Parse Trees of Arabic Sentences Using the Natural Language Toolkit

Maad Shatnawi and Boumediene Belkhouche
College of IT, UAE University, Al Ain
{shatnawi, b.belkhouche}@uaeu.ac.ae

Abstract

We develop a framework for using the Natural Language Toolkit (NLTK) to parse Quranic Arabic sentences. This framework supports the construction of a treebank for the Holy Quran. The proposed model succeeds in parsing different Quranic chapters (Suras) in addition to Modern Standard Arabic (MSA) sentences. The availability of such parser will be useful in various natural language processing applications such as machine translation, speech synthesis, and information retrieval.

Keywords: Arabic parser, Quranic sentences parsing, NLTK.

1. Introduction

Parsing has several applications including semantic disambiguation, machine translation, and speech synthesis. Traditional Arabic grammar is considered to be one of the historical origins of modern dependency grammar [19, 26]. There are several state-of-the-art Arabic parsers such as Bikel parser [4, 12, 20], Malt parser [16, 23, 24], Stanford parser [13, 18], and Attia’s rule-based parser for Modern Standard Arabic [1, 31]. These parsers require the availability of a treebank [14].

Like all Arabic language processing tasks, Arabic parsing faces many challenges including the unique features of the language, the exceptional degree of ambiguity in its writing system, the rich morphology, the highly-complex word formation process of roots and patterns, the length of sentences, the free order of words in the sentence, and the presence of the elliptic personal pronoun “al damiir al mustatir” [25, 30]. As a result, alternative parses for a particular sentence can suggest alternative meanings [8].

One of the significant challenges and prerequisite for parsing and syntactic analysis of a highly inflected language such as Arabic is the morphological segmentation [2, 15]. In the Penn English Treebank, verbal contraction such as “weren’t” are split into separate segments “were” and “n’t” with a different part-of-speech tag for each of them [3]. These segments form individual units in syntactic analysis and separate leaf nodes in a syntax tree.

Modern Standard Arabic (MSA) is the literary standard Arabic currently used across the Middle East and North Africa, and one of the official six languages of the United Nations. All printed books, newspapers, magazines, official and educational documents are written in MSA. Furthermore, MSA is used in news, documentary and scientific programs across different media and online portals.

Quranic Arabic is a unique form of Arabic used in the Quran and it is the direct ancestor language of MSA. Annotating the Quran faces an extra set of challenges compared to MSA due to the fact that the text is over 1,400 years old, and the Quranic script is more varied than modern Arabic in terms of orthography, spelling and inflection. The same word is spelled in different ways in different chapters. However, the Quran is fully diacritized which reduces its ambiguity [8].

With respect to morphological segmentation, 54% of the Quran’s 77,430 words require segmentation, resulting in 127,806 segments. A typical word in the Quran consists of multiple segments combined into a single whitespace-delimited word form. For example, “اذﻛﺮوﻧﻲ” which means “so remember me” should be segmented to four segments; “اذﻛﺮ”, “و”, “و”, “و”, and “و”, where each segment represents an individual syntactic unit as follows: conjunction prefix, verb, subject pronoun, and object pronoun, respectively.

The computational analysis of the Quran remains an intriguing but unexplored field since little and isolated efforts have been done in this field [6]. For the young learner, this may be not just a challenging requirement, but a deterrent.

The rest of this paper is organized as follows. The next section provides a brief overview of the existing Arabic treebanks for both MSA and Quranic sentences. Section 3 presents our methodology to parse Arabic sentences through NLTK. Analysis of our results is discussed in section 4. Concluding remarks are presented in Section 5.

2. Arabic Treebanks

A treebank is a collection of manually checked syntactic analyses of sentences. Treebanks constitute important resources for building and evaluating parsers and are used in several applications such as tokenization, diacritization, part-of-speech (POS) tagging, morphological disambiguation, chunk parsing, and semantic role labeling [14].
The three well-known Arabic treebanks for MSA are the Penn Arabic Treebank (PATB) [21, 22], the Prague Arabic Dependency Treebank (PADT) [17, 28, 29], and the Columbia Arabic Treebank (CATb) [15, 16].

The first Arabic treebank project is PATB which was started in 2001 at the Linguistic Data Consortium (LDC) and the University of Pennsylvania. The creation of PATB is considered a great achievement for Arabic Natural Language Processing. It has been a great research resource in morphological analysis, tokenization, POS tagging, disambiguation, and parsing. In addition to that, it has been used as a foundation in developing other treebanks [14].

The parsing tree in PATB for the Sentence ‘Fifty thousand tourists visited Lebanon and Syria last September’ (خمسون ألف سائح زاروا لبنان وسوريا في أيلول الماضي) is shown in Figure 1. The parsing trees of the same sentence in PADT and CATb are shown in Figures 2 and 3, respectively.

The Quranic Arabic Dependency Treebank (QADT) [9] has been developed and implemented in Java at the University of Leeds, UK, as part of the Quran Corpus project [7]. It works as a treebank for Quranic sentences by providing a deep computational linguistic model based on historical traditional Arabic grammar (اعراب القرآن الكريم). The treebank is freely available under an open source license. QADT provides two levels of analysis: morphological annotation and syntactic representation. The morphological segmentation has been applied to all the 77,430 words in the Quran. The treebank also represents Quranic syntax through dependency graphs as shown in Figure 4 [9, 10]. QADT is applied to each individual Quranic verses (or part of the verse) even if that verse does not form a complete sentence unless it is joined with its neighbor verse. In addition to QADT, there are very few works done in the field of morphological analysis of Quran such as [6].

Othman et al. [25] developed a rule-based chart parser for Analyzing MSA sentences with the use of lexical semantic features to disambiguate the sentence structure and the integration with a morphological analyzer.
3. Methodology

We apply the top-down (Recursive Descent) parsing algorithm to parse Arabic sentences through the Natural Language Toolkit (NLTK) [5]. NLTK is an open source suite of libraries and programs which can be integrated within the Python environment and then used to perform different statistical and rule-based natural language processing tasks such as POS tagging and parsing. Starting from an initial symbol S, the Recursive Descent parser applies the grammar rules forward until all the words of the given sentence are consumed. In this work, the recursive descent parser is tested and evaluated in building parse trees of Arabic sentences in general and Quranic sentences in particular. As morphological segmentation is a prerequisite for parsing, the segmentation is now done manually. Work is underway to integrate a morphological analyzer with the parser.

Our first example demonstrates how the NLTK recursive descent parser is used to parse a MSA sentence. We use the same sentence that was parsed by the three previously mentioned Arabic parsers. The sentence is ‘Fifty thousand tourists visited Lebanon and Syria last September’ (خمسون ألف سائح زاروا لبنان وسوريا في أيلول الماضي). The sequence of steps to parse a text is as follows:

1. Divide the text into independent sentences.
2. Create a Python file for each sentence.
3. Perform morphological analysis.
4. Build the lexicon entries for each word in the sentence. Each word is assigned a tag:
   - lex1 = CFGProduction(NN, ['خمسون'])
   - lex2 = CFGProduction(NN, ['ألف'])
   - lex3 = CFGProduction(NN, ['سائح'])
   - lex4 = CFGProduction(VT, ['زار'])
   - lex5 = CFGProduction(Pmn, ['وا'])
   - lex6 = CFGProduction(NN, ['لبنان'])
   - lex7 = CFGProduction(CJ, ['و'])
   - lex8 = CFGProduction(NN, ['سوريا'])
   - lex9 = CFGProduction(P, ['في'])
   - lex10 = CFGProduction(NN, ['أيلول'])
   - lex11 = CFGProduction(NN, ['ماضي'])
5. Build the grammar rules:
   - S_prod3 = CFGProduction(S, [NS])
   - VS_inv_prod1 = CFGProduction(VS, [VT, NP, CP, PP])
   - NS_inv_prod1 = CFGProduction(NS, [Starter, Predicate])
   - Starter_inv_prod1 = CFGProduction([NP])
   - Predicate_inv_prod5 = CFGProduction(Predicate, [VS])
   - CP_inv_prod3 = CFGProduction(CP, [NP, CJ, NP])
   - PP_prod1 = CFGProduction(P, [NP])
   - NP_prod1 = CFGProduction(NP, [PP, NN])
   - NP_prod5 = CFGProduction(NP, [NP, PP, NN])


The above rules can also be represented as follows:

- جملة نسبية → جملة فعلية → جملة مجزورة
- شبه جملة جار ومحجور → جملة نسبية
- جملة فعلية → شبه جملة جار ومحجور
- ضمير →
The generated parse tree of the verse is shown in Figure 6. Figure 7 presents the parse tree of Suratul Falaq (Chapter 113 of the Holy Quran) as a whole. The grammar rules of the last four Suras are presented in Appendix 1.

4. Analysis of the Results

The recursive descent parser has three main limitations [27]. The first limitation is that left-recursive productions such as "NP → NP PP" cause the parser to go in an infinite loop. Secondly, the parser blindly tries all words (lexical entries) and grammar rules that are irrelevant to the input sentence which considerably increases its time complexity. Thirdly, the algorithm does not store the successfully parsed parts which, therefore, need to be rebuilt again later in the backtracking process.
For example, if the NP part in “VS \(\rightarrow\) V  NP” is successfully parsed, the backtracking to “VS \(\rightarrow\) V  NP PP” requires reparsing NP part again. As a result, the parsing processor repeats much of its work.

As the coverage of the grammar increases and the length of the input sentences grows, the number of parse trees dramatically grows. In the worst case, the recursive descent parser has an exponential time complexity as well as an exponential memory space for ambiguous context-free grammar [11]. However, for small texts like Suras 115, 114 and 115, the statistical relationship between the number of words (lexical tokens) in the sentence and the total number of nodes in the parse tree is linear. From parse tree data we collected, it is expressed as:

\[ n = 3.36w + 0.2 \]

where \( n \) is the total number of nodes in the parse tree and \( w \) is the number of words in the sentence. This expression indicates that, at least, for the examples, the complexity of the parse trees grows linearly.

5. Conclusion

We presented an approach to generate parse trees of Arabic sentences in general and Quranic sentences in particular, using the Natural Language Toolkit (NLTK). The process we defined consists of building a lexicon, a context-free grammar and invoking the NLTK recursive-descent parser. The generated parse trees can be viewed as components of a tree bank, similar to the tree banks mentioned earlier.

Our work can be extended in several directions. The integration of the parser with a morphological analyzer would further automate the process. Using diacratized sentences can dramatically decrease the semantic ambiguity and reduce the complexity of the grammar rules. Other types of parsers such as bottom-up can be also examined on Arabic sentences in a future work.

References

Angeles, CA, USA, Association for Computational Linguistics, June 2010.


Appendix A

Each of the tables shows the grammar rules for a given Sura. The combined grammar results from putting together the three grammars.

Combined Grammar Rules

\[ S \_prod 1 = \text{CFGProduction}(S, [VS, CJ, VS]) \]
\[ S \_prod 2 = \text{CFGProduction}(S, [VS]) \quad \#VS: \]
Verbal Sentence
\[ VS \_inv \_prod 1 = \text{CFGProduction}(VS, [VT]) \quad \#VT: \]
Verbal Sentence
\[ VS \_inv \_prod 1 = \text{CFGProduction}(VS, [VT, NN, NN, PP]) \]
\[ VS \_inv \_prod 2 = \text{CFGProduction}(VS, [VT, PP]) \]
\[ VS \_inv \_prod 3 = \text{CFGProduction}(VS, [VT, PP, PP]) \]
\[ VS \_inv \_prod 4 = \text{CFGProduction}(VS, [VT, PP, CP]) \quad \# CP: \]
Conjunctural Phrase
\[ VS \_inv \_prod 5 = \text{CFGProduction}(VS, [VT, VS]) \]
\[ VS \_inv \_prod 6 = \text{CFGProduction}(VS, [VT, PP, CP]) \]
\[ VS \_inv \_prod 7 = \text{CFGProduction}(VS, [VT, PP, CP]) \]
\[ VS \_inv \_prod 8 = \text{CFGProduction}(VS, [CP]) \]
\[ VS \_inv \_prod 9 = \text{CFGProduction}(VS, [VT, PP, PP]) \]
\[ VS \_inv \_prod 10 = \text{CFGProduction}(VS, [NG, VS]) \quad \#NG: \]
Negation Particle
\[ VS \_inv \_prod 11 = \text{CFGProduction}(VS, [VT, PP, CP]) \]
\[ VS \_inv \_prod 12 = \text{CFGProduction}(VS, [VT, PP, CP]) \]
\[ VS \_inv \_prod 13 = \text{CFGProduction}(VS, [VT, VS]) \quad \#VS: \]
Nominal Sentence
\[ VS \_inv \_prod 14 = \text{CFGProduction}(VS, [VT, VS]) \quad \#V: \]
Incomplete Verb
\[ NS \_inv \_prod 1 = \text{CFGProduction}(NS, [Starter, Predicate]) \quad \#Starter: \]
Predicate:
\[ NS \_inv \_prod 2 = \text{CFGProduction}(NS, [Starter, Predicate, Comma, Predicate]) \]
\[ NS \_inv \_prod 3 = \text{CFGProduction}(NS, [Advanced Predicate, Starter]) \quad \#Advanced Predicate:
\[ NS \_inv \_prod 4 = \text{CFGProduction}(NS, [Starter, Predicate, Comma, Predicate, Comma, Predicate]) \]
\[ Starter \_inv \_prod 1 = \text{CFGProduction}(Starter, [NP]) \]
\[ Predicate \_inv \_prod 1 = \text{CFGProduction}(Predicate, [NP]) \]
\[ Predicate \_inv \_prod 2 = \text{CFGProduction}(Predicate, [NS]) \]
\[ Predicate \_inv \_prod 5 = \text{CFGProduction}(Predicate, [VS]) \]
\[ Predicate \_inv \_prod 6 = \text{CFGProduction}(Predicate, [VS, CJ, VS, VS]) \]
\[ Advanced \_Predicate \_inv \_prod = \text{CFGProduction}(Advanced Predicate, [PP, NP]) \]
\[ CP \_inv \_prod 2 = \text{CFGProduction}(CP, [PP, CJ, CP]) \quad \#CP: \]
Conjunctural Phrase
\[ CP \_inv \_prod 1 = \text{CFGProduction}(CP, [PP, CJ, PP]) \]
\[ CP \_inv \_prod 3 = \text{CFGProduction}(CP, [NP, CJ, NP]) \]
\[ PP \_prod 1 = \text{CFGProduction}(PP, [P, PP]) \quad \#P: \]
Preposition
\[ NP \_prod 1 = \text{CFGProduction}(NP, [PP, PP]) \quad \#AdP: \]
Additional Phrase
\[ NP \_prod 2 = \text{CFGProduction}(NP, [PP, PP]) \quad \#CnP: \]
Conditional Phrase
\[ NP \_prod 3 = \text{CFGProduction}(NP, [AdP]) \]
\[ NP \_prod 4 = \text{CFGProduction}(NP, [AdP]) \]
\[ NP \_prod 5 = \text{CFGProduction}(NP, [PP, PP]) \quad \#PP: \]
Pronoun
\[ NP \_prod 6 = \text{CFGProduction}(NP, [RelP]) \]
\[ NP \_prod 7 = \text{CFGProduction}(NP, [AdP, Comma, AdP, Comma, AdP]) \]

<table>
<thead>
<tr>
<th>G1 (Al Nas)</th>
<th>10 rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \rightarrow VS</td>
<td></td>
</tr>
<tr>
<td>VS \rightarrow VT, VS</td>
<td></td>
</tr>
<tr>
<td>VS \rightarrow VT, PP, PP</td>
<td></td>
</tr>
<tr>
<td>PP \rightarrow P, NP</td>
<td></td>
</tr>
<tr>
<td>NP \rightarrow AdP, Comma, AdP, Comma, AdP</td>
<td></td>
</tr>
<tr>
<td>NP \rightarrow NN</td>
<td></td>
</tr>
<tr>
<td>AdP \rightarrow NN, NP</td>
<td></td>
</tr>
<tr>
<td>AdP \rightarrow NN, RelP</td>
<td></td>
</tr>
<tr>
<td>RelP \rightarrow Rel, VS</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G2 (Al Falaq)</th>
<th>15 rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \rightarrow VS</td>
<td></td>
</tr>
<tr>
<td>VS \rightarrow VT, VS</td>
<td></td>
</tr>
<tr>
<td>VS \rightarrow VT, PP, CP</td>
<td></td>
</tr>
<tr>
<td>PP \rightarrow VT, PP, PP</td>
<td></td>
</tr>
<tr>
<td>NP \rightarrow AdP</td>
<td></td>
</tr>
<tr>
<td>NP \rightarrow NN</td>
<td></td>
</tr>
<tr>
<td>NP \rightarrow NN, CNP</td>
<td></td>
</tr>
<tr>
<td>NP \rightarrow NN, PP</td>
<td></td>
</tr>
<tr>
<td>AdP \rightarrow NN, NP</td>
<td></td>
</tr>
<tr>
<td>AdP \rightarrow NN, RelP</td>
<td></td>
</tr>
<tr>
<td>RelP \rightarrow Rel, VS</td>
<td></td>
</tr>
<tr>
<td>CNP \rightarrow CN, VS</td>
<td></td>
</tr>
<tr>
<td>CP \rightarrow PP, CJ, CP</td>
<td></td>
</tr>
<tr>
<td>CP \rightarrow PP, CJ, PP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G3 (Al Ikhlas)</th>
<th>14 rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \rightarrow VS</td>
<td></td>
</tr>
<tr>
<td>VS \rightarrow VT</td>
<td></td>
</tr>
<tr>
<td>VS \rightarrow VT, NS</td>
<td></td>
</tr>
<tr>
<td>VS \rightarrow NG, VS</td>
<td></td>
</tr>
<tr>
<td>VS \rightarrow VI, NS</td>
<td></td>
</tr>
<tr>
<td>NS \rightarrow Starter, Predicate</td>
<td></td>
</tr>
<tr>
<td>NS \rightarrow Starter, Predicate, Comma, Predicate</td>
<td></td>
</tr>
<tr>
<td>NS \rightarrow Starter, Predicate, Comma, Predicate, Comma, Predicate</td>
<td></td>
</tr>
<tr>
<td>NS \rightarrow Advanced Predicate, Starter</td>
<td></td>
</tr>
<tr>
<td>Starter \rightarrow NP</td>
<td></td>
</tr>
<tr>
<td>Predicate \rightarrow NP</td>
<td></td>
</tr>
<tr>
<td>Predicate \rightarrow NS</td>
<td></td>
</tr>
<tr>
<td>Advanced Predicate \rightarrow PP, NP</td>
<td></td>
</tr>
<tr>
<td>NP \rightarrow Prn</td>
<td></td>
</tr>
<tr>
<td>NP \rightarrow NN</td>
<td></td>
</tr>
</tbody>
</table>