Software Tools for Morphological and Syntactic Analysis of Natural Language Texts

Jemal Antidze, and David Mishelashvili

Tbilisi State University, Vekua Institute of Applied Mathematics
antidze@viam.hepi.edu.ge, dm@internet.ge

Abstract. The “Software Tools for Natural Language Texts Processing” is a software system designed for syntactic and morphological analysis of natural language texts.
The tools are efficient for a language which has free order of words and very developed morphological structure like Georgian. For instance, a Georgian verb has several thousand verb forms and it is very difficult to construct finite automaton for establishing a verb form’s morphological categories as well it will be inefficient. Splitting a Georgian verb form into morphemes requires nondeterministic search algorithm, that has many backtracking. To minimize backtracking it is necessary to put constraints that exists among morphemes and verify it as soon as possible to avoid false directions of search. It is possible to minimize backtracking and use parameterized macro insertions by our tools. Software tool for syntactic analysis has means to reduce rules, that have the same members in different order.
Thus proposed software tools have many means to construct efficient parser, test and correct it without programming.

1. Introduction

The “Software Tools for Morphological and Syntactic Analysis of natural Language Texts” is a software system designed for natural language texts processing. The system is used to analyze syntactic and morphological structure of the natural language texts. Specific formalisms has been worked out for this purpose allow us to write down syntactic and morphological rules defined by particular natural language grammar [1]. These formalisms represent a new, complex approach that solves some of the problems connected with the natural language processing. A software system has been implemented according to these formalisms. Syntactic analysis of sentences and morphological analysis of word-forms can be done within this software system. Several special algorithms were designed for this system. Using formalisms described in [2-3] are very difficult for Georgian language.

The system consists of two parts: syntactic analyzer and morphological analyzer. Purpose of the syntactic analyzer is to parse an input sentence, to build a parsing tree that describes relations between the individual words within the sentence, and to collect all important information about the input sentence that was figured out during the analysis process. It is necessary to provide a grammar file to the syntactic analyzer. There must be written syntactic rules of particular natural language grammar in that file. Syntactic analyzer also needs information about the grammar categories of the word-forms of natural language. Information about the grammar categories of the word-forms are used during the analysis process. However it may be quite difficult to include all of the word-forms from the natural language into a dictionary file. To avoid this problem, and to reduce size of dictionary file, morphological analyzer is used. Morphological analyzer uses a dictionary file of unchanged parts of words. Therefore this file will be considerably smaller, because many word-forms can be produced by single unchanged part of word. The morphological analyzer also needs its own grammar file. According to the specific formalism, morphological rules of natural language must be written in that grammar file. An input word is divided into the morphemes when applying these rules. And important information about the grammar categories of word-form can be deduced during the analysis.

An input sentence is passed to syntactic analyzer. Syntactic analyzer passes each word from the sentence to the morphological analyzer. Morphological analyzer will analyze the words according to the rules from the grammar file, using a dictionary of words' unchanged parts. After the successful analysis each word-form will obtain information about its grammar categories, and this information will be returned to the syntactic analyzer. At the end syntactic analyzer will try to parse the sentence according to the rules from the syntax file.

Basic methods and algorithms, that were used to develop the system the, are: operations defined on the feature structures, trace back algorithm (for morphological analyzer), general syntactic parsing algorithm and feature constraints method. Feature structures are widely used on all level of analysis. As an abstract data types they are used to hold various information about dictionary entries. Each symbol defined in a morphological or syntactic rule has an associated feature structure, which is initially filled from the dictionary, or it is filled by the previous levels of analysis. Feature structures and operations defined on them are used to build up feature constraints. With general parsing algorithm it is possible to get a syntactic analysis of any sentence defined by a context free grammar and simultaneously check feature constraints that may be associated with grammatical rules. Feature constraints are logical expressions composed by the operations that are defined on the feature structures. Feature constraints can be attached to rules defined within a grammar file. If the constraint is not satisfied during the analysis, then the current rule will be rejected and the search process will go on. Feature constraints also can be attached to morphological rules. However, unlike the syntactic rules, constraints can be attached at any place within a morphological rule, not at the end only. This speeds up morphological analysis, because constraints are checked as soon as they are met in the rule, and incorrect word-form divisions into morphemes will be rejected in a timely manner.

Formalisms that were developed for the syntactic and morphological analyzers are highly comfortable for human. They have many constructions that make it easier to write grammar files. Morphological analyzer has a built-in preprocessor, which has a capability to process parameterized macro insertions.

The software system is written in C++ programming language standard. It uses STL standard library. Program operates in UNIX and Windows operating systems. Although the program could be compiled and used in any other platform as well, which contains modern C++ compiler.

2. Feature Structures

A feature structure is a specific data structure. It essentially is a list of “Attribute - Value” type pairs. The value of an attribute (field) may be either atomic, or may be a feature structure itself. This is a recursive definition; therefore we can build a complex feature structure, with any level of depth of nested sub-structures.
Feature structures are widely used in Natural Language Processing. They are commonly used:

1. To hold initial properties of lexical entries in the dictionary
2. To put constraints on parser rules. Certain operations defined on feature structures are used for this purpose.
3. To pass data across different levels of analysis

We use following notation to represent feature structures in our formalism. List of “Attribute – Value” pairs is enclosed in square braces. Attributes and values are separated by colon “:”. In example:

```
S = [A: V1
     B: [C: V2]]
```

It is possible to use short-hand notation for constructing feature structures. We can rewrite above example this way:

```
T1 = [A: V1]
T2 = [C: V2]
S = [(S, T1) B: T2]
```

Content of the feature structures listed in the parentheses at the beginning is copied to the newly constructed feature structure.
Below is a fragment of a formal grammar for defining feature structures in our formalism:

```
<feature_structure> ::= “[”[<initialization_part>]
[<list_of_pairs>] “]”
<initialization_part> ::= “(” {<initializer>} “)”
<initializer> ::= <variable_reference> |
<constant_reference>
<list_of_pairs> ::= { <pair> }
<pair> ::= <name> “:” <value>
<name> ::= <identifier>
```

```
<value> ::= “+” | “-” | <number> | <identifier> |
<string> | <feature_structure>
. . .
```

There are several operations defined on feature structures to perform comparison and/or data manipulation. Mostly well known operation defined on feature structures is unification. In addition to the unification, we have introduced other useful operations that simplify working on grammar files in practice. The result of each operation is a Boolean constant “true” or “false”. Below is a list of all implemented operations and their semantics:

- `A := B` (Assignment) Content of the RHS (Right Hand Side) operand (`B`) is assigned to the LHS (Left Hand Side) operand (`A`). Thus their content becomes equal after the assignment. The assignment operation always returns “true” value.
- `A = B` (Check on equality) This operation does not modify content of the operands. Result of the operation is “true” when both operands (A and B) have the same fields (attributes) with identical values. If there is a field in one feature structure which is not represented in the second feature structure or the same fields does not have an equal values then the result is “false”.
- `A <== B` (Unification) Unification returns “true” when the values of the similar field in each feature structure does not conflict with each other. That means, either the values are equal, or one of the value is undefined. Otherwise the result of the unification operator is “false”. Fields, that are not defined in LHS feature structure and are defined in RHS feature structure are copied and added to the LHS operand. If there is an undefined value in LHS feature structure, and the same field in the RHS feature structure is defined, that value is assigned to the corresponding LHS feature structure field.
- `A == B` (Check on unification) Returns the same truth value as unification operator, but the content of operands is not modified.

Check on equality or unification operations (`“=”` and `“==”`) may take multiple arguments. In example:

```
X == (A, B, C)
```

Where `X, A, B,` and `C` are feature structures. Left hand side of an operation is checked against each right hand side argument that way. And the result is “true” only when all individual operations return “true”, otherwise “false” if returned.
There is also a functional way to write operations. In example, we can write “equal(A, B)” instead of “A = B“. Following functions are defined “equal” (check on equality), “assign” (assignment), “unify” (unification), “unicheck” (check on unification), “meq” (multiple equality checking), “muc” (multiple unification checking).

3. Constraints

In our system feature structures and operations defined on them are used to put constraints on parser rules. That makes parser rules more suitable for natural language analysis than pure CFG rules. We have generalized notation of constraint [2]. Constraint is any logical expression built up with operations defined on feature structures and basic logical operations and constants: & (and), | (or), ~ (not), 0 (false), 1 (true).
Parser rules are written following way:

$S \rightarrow A_1 \{ C_1 \} A_2 \{ C_2 \} \dots A_N \{C_N\}$

Where S is an LHS non-terminal symbol, A_i are terminal or non-terminal symbols (for morphological analyzer only terminal symbols are allowed), and C_i are constraints. Each constraint is check as soon as all of the RHS symbols located before the constraint are matched to the input. If a constraint evaluates to "true" value then parser will continue matching, otherwise if constraint evaluates to "false" parser will reject this alternative and will try another alternative. There is a feature structure associated with each (S and A_i) symbol in a rule. If a symbol is a terminal symbol then initial content of its associated feature structure is taken from the dictionary or from the morphological analyzer (for syntactic analyzer). Content for a non-terminal symbols is taken from the previous levels of analysis. Constraints are used not only to check the correctness of parsing and reduce unnecessary variants. They are also used to transfer data to a LHS symbol, thus move all necessary information to the next level of analysis. Assignment or unification operations can be used for this purpose. To access a feature structure for particular symbol, a path notation can be used. Path is written using angle brackets. In example, <A> represents a feature structure associated with the A symbol. Individual fields can be accessed by listing all path components in angle brackets.
The formal syntax for a constraint is defined this way (fragment):

```
<constraint> ::= <constraint_term> “|”<constraint_term>
<constraint_term> ::= <constraint_fact> “&”
<constraint_fact>
<constraint_fact> ::= [“~”] ( <logical_constant> | “+” |
“-” | <constraint_operation> | “(” <constraint_fact> “)”
)
<logical_constant> ::= “0” | “1”
<constraint_operation> ::= < constraint_operator> |
<constraint_function>
<constraint_operator> ::= <constraint_argument> (“:=” |
“==” | “<==”, “=”) (<constraint_argument> |
<list_of_constraint_arguments>)
```

```
<constraint_function> ::= <identifier>
<constraint_function_arguments>
. . .
```

4. Morphological analyzer

Purpose of morphological analyzer is to split an input word into the morphemes and figure out grammar categories of the word. Morphological analyzer may be invoked manually, or automatically by the syntactic analyzer.

Special formalism has been created to describe morphology of natural language and pass it to the morphological analyzer. There are two main constructions in the grammar file of morphological analyzer: morpheme class definition, and morphological rules. Morpheme class definition is used to list all possible morphemes for a given morpheme class. In example:

```
@M1 =
      {
      “morpheme_1” [ … features … ]
      “morpheme_2” [ … features … ]
      . . .
      “morpheme_N” [ … features … ]
      }
```

It is possible to declare empty morpheme, which means that the morpheme class may be omitted in morphological rules. Below is formal syntax for morpheme class definition:

```
<morphem_definition> ::= “@” <identifier> “=” “{”
                                    <list_of_morphemes>
“}”
<list_of_morphemes> ::= <morpheme> { “,” <morpheme> }
<morpheme> ::= <string> <feature_structure>
```

Morphological rules are defined following way:

word -> M_1 { C_1 } M_2 { C_2 } . . . M_N { C_N }

Where M_i are morpheme classes, and C_i (i=1,…,N) are constraints (optional).

5. Syntactic analyzer

Purpose of syntactic analyzer is to analyze sentences of natural language and produce parsing tree and information about the sentence. In order to accomplish this task syntactic analyzer needs a grammar file, and a dictionary (or it may use morphological analyzer instead of complete dictionary). Grammar rules for syntactic analyzer are written like CFG rules. But they may have constraints and symbol position regulators. The rule can be written according to these constructions:

$$S \; \text{->} \; A_1 \; \{ \; C_1 \; \} \; A_2 \; \{ \; C_2 \; \} \; . \; . \; . \; A_N \; \{ \; C_N \; \} \; ;$$

$$S \; \text{->} \; A_1 \; A_2 \; . \; . \; . \; A_N \; : \; R \; \{ \; C \; \} \; ;$$

Where S is an LHS non-terminal symbol, A_i (i=1,…,N) are RHS terminal or non-terminal symbols, C and C_i (i=1,…,N) are constraints, and R is a set of symbol position regulators. Position regulators declare order of RHS symbols in the rule, thus making non-fixed word ordering. There are two types of position regulators:

1. $A_i < A_j$ means that symbol A_i must be placed somewhere before the symbol A_j
2. $A_i – A_j$ means that symbol A_i must be placed exactly before the symbol A_j

6. Example of a syntactic analysis

Below is a sample sentence given to the syntactic analyzer:

“cnobili mSenebeli saxls uSenebs megobars” (Georgian, Latin encoding)
“Famious builder builds a home for his friend”

Result produced by the syntactic analyzer:

```
&> Parsing: cnobili(ZS) mSenebeli(AS) saxls(AS) uSenebs(Z)
megobars(AS)
1 solution(s) was(were) found.
Parse Tree 1:
|
ZJG3P:1
|---------------------------------|-------------|--------------|
SPNS:2                            SPNS:3        ZJG:4
SPNS:5
|                                 |             |              |
SJGM:6                            SJGM:7        Z:8  (uSenebs)
SJGM:9
|                                 |                            |
SJG:10                            SJG:11
SJG:12
|---------------|                 |                            |
AT:13           SJG:14            AS:15 (saxls)
AS:16 (megobars)
|               |
ZS:17 (cnobili) AS:18 (mSenebeli)

1: ZJG3P
```

```
[obj1: [brunva: mic
        cat: AS
        lex: saxls
        piri: 3
        ricxvi: mx]
 obj2: [brunva: mic
        cat: AS
        lex: megobars
        piri: 3
        ricxvi: mx]
 pred: [cat: Z
        dro: awmyo
        ir_obj_piri: 3
        ir_obj_ricxvi: mx
        lex: uSenebs
        pir_obj_piri: 3
        piri: 3
        pirianoba: 3
        ricxvi: mx
        seria: 1]
 subj: [brunva: sax
        cat: AS
        lex: mSenebeli
        piri: 3
        ricxvi: mx]]
```

Symbols translation

ZS	Adjective
AS	Noun
Z	Verb
ZJG3P	Verb group 3
SPNS	Noun or pronoun
ZJG	Verb group
SJGM	Driven noun group
SJG	Noun group
AT	Attribute
brunva	Case
piri	Person
ricxvi	Number
dro	Tense
ir_obj	Indirect object
pir_obj	Direct object

7. Example of the construction of a constraint for morphological analysis of Georgian word forms

In order to find out how use our software tools for splitting of Georgian word forms into morphemes and how by received morphemes and corresponding information obtained from dictionary establish the morphological categories, most clearly is shown from morphological analysis of a verb form. Having noted, that one lexical unit of a verb may have several thousand of verb forms. This situation complicates to find the morphological categories of a verb form, while the verb form should be split correctly on morphemes and should be found the formal rules for establishing corresponding morphological categories. Solving of the problem becomes more complicated by the fact, that different lexical units produce verb forms differently. We used the classification of verbs proposed by D.Melikishvili [4] and formal rules are established by us. For better understanding the example discussed in the paragraph, we should look at the general structure of Georgian verbs. In general, Georgian verb morphemes are divided on 10 classes of morphemes, which we encounter in verb forms from left to right according to the class number. If the verb form has any class representative, it must be only one class representative. The neighboring class representatives can be identical, for example $a_1a_3a_4$lebs (light, future tense, third person, singular).

Index shows to which class belongs the morpheme a. Also possible, that some of them does not exist in a verb form. Such situation makes complicated to find to which class belongs concrete a. In each class can be one or several tens morphemes. There are following classes of morphemes: 1. prefix; 2. person prefix; 3. vowel prefix; 4. root; 5. d-passive; 6. theme; 7. causation; 8. series; 9. person suffix; 10. number. Lets look at several examples:

1 a_1-v_2-a_3-$shen_4$-eb_6-$ineb_7$-d_8-i_9-t_{10} (build, first person, future tense, plural, causation);

2. a_1-$shen_4$-d_5-i_8 (build, second person, perfect tense, singular);

3. $cham_4$-d_8-$nen_{9,10}$ (eat, third person, plural, imperfective aspect).

In these examples the index shows the number of the class. Two indexes on one morpheme shows, those two classes are united and they are not dividable. We can see from the examples, that “d” morpheme belongs to two different classes. If the representative of some class does not exist in a verb form, this gives also significant information for finding morphological categories. Classes of morphemes are considered as word forms components and in the dictionary they are written in with features and corresponding meanings. Among morphemes classes, especially important class is root. Some representatives of root compose verb forms equally. They form a class. Each representative of root has its class number, which is considered as a feature. When we intend to find, if the representative of concrete class of morphemes exists in verb form, we write the name of this class in the rule , and when we want to verify if we found the concrete representative of this class, than we write `<the name of class lex> = ″ the meaning of concrete` `morpheme ″`. This is the simple logical expression, which gives the true meaning, in case if during dividing the verb form on morphemes the concrete verb form meaning was found, otherwise we will have false meaning. Which morphemes belong to concrete class is given in the dictionary. For example, we want to find person of the verb form `vasheneb (build, first person, singular, present tense)`, having in mind, that this verb form already is divided on morphemes. We have:

```
<prefix lex> = ″″; <person-prefix lex> = ″v″; <vowel-
prefix lex>=″a″; <root lex> = ″shen″; <d-passive lex> =
 ″″; <theme lex> = ″eb″; <causation lex> = ″″; <series
lex> = ″″; <person-suffix lex> = ″″; <number lex> = ″″.
```

The corresponding constraint we can write in so:

```
[<person-prefix lex> = ″v″  & <person-suffix lex> = ″″]
```

If this constraint is satisfied (the logical expression is true), then the verb form has first person, otherwise we must consider other alternatives. In general, a compound expression consisting of such simple constraints forms more complicated constraints [5]. Here we have the simplified constraint, as we had in mind that the verb form has the present tense and it forms first subjective person by v-i signs of person.

References

1. Antidze, J., Mishelashvili, D.: Instrumental Tool for Morphological Analysis of Some Natural Languages. Reports of Enlarged Session of the Seminar of IAM TSU,vol.19. Tbilisi (2004) 15-19
2. McConnell, S.: PC-PATR Reference Manual, a unification based syntactic parser. version 1.2.2. http://www.sil.org
3. Antworth, E., McConnell, S.: PC-Kimmo Reference Manual, a two-level processor for morphological analysis. version 2.1.0. http://www.sil.org
4. Melikishvili, D.: System of Georgian verbs conjugation. Tbilisi (2001) (in Georgian)
5. Antidze, .J., Melikishvili, D., Mishelashvili, D.: Georgian Language Computer Morphology. Conference – Natural Language Processing, Georgian Language and Computer Technology. Tbilisi (2004) 39-41