Application Project 2
Building a parser

Dinno Koluh
Boğaziçi University
Department of Computer Engineering
Natural Language Processing
dinno.koluh@studio.unibo.it

Abstract

This paper describes a system which takes as an input a user-defined context-free grammar (CFG) and parses a sentence using the implementation of the CKY parser for English. The goal of the parser is to be able to parse declarative, imperative and question sentences using a suitable CFG. It is assumed that we have the SVO word order in the English language.

Satisfactory results are reported for a variety of tests which are available together with the source code here.

1 Introduction

Knowing the meaning of a sentence (i.e. its syntactic analysis) is one of the crucial tasks in NLP. The groundwork for that is *syntactical parsing*. That is the task which we are going to engage in this paper.

The first part of any parsing are the PoS (Part of Speech) tags which are needed to classify specific words in the sentence. Those tags will be then used as the bridge between the grammar and the actual words in the sentence when obtaining the final parse.

To actually parse a sentence we need a set of rules by which we should analyze the sentence. Those rules are usually referred as the *grammar* and the grammar model which we are going to use in this paper is the *Context-Free Grammar* (CFG).

The parser model which we are going to use is the CKY (Cocke-Kasami-Younger) parser which requires a normalized CFG which is the CNF (Chomsky Normal Form). This detail has to be implemented as well.

To sum up, the things we will need to implement are:

1. Getting the PoS tags
2. Design of the CFG
3. Design of the CKY parser

In the coming sections we are going to look at these points. At last, let us look at the whole parsing pipeline. Assuming that we have an input text, firstly we need to use sentence splitting and then tokenize each sentence individually. After tokenization, we will get the PoS tags, but all the tokens need to be considered in this process. To obtain the PoS tags context might be important. The next part is the design of the grammar. Depending on our specific needs and the language in question an appropriate grammar has to be designed in the CFG form. And as the last part we have is the actual parsing, which is done according to the CKY algorithm and the final result is the parse of the sentence which is going to be shown in the bracket and tree notations. As a side note, we might add that we could get as an output more than one parse tree as there could be more than one interpretation of the meaning of a sentence.

2 Getting the PoS tags

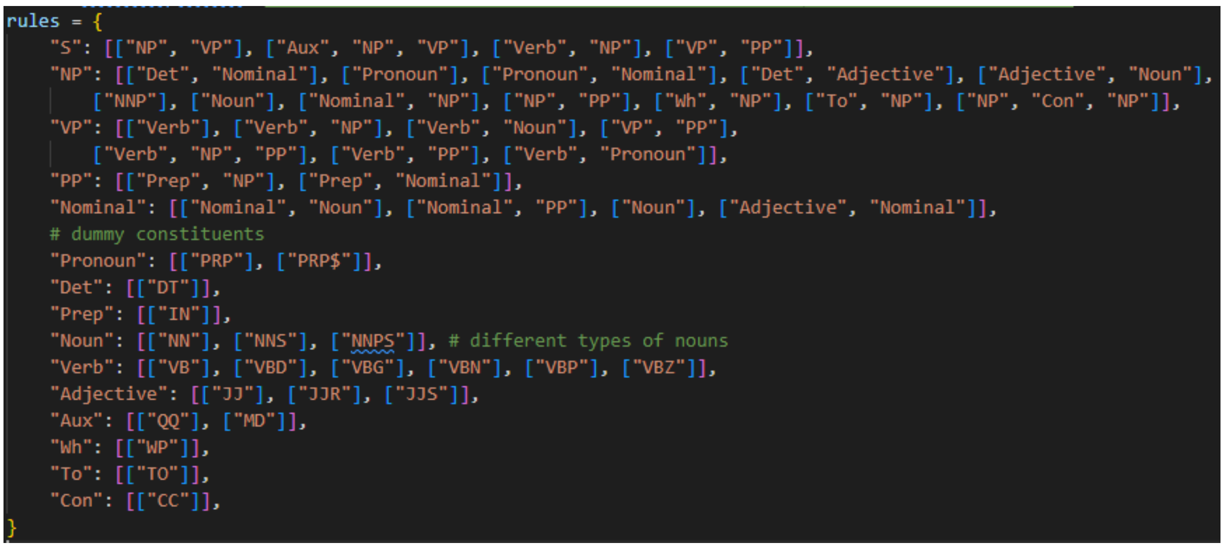
The PoS tagset which we are going to use are the Penn Treebank tags [1]. It is a tagset which contains 36 tags. This system is developed in Python, so the easiest way of getting the PoS tags, which can be immediately embedded inside Python, is using the extensive *nltk* [2] library for NLP in Python. This library has a broad set of corpora labelled with different tagsets. The one chosen for this system is the Brown corpus which is labelled with the Penn Treebank tagset. In this way we have a direct access to the PoS tags by injecting the tokenized sentence in the nltk tagger function. It is good to outline again that this tagging function takes in an array of tokens as context is important when tagging a sentence.

3 Designing the grammar

The grammar model used for this system is the CFG model. The grammar is built so that is handles simple types of sentences without sub-clauses but is able to handle declarative, imperative and question sentences. The grammar was partially built from lecture slides and partially through trail and error on different examples.

For the constituent rules of the CFG two different types of tags were used; the ones that are not part of the Penn Treebank (e.g. S, NP, VP, etc.) which are also the non-terminal symbols and those which are part of the treebank (e.g. DT, IN, PRP etc.) which are also the terminal symbols. The reason for such a choice of the structure of the grammar is so that we don't need to directly build the lexical rules (e.g. directly encoding that Det $\rightarrow$ *the* which would've needed to be done manually). To avoid this, a dummy constituent is created (e.g. Noun) and constituent rules can be defined which directly link the dummy constituent to the PoS tags of the Penn Treebank (e.g. "Noun": [["NN"], ["NNS"], ["NNP"], ["NNPS"]]). With this in mind, lexical rules don't need to be defined because the tokens will be directly linked to the non-terminals via the dummy constituents which are linked via the PoS tags.

The actual grammar rules were implemented as a Python dictionary where the keys of the dictionary are the LHS of the rule, and for each key there can be multiple RHS rules which is just a more compatible way of encoding the grammar. In figure 1 the CFG used for this system is shown.



```python
rules = {
    "S": [["NP", "VP"], ["Aux", "NP", "VP"], ["Verb", "NP"], ["VP", "PP"]],
    "NP": [["Det", "Nominal"], ["Pronoun"], ["Pronoun", "Nominal"], ["Det", "Adjective"], ["Adjective", "Noun"],
        ["NNP"], ["Noun"], ["Nominal", "NP"], ["NP", "PP"], ["Wh", "NP"], ["To", "NP"], ["NP", "Con", "NP"]],
    "VP": [["Verb"], ["Verb", "NP"], ["Verb", "Noun"], ["VP", "PP"],
        ["Verb", "NP", "PP"], ["Verb", "PP"], ["Verb", "Pronoun"]],
    "PP": [["Prep", "NP"], ["Prep", "Nominal"]],
    "Nominal": [["Nominal", "Noun"], ["Nominal", "PP"], ["Noun"], ["Adjective", "Nominal"]],
    # dummy constituents
    "Pronoun": [["PRP"], ["PRP$"]],
    "Det": [["DT"]],
    "Prep": [["IN"]],
    "Noun": [["NN"], ["NNS"], ["NNPS"]], # different types of nouns
    "Verb": [["VB"], ["VBD"], ["VBG"], ["VBN"], ["VBP"], ["VBZ"]],
    "Adjective": [["JJ"], ["JJR"], ["JJS"]],
    "Aux": [["QQ"], ["MD"]],
    "Wh": [["WP"]],
    "To": [["TO"]],
    "Con": [["CC"]],
}
```

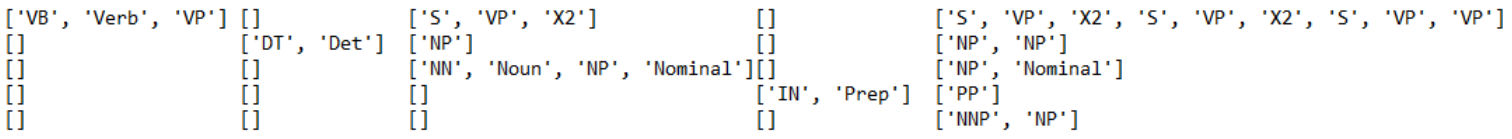
Figure 1: CFG rules as a Python dictionary

4 Designing the parser

The choice of the parser for this system was the CKY parser. As far as the parse table for the CKY parser is concerned, it was constructed in the usual way as described by the algorithm. The more interesting part is the way the links between the cell entries were encoded and how the bracket and tree notation was obtained.

The best way to show this is on an actual example: *"Get a flight through Huston"*.
We get the following PoS tagging: [('Get', 'VB'), ('the', 'DT'), ('flight', 'NN'), ('through', 'IN'), ('Huston', 'NNP')].

The parse table gotten from the CKY parser is shown in figure 2.



```
['VB', 'Verb', 'VP'] []             ['S', 'VP', 'X2']               []             ['S', 'VP', 'X2', 'S', 'VP', 'X2', 'S', 'VP', 'VP']
[]                   ['DT', 'Det'] ['NP']                          []             ['NP', 'NP']
[]                   []             ['NN', 'Noun', 'NP', 'Nominal'][]             ['NP', 'Nominal']
[]                   []             []                              ['IN', 'Prep'] ['PP']
[]                   []             []                              []             ['NNP', 'NP']
```

Figure 2: CKY parse table

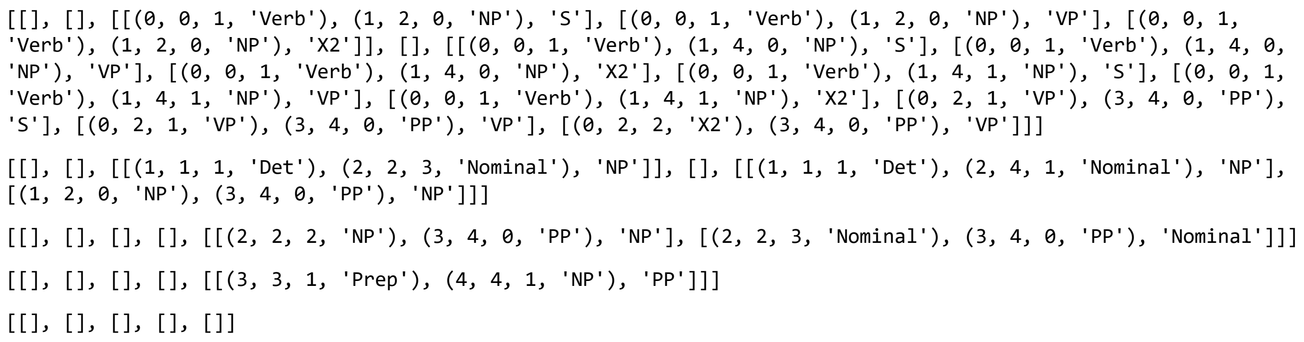
To get the diagonal entries of the table we take the PoS tags assigned (i.e. VB, DT, etc.) and look them up in the dummy constituents in the grammar. Then we propagate to all the non-terminals with these dummy rules to obtain all the possible tags for a token (i.e. flight $\rightarrow$ NN, Noun, NP, Nominal).

To obtain the off-diagonal entries we need to take the Cartesian product of the child nodes (the left and bottom cells) which were used to make that particular cell and check if there is a RHS rule which is among the elements of the Cartesian product. If we look at the cell $(0, 2)$, it was made from the cells $(0, 0) \times (1, 2)$ and $(0, 1) \times (2, 2)$. The second pair or cells doesn't yield anything but for the first pair the Cartesian product gives us:

$$['VB', 'Verb', 'VP'] \times ['NP'] = [['VB', 'NP'], ['Verb', 'NP'], ['VP', 'NP']]$$

And from these, the rule ['Verb', 'NP'] yields 'S', 'VP' and 'X2' (which is part of the CNF but not included in the initial grammar).

For the linking between the cells a new link table was constructed. It has entries only above the main diagonal as only there we have node parents. The link table is show in figure 3.



```
[[], [], [[(0, 0, 1, 'Verb'), (1, 2, 0, 'NP'), 'S'], [(0, 0, 1, 'Verb'), (1, 2, 0, 'NP'), 'VP'], [(0, 0, 1,
'Verb'), (1, 2, 0, 'NP'), 'X2']], [], [[(0, 0, 1, 'Verb'), (1, 4, 0, 'NP'), 'S'], [(0, 0, 1, 'Verb'), (1, 4, 0,
'NP'), 'VP'], [(0, 0, 1, 'Verb'), (1, 4, 0, 'NP'), 'X2'], [(0, 0, 1, 'Verb'), (1, 4, 1, 'NP'), 'S'], [(0, 0, 1,
'Verb'), (1, 4, 1, 'NP'), 'VP'], [(0, 0, 1, 'Verb'), (1, 4, 1, 'NP'), 'X2'], [(0, 2, 1, 'VP'), (3, 4, 0, 'PP'),
'S'], [(0, 2, 1, 'VP'), (3, 4, 0, 'PP'), 'VP'], [(0, 2, 2, 'X2'), (3, 4, 0, 'PP'), 'VP']]]

[[], [], [[(1, 1, 1, 'Det'), (2, 2, 3, 'Nominal'), 'NP']], [], [[(1, 1, 1, 'Det'), (2, 4, 1, 'Nominal'), 'NP'],
[(1, 2, 0, 'NP'), (3, 4, 0, 'PP'), 'NP']]]

[[], [], [], [], [[(2, 2, 2, 'NP'), (3, 4, 0, 'PP'), 'NP'], [(2, 2, 3, 'Nominal'), (3, 4, 0, 'PP'), 'Nominal']]]

[[], [], [], [], [[(3, 3, 1, 'Prep'), (4, 4, 1, 'NP'), 'PP']]]

[[], [], [], [], []]
```

Figure 3: The link table

Each cell of the link table contains the links of how the constituents of that cells were obtained in the CKY parse table. If we again look at the example of the cell entry $(0, 2)$ from the parse table, the link table in that cell contains an array of length 3 for the three non-terminals that makeup that cell. That array is:

[[(0, 0, 1, 'Verb'), (1, 2, 0, 'NP'), 'S'], [(0, 0, 1, 'Verb'), (1, 2, 0, 'NP'), 'VP'], [(0, 0, 1, 'Verb'), (1, 2, 0, 'NP'), 'X2']]

Let us look at the first element of this array:

[(0, 0, 1, 'Verb'), (1, 2, 0, 'NP'), 'S']

The first two entries are the child nodes and the third entry is the symbol of the parent node which is 'S'. A child node is made out of 4 entries where:

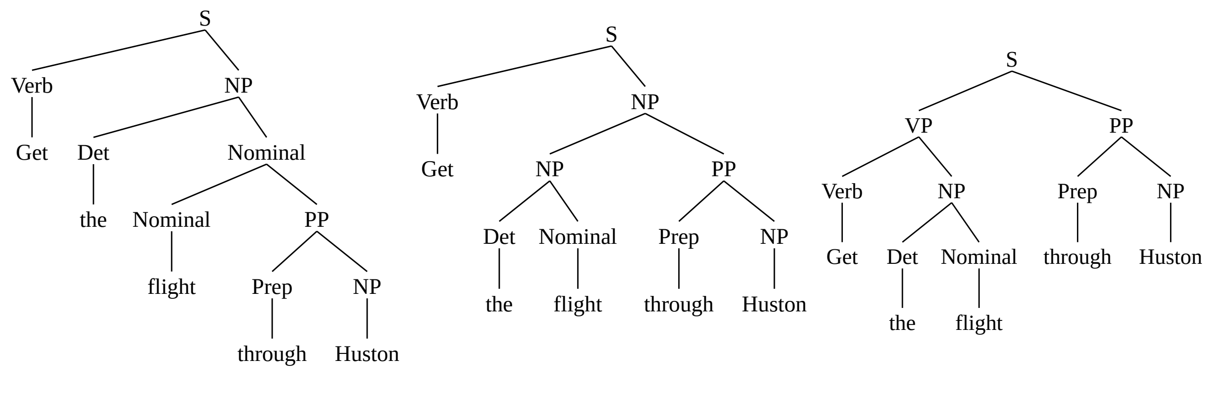
- the first two entries are the cell indices where this node is found (e.g. the symbol 'Verb' that made 'S' is found at $(0, 0)$ and the symbol 'NP' that made 'S' is found at $(1, 2)$)
- the third entry is the ordinal number of that particular symbol (e.g. the symbol 'Verb' has ordinal number 1 as it is at position 1 (we count from 0) in the cell $(0, 0)$ in the parse table (['VB', 'Verb', 'VP'])). This is done as it might happen that there are more identical symbols in a cell and it is important to know which one is the child symbol.
- the fourth entry is just the symbol of the child (e.g. the symbol of the first child is 'Verb' and of the second 'NP')

In this way we get a fully linked tree to the root symbol (which is 'S'). A side note to mention is that for the leaves of the tree which are at the diagonal entries, we may look back to the parse table. This is going to be useful when displaying the bracket and tree notation.

The bracket and tree notation are done recursively using the starting symbol 'S' as the root and from there the children nodes are called using the link table until the diagonal elements are reached. When that happens, the leaves which are the diagonal elements are called from the parse table and they are the condition for ending the recursion. The obtained result for the above example is:

- Bracket notation:
  [S [Verb 'Get'] [NP [Det 'the'] [Nominal [Nominal 'flight'] [PP [Prep 'through'] [NP 'Huston']]]]]
  [S [Verb 'Get'] [NP [NP [Det 'the'] [Nominal 'flight']] [PP [Prep 'through'] [NP 'Huston']]]]
  [S [VP [Verb 'Get'] [NP [Det 'the'] [Nominal 'flight']]] [PP [Prep 'through'] [NP 'Huston']]]

- Tree notation:



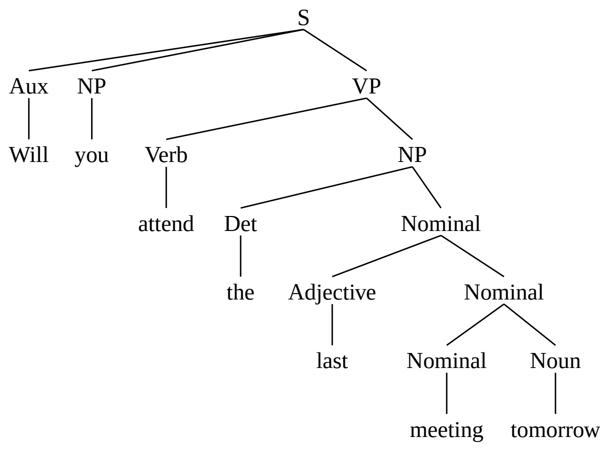
One last thing to address here is the conversion to and from CNF. There is a specific function that converts from CFG to CNF notation by iteratively dividing rules whose RHS is longer than 2 until it reaches length 2. For the conversion back to CFG, it is implicitly implemented when making the bracket and tree notation such that parents who might have more than two children get the number of children extended back. This is done by explicitly checking if the child symbol starts with an 'X' as all the additional CNF symbols by default are of the form 'X#' where '#' is a number index used to distinguish between the corresponding symbols.

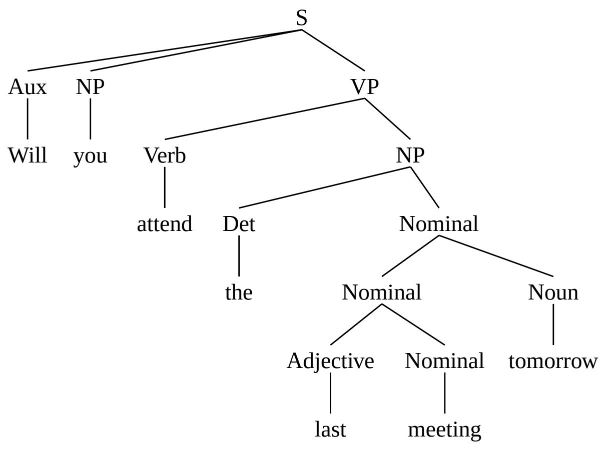
5 Results

For the sake of showing parse trees with nodes which have more than two children let us look at the following example: *”Will you attend the last meeting tomorrow?”*.

The PoS tags are given as:
[(’Will’, ’MD’), (’you’, ’PRP’), (’attend’, ’VB’), (’the’, ’DT’), (’last’, ’JJ’), (’meeting’, ’NN’), (’tomorrow’, ’NN’)].

We get two parses as the result. In the bracket notation we have:
1. [S [Aux ’Will’] [NP ’you’] [VP [Verb ’attend’] [NP [Det ’the’] [Nominal [Adjective ’last’] [Nominal [Nominal ’meeting’] [Noun ’tomorrow’]]]]]]
2. [S [Aux ’Will’] [NP ’you’] [VP [Verb ’attend’] [NP [Det ’the’] [Nominal [Nominal [Adjective ’last’] [Nominal ’meeting’]] [Noun ’tomorrow’]]]]]





6 Agreement phenomena

On of the given tasks to this system is to also recognize several types of agreement restrictions. A few examples that were given of such cases are:

- I buys a gift for my friend.
- I read a historical novels.
- I went the school.

The idea would be to place a filter before the parsing to detect such sentences. The good thing about the Penn Treebank tags is that it distinguishes between verbs which are 3^{rd} person singular present (VBZ) and non-3^{rd} person singular present (VBP) but pronouns are marked just as PRP so we would need to check explicitly if a pronoun is 3^{rd} person singular to make the agreement with the verb.

I also want to address the third example given above. We are obviously missing the *to* (TO) token (I went to the school). But if we generalize this sentence to ”I [VBD] the [NN]” we can find a past tense verb and a singular noun which could actually fit this generalization which is wrong for the example given above. An example which would fit such a generalization would be: ”I rode the bike” so we would actually need to know the intrinsic meaning of the words we are parsing to be able to cover all the cases.

(*NOTE: Due to my time constraints I was not able to implement the agreement phenomena problem for the parser!)

7 Discussion and Conclusion

In this paper a system for parsing based on CFG grammar was developed. The chosen parser was the CKY parser. Alongside the CKY parser which outputs only a parse table, another mechanism for getting the links between the nodes of the parse tree was developed. For a better visual representation the bracket and tree notation output was implemented. Using a limited CFG satisfactory results were obtained. Using a more sophisticated CFG a wider range of examples could be performed. At last, a satisfactory system was build that has promising outcomes for future development.

References

[1] Penn Treebank PoS tags.
URL: `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`

[2] Python NLTK library documentation.
URL: `https://www.nltk.org/`