

Zemberek, an open source NLP framework for Turkic Languages

Ahmet Afşın Akın
Softek Inc. / Puerto-Rico
ahmetaa@gmail.com

Mehmet Dündar Akın
TUBITAK - UEKAE / Turkey
mdakin@uekae.tubitak.gov.tr

Abstract

Most of the NLP solutions in the IT world are based on Indo-European languages. The inherent challenges of agglutinative languages and lack of present solutions for Turkic languages called for an extensible, general purpose NLP library. Although it started with Turkish, Zemberek project now aims to fill this space and provide a flexible, open source, platform independent NLP framework not only for Turkish but also all Turkic languages. In This paper we will try to explain the architecture of Zemberek NLP framework

1 Introduction

Today, Natural Language Processing (NLP) is one of the most popular and challenging subjects of computing. Even though countless research has been done, only a handful of successful real life products emerged. Because of the fundamental differences of agglutinative languages i.e. extreme usage of affixes, making NLP research based on those languages is much more difficult. For Turkic Languages situation is even worse. Even though Turkic Languages are spoken by around 140 million people from Europe to Siberia[1], because of the lack of open computing libraries, performing even the simplest NLP operations such as spell checking is troublesome. There have been many academic studies on the subject, but most of the time only a specific Turkic Language[2], especially Turkish[3] [4] [5] was used. Most importantly, there existed almost no usable open source library¹ until the introduction of Zemberek. Zemberek is now aiming to provide a generic NLP framework not only for Turkish, but also for other much neglected Turkic languages. Currently, the

framework provides basic NLP operations such as spell checking, morphological parsing, stemming, word construction, word suggestion, converting words written only using ASCII characters (so called 'deasciiifier') and extracting syllables.

2 Structure

The library consists of two main parts; language structure information and NLP operations. Core library contains NLP specific algorithms and provides necessary tools to the language implementations. Although core library is designed specifically for Turkic languages, it does not contain any specific language implementation. In order to provide this flexibility, several helper mechanisms and abstractions are employed. Each language implementation is responsible for complying with predefined grammar requirements and providing necessary language data. Most likely, NLP related operations do not need to be modified for a new language implementation. After a language is implemented, core NLP functions use these information in a generic manner and provides services to the end users through an easy to use software access mechanism.

Implementing a Turkic language is relatively easy. In Zemberek, in order to make language developer's work easier, some language data are externalized to text based configuration files. But externalization usually affects flexibility and performance in a negative way. Therefore, some information such as special cases and suffix production mechanism are kept in the code. In Figure-1, language information elements and necessary external language data are shown.

¹ Although there seems to be an open source framework named TOY[12] was developed for Turkish, we could not reach the public source code

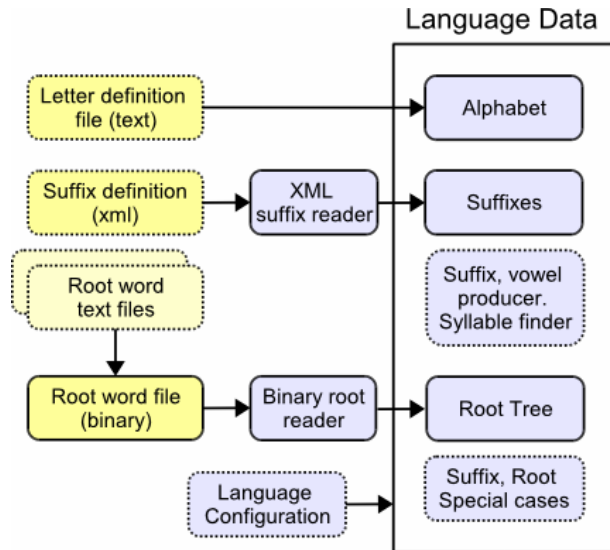


Figure-1

Here is the list of steps to incorporate a new Turkic language implementation into Zemberek.

- A letter file containing alphabet and related letter data in it.
- A suffix file containing suffix data.
- A text file containing root words and special case tags.
- Optional cache file containing most used words to be used in spell checking.
- Optional syllable finder.
- A suffix production class which will interpret the suffix production elements to create a concrete suffix for a certain word.
- If there is any special case for suffix generation, a class for containing those special cases, and possibly a class per special case.
- A class for root word special cases and a container for it. The container may have specific methods for interpreting the special cases for producing altered versions of root words.
- A helper for parsing operation. It is usually almost empty if abbreviations are not parsed.

- A class containing basic information about the language specific classes and some simple hash table data; such as word type names used in root word file, names of the root suffixes, names of the root word files .

Creating a minimal working set of data for languages is a trivial job for a software developer. However, inevitably, adding all language knowledge takes time and may require strong Language knowledge. Now we will explain those elements in detail.

2.1 Letters and Alphabet

Each language must define its alphabet information. For this, there is a simple text based file utilized to carry alphabet information. The file contains Turkic Language specific alphabet information such as letters, vowels, type of vowels (like frontal or round – *ince*, *yuvarlak*), unvoiced consonants (*sert*), non-ASCII letters. Information provided in this file is parsed and stored in special Turkic letter objects. Therefore each Turkic letter object carries its own grammatical information. In the framework, Alphabet is used as an access point to the letters information. Alphabet also provides common helper methods like preparing or validating a string before using in NLP related functions.

2.2 Suffix information

The core of all Turkic languages are suffixes (We will not use the term “affix” since framework is based on “suffixes”). Suffixes are defined in a special XML configuration file.

The configuration file contains two main sets of information, suffix groups (ek-kumeleri, ek-kumesi) and individual suffix information (ekler, ek). Suffix groups are used for convenience and suffix information elements contain the actual suffix data. Here is an example from the Turkish suffix file :

```
<ek ad="ISIM_COGUL_LER" uretim="1Ar">
<ardisil-ekler>
<kume>ISIM_HAL</kume>
<kume>IMEK_ZAMAN</kume>
<aek>ISIM_SAHIPLIK_BEN_IM</aek>
<aek>ISIM_SAHIPLIK_SEN_IN</aek>
<aek>ISIM_SAHIPLIK_O_I</aek>
...
</ardisil-ekler>
</ek>
```

Each suffix needs to define a unique name (attribute “ad”) which can be different depending on the language. Currently we use our own definitions for suffix names, but the naming scheme can be improved by adding standard based names. There is no agreed standard naming scheme defined for Turkish yet. Most suffixes contain a production word (attribute “uretim”). This word represents the production elements of a suffix. Later, those production elements and information from the word the suffix is to be appended will be used for forming an actual suffix.

For now, There are three types of suffix production elements defined:

1. Letters (represented by small case letters): They are directly added to the suffix when a specific suffix word is created.
2. Vowel rule elements (represented by capital letters as in A, I, E): they represent different vowel production rules. For Turkish, A means an 'a' will be added to the word if the appended word's last vowel is not frontal (a, ı, o, u), else 'e' will be produced.
3. First Letter addition or modification: they modify or add a new letter in certain circumstances, such as '+n' represents 'n' is added if the appended word ends with a vowel.

An example is shown below.

suffix production word: 'lar'

suffix production elements: [letter:l, vowel rule:A, letter:r]

word to be appended: elma

suffix producer result: 'lar'

One common characteristic of the suffixes is that apart from some special conditions, they can only be followed by certain suffixes. Actually Zemberek's main morphological parser is based on this simple principle. Therefore most suffix definitions contain subsequent suffix information (“ardisil-ekler” element). For convenience, subsequent suffix information may be individual suffix names, suffix sets or a complete copy from another suffix.

There are also special “initial suffixes” used for determining the starting point of the suffix tree. This is because when we want to add a new suffix to a root word (such as a noun without any suffix),

not all suffixes can be added. The type of the word gives us a hint for where to start. Therefore, we first add an empty initial suffix which carries the list of subsequent suffix information that can be appended to that root word. In Turkish implementation, they are marked as “KOK - root”. Here is an example for a *initial suffix* defined for the type “number”.

```
<ek ad="SAYI_KOK" uretim="">
<ardisil-ekler kopya-ek = "ISIM_KOK">
<aek>SAYI_ULESTIRME_ER</aek>
<aek>SAYI_KESIR_DE</aek>
<aek>SAYI_SIRA_INCI</aek>
<aek>SAYI_TOPLULUK_IZ</aek>
<aek>SAYI_KOSE_GEN</aek>
</ardisil-ekler>
</ek>
```

2.3 Suffix special cases

Special cases for suffixes exist in all Turkic languages. For example, in Turkey, Turkish continuous tense suffix drops the last letter of the appended vowel. Or, passive suffix may form differently depending on the last consonant.

ara→**ar-ıyor**, not 'ara-yor' or 'ara-ıyor'

kes→**kes-il-mek** (to be called, to be searched)

gel→**gel-in-mek**→ (to be come), not **gel-il-mek**

One common special case is change of the suffix form depending on the previous suffix type. They are defined as “ON_EK” (previous suffix) special cases.

```
<ek ad="ISIM_YONELME_E" uretim="+yA">
<ozel-durum ad="ON_EK" uretim="nA">
<on-ek ad="ZAMAN_BELIRTME_KI"/>
<on-ek ad="ISIM_BULUNMA_KI"/>
<on-ek ad="ISIM_SAHİPLİK_O_I"/>
<on-ek ad="ISIM_TAMLAMA_I"/>
</ozel-durum>
....
```

Here normal suffix production rule is defined as “+yA” which produces {a,e,ya,ye} suffixes for Turkish (For example “*kedi-ye bak*” - “look at the cat”) . However, if this suffix comes after 3rd person possessive suffix (that suffix is defined in related “ON-EK” element), it uses “nA” production word (as in example “*Ahmet'in kedi-si-ne bak*” - “look at Ahmet's cat”) .

Some of these special cases may be eliminated by introducing a full blown suffix state machine [2], but since these cases are not frequent, Zemberek

preferred following the simple tree based approach.

Once suffix file is loaded, data is transferred to Suffix objects. Not all the suffix data are represented in this file, some data is calculated after data has been read. Such as whether a suffix can start with a vowel or which letters can be the first letter for that suffix. Suffix objects also contain list of possible subsequent suffixes and special cases attached to that suffix. There are common special cases for all languages defined in the core library, however, each language must provide its own special cases.

3 Root word dictionary

A *root word* represents a meaningful word without a suffix. Root words are defined in a text-based file. For example, a part of the root definition file for Azeri Language may look like this:

```
sağlıq AD YUM
al EY
gel EY
istiot AD
bir RA
dünen ZAMAN
```

The format is very simple, it has the root word, type of the word and if exists, special case(s). For making the files easy to modify by hand, most tags are defined using short words. The word type names are language dependent. Each language defines its own word type and special case names. For performance reasons, in runtime, Zemberek does not use the text file directly, instead, it uses a special binary representation of the root file.

3.1 Root Special Cases

In Turkic languages, there are special cases for root words. Most of these cases occur in loan words, not originally existing in the language. However, some cases are actually part of the language and they are processed as special cases because of the algorithmic difficulties. Some examples from Turkey Turkish:

saat→saatler, not *saatlar*. Breaks vowel harmony, second 'a' is pronounced frontal. (of Arabic origin).
red→reddi, not *redi*. Last consonant repeats (of Arabic origin).

burun→burnu, not *burun-u*. Last vowel of the root drops. (Original Turkish grammar rule.)

su→suyu, not *sunu*. This is only for word "su" - water

ben→bana, not *bene*. Vowel changes without a rule. this is only for first person subjects "ben" and "sen"

Special cases have properties such as whether the case is root altering, whether it is optional or whether it occurs if the following suffix starts with a vowel. If a particular root special case modifies the root word, (like converting word *kitap* to *kitab*) it uses a *word modifier*. Word modifiers are generic simple objects designed for altering words. For example, in the special case of dropping a noun's last vowel, a word modifier which erases the last vowel from a word is used.

Since most root special cases occur only when followed by certain suffixes, rather than relying solely on configuration files, root special cases require some custom coding in Zemberek.

3.2 Root word Tree

Since Zemberek uses a root dictionary-based parser, it requires a root finding mechanism. Parser starts the parsing operation by finding the root candidates.

During the initialization of the library, Zemberek first loads the binary root file. Once a root word is read, related special cases are attached to the root object, and the resulting object is stored into a special Direct Acyclic Word Graph (DAWG) tree to provide fast access and ease of extensibility. Simplified structure of such a tree is shown in Figure-2.

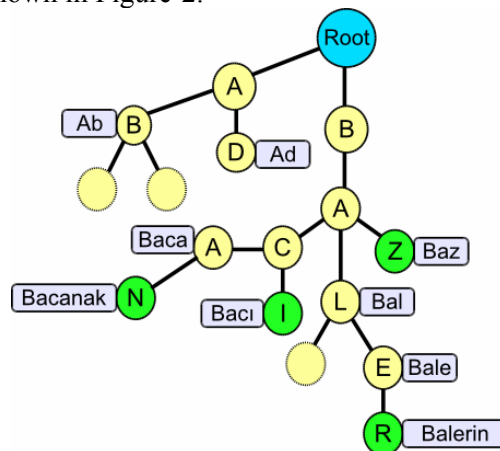
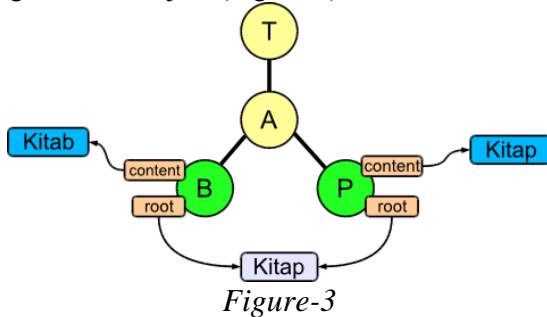


Figure-2

After the root word is added to the tree, the system also calculates the deformed states of the root word which may occur if a special case is applied to it. This is necessary because the word to be parsed may not contain the whole root word within. Such as in Turkish, the word "*kitaba*" does not contain the root word "*kitab*" in it because the last letter is changed after a vowel-starting-suffix is appended. Therefore, the calculated deformation "*kitab*" is also added to the tree with the reference to the original Root object (Figure-3).



This tree is used as a dictionary and it allows different kinds of root selectors to be implemented easily. Currently, there are three root selectors in Zemberek; the first one is used for normal strict root selection for a given word. For example, for the Turkish word "*elmaslar*" it finds *{el:noun, elma:noun, elmas:noun}* set. Second selector applies a tolerance level while selecting candidate roots using a string similarity algorithm. Naturally, It generates more results then the strict selector. The third selector has a tolerance for letters which do not exist in the ASCII encoding. The tree and selectors are completely independent from the language implementations.

4 Standard Morphological Parser

Morphological parser basically finds the possible root and suffixes of a given word. Its structure can be defined as a simple dictionary based top-down parser.

There are two main steps in morphological analysis of an input word:

- Preprocess the word.
- Find root candidates for the input word using appropriate root selector,
- For each root candidate, continue adding possible suffixes to the root, apply special cases if necessary, until either the input

word is constructed, or there are no suffix alternatives left.

- Post process the parser results.

Preprocessing is accomplished by preparing the word to the parsing operation. This includes removing accents, hyphens etc. and converting it to lowercase. If the word contains characters that cannot occur in the defined alphabet, operation terminates. Root candidates are root words that may appear as the starting part of the input word. For finding the root candidates, depending on the operation, appropriate root selectors are used. After the root candidates are found, the system finds the first *root suffix* depending on the type of the root word. As explained above, *initial suffix* (and almost all suffixes) contains a list of suffixes that can follow it.

So we start from the first suffix and check if it causes a root special case. If the suffix actually creates a special case, the modification is applied to the root word. For example, if the root was "*burun*", and if the following suffix starts with a vowel, the parser drops the last vowel from the root, (it becomes "*burn*") by applying the *word modifier defined as part of the suffix*. Then a suffix function is called using the input word, and the currently formed word. The Suffix receives the parameters and by using its own suffix production elements, it creates the suffix for the currently formed word using a language dependent mechanism called "*suffix producer*". Each suffix producer is language dependent, they are responsible for creating concrete suffixes by applying language specific operations. A sample suffix production pseudo-code for Turkish is as follows:

```
produce( current, input , suffix_production_elements)
    last_vowel = current.last_vowel
    for each element in suffix_production_elements
        switch (element.rule)
            case LETTER:
                result.append(element.letter)
            case COMBINE:
                if (current.last_letter is vowel)
                    result.append(element.letter)
            case HARDEN:
                if (current.words last_letter is unvoiced)
                    result.append(element.letter.harden)
                else
                    result.append(element.letter)
            case VOWEL_TYPE_A:
                last_vowel= vowelProducer.typeA(last_vowel)
                result.append(last_vowel)
            case VOWEL_TYPE_I:
                last_vowel= vowelProducer.typeI(last_vowel)
                result.append(last_vowel)
    return result
```

After a suffix is produced, it is compared to the input word to determine if it matches correctly. Different types of word comparators can be used depending on the operation, such as exact, ASCII tolerated or error tolerated. If a successful suffix is created, the current formed word and the suffix state is stored in a stack. and operation continues with the subsequent suffix information. Once the input word and the root with a series of successful suffixes are exactly matched, either the system early returns the result (i.e if it is a simple spell check operation), or it tries to follow other possible solutions until there are no suffix matches left. A simplified block diagram of parser is shown in figure-5

Generally, the parsing mechanism is language independent. However, once correct possible solutions are found, a post-processing operation follows, to check if the symbols, or upper-case letters are used correctly in the input word because these were eliminated in the preprocessing phase. This is especially important for parsing abbreviations and words containing special letters such as "Ahmet'in" or "prof.e" . This post-processing operation is language dependent and carried out by a helper class defined in all language implementations.

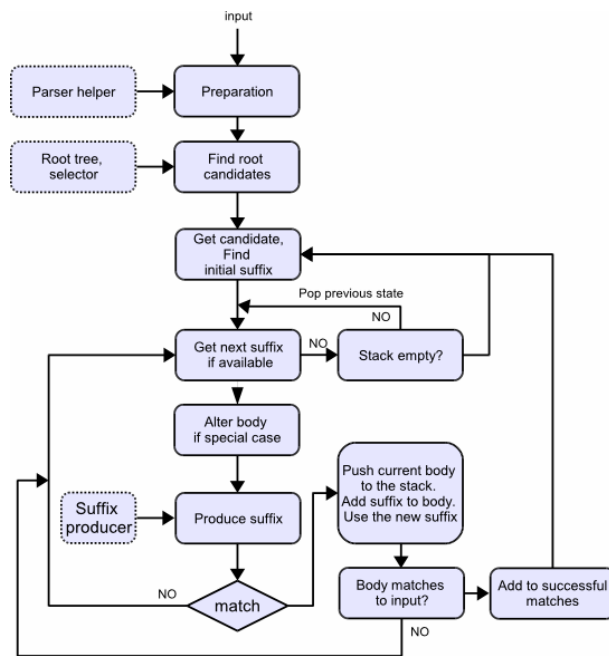


Figure-5

Turkic languages do not have prefixes. There are some uncommon exceptions, such as in Turkish emphasizing prefixes: “*mas+mavi*”, “*bem+beyaz*” . But those prefixes do not follow a certain pattern. However, some rare loan words contain prefixes. Currently, Zemberek does not parse prefixes.

Also, Zemberek does not parse some word construction suffixes; such as *-it* suffix; *yak*→*yakit* (to burn→fuel) or *kes*→*kesit* (to cut→cutaway). Instead, it contains the synthesized word inside the dictionary. Nevertheless, some word construction suffixes are applied in Zemberek by marking them as root words. One example is Turkish special “echo” root words.

For parser, other approaches could be tried. such as backwards suffix stripping [9], or using a letter driven suffix state machine but since current mechanism works well, those mechanisms has not been implemented.

4.1 Spell checker

Spell checker uses the standard Morphological parser for its operation. However, because spell checking does not need full parser results some performance operations are applied. First, standard parser does not try to find all the results, it returns early if a match is found.

4.2 Error Tolerated Parser and Word suggestion

Error tolerated parser operates similar to the standard parser as in finding the root candidates and exhaustive trial of creating and appending suffixes. In the current implementation, Error tolerant root selector uses Damerau-Levenshtein Edit Distance Algorithm, therefore errors with one character distance are detected in the system. Although not used by default, Jaro-Winkler String similarity algorithm can also be used. After finding possible positives from the error tolerated parser, Zemberek also checks if the input word is actually two words combined accidentally. Lastly Zemberek uses ASCII tolerated parser to see if the word was written without using Turkish specific characters. Then results are sorted using the root word usage frequency to have the best possible suggestions. However, a suggestion mechanism using a good word disambiguation scheme would be more accurate.

4.3 Word Construction and suffix content extraction

Zemberek has the ability to construct words. This is a simpler operation than Morphological parsing, it requires a root word object and a list of suffix objects. The system basically creates the suffixes and appends it after the formed word. similar to the parsing operation, special cases are applied. For example, for Turkmen language,

Root: {it:Noun} Suffixes {NOUN_AFFECTIONATE+NOUN_DATIVE}
produces "itjegize"

This mechanism is also used for generating a special list containing the suffix contents. The operation uses both the parser and the constructor. The Parser finds the root and the suffixes (but not the suffix words, only the type of suffixes) and then word constructor re-creates the word but this time storing the produced suffix contents. For example, for the Turkish word "koyunlara", parser produces:

Root: [koyun, NOUN]

Suffixes: [NOUN_PLURAL + NOUN_DATIVE]

We use the result in the constructor, it produces:
[koyun] + [lar, a]

5 Statistics

Zemberek provides basic statistical analysis tools to generate statistical information for letters, words, word types, roots, suffix patterns, word bi-grams, syllable counts, syllable bi-grams etc. Library has an extendable statistics reporting interface.

6 Usage and applications

Most high level operations of the library can be accessed using a special "façade" class called Zemberek. It is instantiated using the necessary *language information* class. For example for spell checking in Turkmen language using Zemberek, these two lines of code are enough.

```
Zemberek z = new Zemberek(new  
Turkmence())  
boolean result = z.denetle("pişige");
```

Zemberek uses Java language and platform for platform independence and speed. it is already used successfully not only in academic

researches[10][11] but also in real world applications such as OpenOffice.org [6] and Turkish Linux Distribution Pardus [7].

7 Performance and footprint

In the design of Zemberek, performance of the framework had a high priority because it was also intended to be used in real world applications. Particularly the most visible function, spell checking had to be very fast. However, creating a generic framework requires some compromises. Nevertheless, there are some special performance work applied to the code. such as:

- Usage of a binary root file reduced the initialization time of creating the root tree.
- Alphabet uses big character arrays for direct mapping of character - Turkish letter objects.
- Although its structure is quite complex compared to simple hash tables or list structures, usage of the special root tree provides much better performance than simple hash table based approaches.
- Early return and word cache in spell checking is used.
- Calculating the possible first letters of suffixes (Language implementer has to provide the method) and checking that data before actually creating and comparing a suffix content during standard parsing.
- Some suffixes have a statistically higher chance of following than others. Therefore suffix file optionally contains "priority suffix" information. Therefore some suffixes have priority in the parsing operation, The suffix statistics data was gathered using Zemberek's statistical information gathering tool.

Some performance optimizations are still possible but they are not yet implemented because of the added complexity.

Spell checking performance is tested with a Turkish novel consisting of 71.925 words. Average test result is 75.000 words per second. Compared to spell checking, full morphological analysis is a slower operation. For the same test case, Zemberek can analyze 12.000 words per second. Testing

environment: AMD Athlon 64 3000+ CPU, Windows XP OS, 512MB memory, using Java SE 6 Beta-2.

8 Other Turkic Languages

Currently Zemberek contains full implementations of Turkish and Turkmen languages. There is a skeleton Azeri Language implementation ready for entering real suffix and root word data. After that, Turkic languages using Latin alphabet such as, Uzbek and Tatar can follow. Kirgiz and Kazak languages use the Cyrillic alphabet. Since Cyrillic alphabet is similar to Latin alphabet in the sense that it has distinct symbols per sound, it is rather easy to implement.

Uyghur, Azeri Language spoken in Iran and Ottoman Turkish poses a bigger challenge because they use the Arabic alphabet. Because Arabic alphabet does not contain enough symbols for vowels, it requires special operations before processing it, such as converting words to a Latin alphabet.

9 Future Work

There are a number points to be improved in the Zemberek library.

- Other Turkic languages, especially ones using Cyrillic alphabet should be implemented. Tools necessary for implementing Arabic alphabet need to be investigated.
- A word sense disambiguation system is needed.
- Multi word root words and phrases need to be added.
- The lack of an open corpus and wordnet for Turkic Languages is preventing further and more advanced studies. We are planning to start an open and free corpus and wordnet project with a web based interface.

Conclusion

Zemberek is a very easy to use and open library for anyone who wants to study NLP for Turkic languages. Its MPL license allows anybody to use it in anyway he/she wants. Even though it provides basic NLP operations, we believe it is filling a big gap in the Turkic Language research area by

making it more accessible. However, Zemberek only scratches the surface of NLP and in time we hope to enter more challenging subjects. Zemberek project is hosted at <http://zemberek.dev.java.net>, and latest code can be found at: <http://code.google.com/p/zemberek/>

References

- [1]General information about Turkic Languages http://en.wikipedia.org/wiki/Turkic_languages
- [2]İlyas Cicekli, Türkçe ve Kırım Tatarcası Arasında Bir Çeviri Sistemi, in: Bilgisayar Destekli Dil Bilimi Çalıştayı Bildirileri, Ankara, Turkey, 2005, pp. 123-132 (in Turkish).
- [3] Oflazer Kemal, Two-level Description of Turkish Morphology, Literary and Linguistic Computing, vol. 9, No:2, 1994.
- [4]Atalay Nart B., Oflazer Kemal and Say Bilge, The Annotation in The Turkish Treebank, in Proceedings of the EACL Workshop on Linguistically Interpreted Corpora - LINC, April 13-14, Budapest, Hungary, 2003.
- [5]Oflazer Kemal, Say Bilge, Hakkani-Tür Dilek Zeynep, and Tür Gokhan. Building a Turkish Treebank, chapter in Building and Using Parsed Corpora, Anne Abeille Editor, Kluwer Academic Publishers, September 2003.
- [6]Official Turkish Open Office.org page: <http://www.openoffice.org.tr>
- [7]Pardus project page: <http://pardus.org.tr>
- [8]Allen James, Natural Language Processing (second edition), The Benjamin/Cummings Publishin Company, Inc., 1995.
- [9] Eryiğit, G. and Adali, E., 2004. An Affix Stripping Morphological Analyzer For Turkish, Proceedings of the IASTED International Conference on Artificial Intelligence and Applications, Innsbruck, Austria.
- [10] Guychmyrat Amanmyradov, Türkçe Türkmençe Bilgisayarlı çeviri sistemi, Karadeniz Teknik Üniversitesi FBE Bilgisayar Muendisligi anabilim dali Yüksek Lisans Tezi, Haziran 2006.
- [11]Gorkem Ozbek, Siddharth Jonathan, TURKALATOR, A Suite of Tools for Augmenting English-to-Turkish Statistical Machine Translation, Natural Language Processing Final Project. June 2006.
- [12]Özlem Çetinoğlu, A Prolog Based Natural Language Processing Infrastructure for Turkish. Ms Thesis, Bogazici University, 2001