the right of the right of CT. Similar to the word variables the tag variables can be used either on their own, or specified by the tag attributes (parameters), for example, as follows: RT = {∼Brd, ∼Btb, ∼Conj, ∼DE, ∼Pg}, LT = {∼Adv}, or CT < ∼Ns meaning that a current tag CT (multiple or single) includes a single tag component that is listed in a set of values of the coarse tag ∼Ns. This coarse tag should be declared in the compiler, and in our example ∼Ns is declared with the tag values for nouns of different semantic classes, see Fig. 4. The rule displayed in the screenshot in Fig. 4 disambiguates a multiple tag that could be assigned, e.g., to such a word as "opening": {opening} ∼Adj∼N∼P∼Ger, which means that this word can be an adjective, singular noun, present participle active or gerund. ISLAST, ISFIRST denote conditions, specifying that CT is the last/first in the segment. A processing module (the tagger in this case) can make several passes through he rules and the condition PASS <compare operation> <number> specifies the ordinal number of the pass, during which a certain part of the rules should be applied.

The compiler program description in the EBNF language

```
<program> :=
   [<declare-part>]
    <rule-part>

<declare-part> :=
[SETVAR <variable-name> = {"<string>" [, "<string>"
[,...]] } | SETTAG <tag>]
   . . .
[SETCOARSE <coarse-tag> = {<single-tag> [,<single-tag>
[,...]]}]

<rule-part> :=
   [<if-block>]
      . . .
<if-block> :=
   IF <condition>
   THEN
      <if-block> | <action-block>
   [ELSE
      <if-block> | <action-block>]
   ENDIF

<condition>  := <simple condition> | ( <condition> ) <bi-
nary logic operator> ( <condition> ) | NOT ( <condition>)

<simple condition> :=
  <tag variable> = {<multi-tag> [,<multi-tag> [,...]]}|
  ISFIRST (<tag variable>) | ISLAST|
  PASS <compare operation> <number> |
  <word variable> = {"<string>" [,"<string>" [,...]]} |
  <word variable> = <variable-name> |
  <tag variable> < <sinlge-tag> |
  ISLW | ISRW | ISLLW | ISRRW |

<compare operation> := = | < | >

<binary logic operator> := AND | OR |

<tag variable> := CT | LT | RT | LLT | RRT |

<word variable> := CW | LW | RW | LLW | RRW |

<action-block> :=
   [<action>]
    . . .
   <action> :=
   CT = <multi-tag> |
   DELETE <single-tag>

<multi-tag> := <single-tag>[<multi-tag>]
<single-tag> := ~<tag symbols>
```

3.4 Syntactic Transfer Rule Compliler

This compiler is used in our machine translation applications to acquire rules for syntactic restructuring and agreement in the strings of words that fill the case-role slots in the final parse structure (see Fig. 1) after TL lexicalization. In other words, the input to this compiler is separate strings of words in TL in base forms; the output is the correctly translated TL phrases filling particular case-roles in the TL predicate-argument structures.

At this stage, the feature space to formulate rule conditions includes the knowledge about SL and TL equivalents as specified in the bilingual lexicon and the knowledge produced by the previous processing steps that, in turn, includes

(a) the base forms of TL words with their base form tags,
(b) semantic classes of the predicates (cross-linguistic invariants) governing case-roles filled with the strings to be translated,
(c) case-role types (cross-linguistic invariants), to which the strings belong,
(d) phrase types (e.g., NP), to which the strings belong (in case processing included the phrase chunking stage[1]), and
(e) tag histories. The tag history is the knowledge about the "old" SL disambiguated tag of the SL word of its TL equivalent. The tags can only be single fine or coarse tags as multiple tags are disambiguated by this time.

All coarse tags, both for TL and SL should be necessarily declared in the compiler like, e.g., SETCOARSE ∼NounInstr = {∼Nfi, ∼Ni, ∼Nni, ∼Detdi, ∼Npersi, ∼Nai}. The TL fine tags are automatically taken from the lexicon, while the SL tags that are to be used in tag history should be declared in the compiler as SET-TAG ∼Nameg. It is also necessary to declare lists of TL words that can be used as attributes (or parameters): SETVAR ListDifferent = {"different", "various", "similar"}.

The syntactic transfer rules are formulated in terms of tag templates composed of the strings of fine and/or coarse tags. The condition part contains "raw" TL tag templates as produced by lexicalization; the action part contains a template corresponding to a well-formed TL phrase. A "raw" tag template can be converted into a well-formed template by reordering, deleting, changing or inserting new tags associated with certain words. To make it possible the tags can be conditioned by the following attributes or parameters (Fig. 5):

(a) the number of their possible repetitions in the tag template, for which the Kleene + or * are used,
(b) their "parent" old SL tag: ∼Gerund{∼oldNoun},
(c) inclusion of the words from a certain list: Adj{ListDifferent},
(d) exclusion of the words, from a certain list: ∼Adj(- ListDifferent} and (e) ordinal numbers in the output template in case a "raw" template contains several coinciding tags. This is done to apply appropriate changes to the relevant template component, in case the raw template contains, e.g. several nouns.

[1] It might not always be the case as there are applications that skip phrase chunking and work directly on lexicon tags.

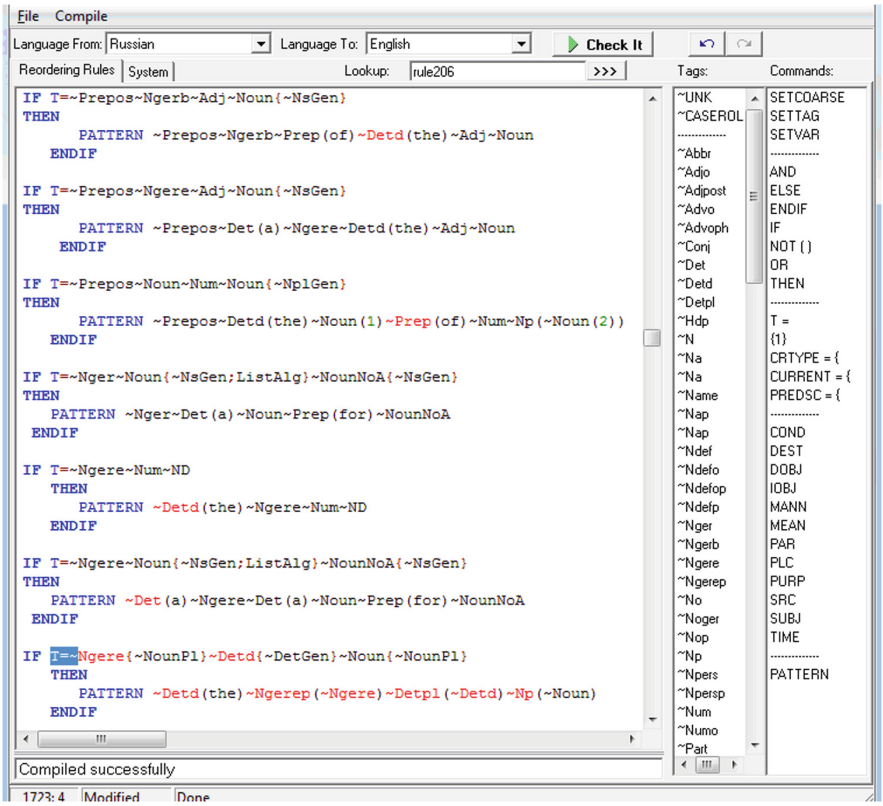

Fig. 5. A screenshot of the compiler interface with the rules for machine translation for patent claims and scientific and technical papers from Russian into English.

In the EBNF description of the compliler the following notations are used: T = ∼Tag∼Tag2… is a "raw" template; ∼Tag can have parameters (a) – (e) listed above;

PATTERN is an output pattern, indicating all changes.

PREDSC means semantic class of the predicate as specified in the lexicon. CRTYPE is case-role type SUBJ, DOBJ, IOBJ, PLC, MANN, … PAR are case-role notations as specified in the system lexicon, meaning, "subject", "direct object", "indirect object", "place", "manner", … "parameter", correspondingly.

The compiler program description in the EBNF language

```
<program> :=
 [<declare-part>]  // as in tagging;
 <rule-part>       // as in tagging but simple condition;
  ...
 <simple-condition>:=
   T = ~Tag[ {<tag-parameters>} ] ...  |
   PREDSC = {"sem-class", ... } |
   CRTYPE = {<cr-type> , ...}   |
   CURRENT = {~Border-tag=Phrase tag | ~CASEROLE, ...}

   <cr-type> := SUBJ | DOBJ | IOBJ | PLC | MANN | MEAN |
PURP | COND | TIME | SRC | DEST | PAR

  <tag-parameters> := [ + | * | <Number>]; [[-]ListName];
[[-]OldTagInLanguageFrom]

 <Number> := 1 | 2 | 3 | ...

<action>:=

   PATTERN <tags>[+]...

<tags> :=

    ~Tag [(<Number>)] |
    ~Tag (<word or words>) |
    ~NewTag (~Tag[(<number>)])
```

4 Conclusions

The paper presented a set of portable computational formalisms for rule-based tagging, analysis, transfer and generation that allow migrating from one rule-based application to another within one language or cross-linguistically. All formalisms are implemented in processing modules and developer environment that includes a lexicon shell with flexible settings and a number of compilers for writing rules in language independent formalisms. The developer environment can be used by linguists without advanced programming skills and allows acquiring or editing lexical resources, specifying tags, writing processing rules, and control the correctness of processing. The formalisms and compilers described have been successfully used (in different combinations and with different depth of analysis) in a number of unilingual and multilingual applications in English, Danish, French and Russian.

References

1. Camilleri, J.J.: An IDE for the grammatical framework. In: Proceedings of a Workshop on Free/Open-Source Rule-Based Machine Translation, Gothenburg, Sweden, pp. 1–12 (2012)
2. Feldman, A., Jirka, H., Brew, Ch.: A cross-language approach to rapid creation of new morpho-syntactically annotated resources. In: Proceedings of LREC 2006 (2006)
3. Joshi, A.K.: Unification and some new grammatical formalisms. In: Proceedings of the Workshop on Theoretical Issues in Natural Language Processing TINLAP 1987, pp. 45–50. Las Cruces, New Mexico. USA (1987)
4. Joshi, A.K, Srinivas, B.: Disambiguation of super parts of speech (or Supertags): almost parsing (1994). http://acl.ldc.upenn.edu/C/C94/C94-1024.pdf
5. Neumann, C.: Generating patent claims in english from a Japanese-only interface. In: Proceedings of the Workshop on Patent Translation in conjunction with the MT Summit X. Phluket, Thailand, September 2005
6. Paul, M.: Translation knowledge recycling for related languages. In: Proceedings of MT Summit VIII. Santiago de Compostela, Galicia, Spain, 18–22 September 2001
7. Petrov, S., Das, D., McDonald, R.: A universal part-of-speech tagset. In: Proceedings of the Conference on Language Resources and Evaluation (LREC 2012). Istanbul, Turkey (2012)
8. Pereira, F.C.N., Shieber, S.M.: The semantics of grammar formalisms seen as computer languages. In: Proceedings of the Tenth International Conference on Computational Linguistics. Stanford, CA, USA (1984)
9. Pinkham, J., Corston-Oliver, M., Smets, M., Pettenaro, M.: Rapid assembly of a large-scale French-English MT system. In: Proceedings of MT Summit VIII. Santiago de Compostela, Galicia. Spain, 18–22 September 2001
10. Przepiórkowski, A.: A preliminary formalism for simultaneous rule-based tagging and partial parsing data structures for linguistic resources and applications. In: Rehm, G., Witt, A., Lemnitzer, L. (eds.) Tübingen:Gunter Narr Verlag, pp. 81–90 (2007)
11. Sharoff, S., Nivre, J.: The proper place of men and machines in language technology processing Russian without any linguistic knowledge. In: Proceedings of the International Conference on Computational Linguistics "Dialogue"- 2011. Russian State Humanitarian University, Moscow (2011)
12. Sharoff, S., Kopotev, M., Erjavec, T., Feldman, A., Divjak, D.: Designing and evaluating a Russian tagset. In: Proceedings of the Sixth Language Resources and Evaluation Conference, LREC 2008, Marrakech (2008)
13. Sheremetyeva, S.: Application adaptive electronic dictionary with intelligent interface. In: Proceedings of the Workshop on Enhancing and Using Electronic Dictionaries in Conjunction with the 20th International Conference on Computational Linguistics. COLING 2004, Geneva, Switzerland, 23–28 August 2004
14. Sheremetyeva, S.: On portability of resources for quick ramp-up of multilingual MT for patent claims. In: Proceedings of the workshop on Patent Translation in conjunction with MT Summit XI, Copenhagen, Denmark, 10–14 September 2007
15. Sheremetyeva, S.: On teaching technical writing with an authoring tool. In: Proceedings of the 10th International Technology, Education and Development Conference INTED 2016, Valencia, Spain, 7–9 March 2016
16. Takeda, K.: Portable knowledge sources for machine translation. In: Proceedings of the 15th International Conference on Computational Linguistics, Kyoto, Japan, 5–9 August 1994