This paper introduces the Graphical Grammar Studio (GGS) software and describes a grammar that was built with it for deep Romanian noun phrase chunking. GGS is an open source java tool for NLP tasks which, unlike most grammar languages and matching tools, allows the graphical design of grammars, somehow resembling the old recursive transition networks. Such a grammar is usually used for finding sequences of words which respect certain conditions. GGS grammars can also be used to annotate the matched sequences. With GGS one can create complex grammars which can even work as standalone NLP tools. The second part of this paper presents a complex grammar which recursively detects and annotates noun phrase chunks for Romanian text.

1. Introduction

Graphical Grammar Studio (GGS) is a tool developed by the author and published as an open source project on SourceForge1. GGS offers the tools for creating and applying Recursive Transitional Networks (RTN). But unlike classic RTNs which consume one character at a time, GGS consumes one token at a time. For this reason GGS is a tool for finding and annotating sequences of tokens and it is somewhat oriented towards syntactic layer of NLP.

Like Finite State Automata (FSA), RTNs are recognizers (acceptors) of sentences generated by grammars. But, additionally, their states can be structured into subnetworks (or graphs of nodes), with the possibility to create jumps from one subnetwork to another. Recursive jumps can be created. Each graph / subnetwork has an initial and a final state and each RTN has a main graph. W. A. Woods claims that by adding a recursive mechanism to a finite state model, parsing can be achieved much more efficiently (Woods 1970).

A GGS grammar represents a RTN in a fashion which is more convenient for NLP tasks. Unlike classic RTNs whose arcs are labeled with terminal symbols (acceptance conditions) and their nodes with arbitrary symbols, GGS grammars only have their nodes labeled with acceptance conditions. The arcs have no labels in GGS representation. But one can easily consider the condition of a GGS arc to be the condition of the state that it points to.

GGS grammars can generate output while matching an input text, thus creating annotations. A similar behavior in literature can be found in models called Finite State

1 http://sourceforge.net/projects/ggs/
Transducers, which are classical Finite State Automata but which also generate an output while consuming an input. A GGS grammar could be thus considered a Recursive Transition Transducer. In one of his papers devoted to state machine techniques, Emmanuel Roche says: "Finite-state transducers should appeal to the linguist looking for precise and natural description of complex syntactic structures [...] The parsing programs derived from this approach are both simple, precise linguistically and very efficient" (Roche, Parsing with finite state transducers 1997). In another paper, published in the same book, he describes an efficient method for deterministic part of speech tagging using transducers (Roche, Deterministic Part-of-Speech Tagging with Finite-State Transducers 1997).

2. Graphical Grammar Studio

GGS is a tool oriented towards syntactical analysis. GGS grammars are meant to consume sequences of tokens; a state can consume one token at a time. Each token can have an unlimited number of attributes, and a GGS grammar can relate to these to specify acceptance conditions for its nodes.

At its core, GGS is a tool very similar with Nooj (Silberztein 2004), but the latter is meant to become a comprehensive development environment for NLP tasks. Nooj is an improved version of the INTEX system (Silberztein, 1994). It has support for dictionaries, morphological grammars, paradigmatic representations etc. and can read an impressive number of input formats. It also provides means to create chains of grammars.

GGS on the other hand is best suited for matching and annotating sequences. It is a tool that can be easily integrated in various processing chains. It doesn’t have support for dictionaries, like Nooj, and it doesn’t require the presence of certain attributes for its input tokens. All token attributes are treated as key-value pairs which a GGS grammar can refer to, using regular expressions for both the keys and the values.

Being a tool specialized on matching and annotating sequences of tokens, GGS contains several features which Nooj lacks, like recursive depth and loop limits, or look ahead and look behind assertions. There is, though, a feature which GGS lacks in the present version, but which will be introduced with a future version: the possibility to define variables.

But before anything else, GGS is meant to be an open source project which can be used by anyone. Also, the Java platform might offer some technical advantages for some users, over the .NET platform.

GGS’s main component is the GGS Editor. A secondary component is a java library used for integrating the GGS engine in java code.

When a grammar is applied, the GGS engine compiles it first. This process transforms the RTN in a FST (Finite State Transducer) which consumes one token at a time, and which maintains a call stack of jumps between graphs. This conversion creates the possibility to efficiently check for inconsistencies in the grammar before actually running the state machine (e.g. infinite loops, left recursions).
At the moment, GGS accepts only xml input. It requires the name of the tags which represent text units (usually sentences), and expects to find sequences of token tags as the children of these xml nodes. A token tag can have an unlimited number of XML attributes.

Below is a sample of GGS input text.

```
<S>
  <W LEMMA="hol" POS="NOUN" Type="common" Gender="masculine" Number="singular" Definiteness="yes">Holul</W>
  <W LEMMA="bloc" POS="NOUN" Type="common" Gender="masculine" Number="singular" Definiteness="yes">blocului</W>
  <W LEMMA="mirosi" POS="VERB" Type="main" Mood="indic." Tense="imperfect" Person="third" Number="singular">mirosea</W>
  <W LEMMA="a" POS="ADPOSITION" Type="preposition" Formation="simple">a</W>
  <W LEMMA="varză" POS="NOUN" Type="common" Gender="feminine" Number="singular" Definiteness="no">varză</W>
  <W LEMMA="călit" POS="ADJECTIVE" Type="qualificative" Degree="positive" Gender="feminine" Number="singular" Definiteness="no">călită</W>
  <W Type="PERIOD" POS="PERIOD" LEMMA="." /></S>
```

A node of a GGS graph can be one of three types:

- Token Matching Node
- Empty Node
- Jump Node

A **Token Matching Node** consumes the next input token only if the conditions that it imposes are met. **Empty Nodes** do not consume input tokens. They always match in the matching process and are used for organizing grammars and for other features of GGS. A **Jump Node** represents a transition to another graph.

Till the matching process reaches the final node of a grammar, a variable number of input tokens are consumed, representing the matched sequence. This sequence starts with the first token from the given input stream. When searching for multiple matches, some of which are starting in the middle of a sequence of tokens (sentence), GGS applies the matching process described repeatedly, each time offsetting the starting token by one index to the right.

In GGS, a matching / acceptance condition of a node is described as a sequence of key-value pairs which must be present or not in the attributes map of the next token from the input stream. Moreover, both the keys and the values can be specified using regular expressions, providing a great amount of flexibility. GGS comes with an user guide which contains many practical examples.

**GGS features**
To best describe the features of GGS, a few examples are provided. The main graph in Figure 1 matches all pair of words in which the first one is “the”. The “<>” node matches any token because it doesn’t impose any conditions.

Figure 1: The main graph of a simple grammar

The graph in Figure 2 is named „mySequence” and matches sequences of adjectives (provided that ana=#Afp is the annotation for adjectives) which can be optionally separated by „or”, ”and” and comma. <E> stands for the empty node.

The main graph showed in Figure 3 has two jumps to the „mySequence” graph, and matches adjectives sequences followed by nouns or nouns followed by the copulative verb „be” and adjectives.

Figure 3: A main graph which contains jump nodes to mySequence

In the previous examples, the regex /#Nc.*/ is intended to match part of speech tags which stand for common noun. #Afp is intended for adjectives. GGS does not impose any particular tagsets for the annotations of the input tokens. The designer of the grammar is the one which establishes what attribute names and what tagsets is the grammar expecting to receive as input. If a grammar is applied on a text which is not annotated accordingly, then it will not behave as expected.

GGS grammars can also annotate matches. A GGS annotation marks a continuous sequence of tokens. It has a name, and it can contain an unlimited number of attributes. GGS nodes can be used to specify the start and end of annotations. When a path is matched, i.e. when the matching process reaches the final node of the grammar, if the nodes of the parsed path contain annotation specifications, then these are written to the output. This output can then be serialized. The grammar in Figure 4 annotates the sequences it matches.
When the matching process reaches a node which leads to two or more possible paths, by default, the matching process will try to match the input by going on all possible paths, searching for the longest match. The user can set a priority on arcs emerging from the same node. If by going on a priority arc, the matching process manages to find a match (reaches the final node of the grammar), then the rest of the alternative arcs are ignored. This is very useful for creating efficient and precise recursive tools.

With GGS the user can also set recursive limits on certain nodes which play an important role in some recursive mechanism.

Inspired by the look-behind and look-ahead assertions from regular expressions, GGS is the first to offer an implementation of such a mechanism at the token level. An assertion acts as a node condition. The user can restrict the matching process from continuing after a certain node if a secondary grammar (assertion grammar) doesn’t (or does) match the input text. Unlike classic look behind assertions from regular expressions, GGS look behind assertions are not restricted to only fix sized grammars (grammars which consume a constant number of tokens).

Obtaining certain behaviors like determining if a word is the first or the last one in a sentence can be achieved only by using assertions. For example, one can restrict a node from matching if there is a certain token present at a maximum token distance \( d \) to it’s left, by using a negative look behind assertion.

3. **Romanian Prepositional and Noun Phrases**

Noun phrase chunking (NP chunking) is a partial parsing which outputs the phrase structure of only the noun phrases, including usually some other internal phrasal constituent types, like prepositional phrases or verbal phrases (VP) (which both can contain other NPs and so on). Therefore a NP is recursive in structure. More rigorous formalizations support that the NP is even more complex, i.e. there are intermediate phrases such as Quantifier Phrase (QP) and Adjective Phrase (AP) (Abney 1987) which contribute to the structure of a NP.

In the context of this paper NPs can only contain PPs and other NPs. And PPs must contain at least a NP. Adjectives, articles, determiners etc are present in NPs, but not as separate constituents. For visual simplicity, the PPs will not be represented in the annotations, but only their inner NP will be visible as a child of another NP.

The general structure and chunking rules of NP is language dependent. Like in most highly flexional languages and rich in agreements, the order of the words is not very
strict in Romanian. The presence of the agreements make it possible to have very complex NP structures.

A grammar which parses Romanian text in such a manner has been created successfully with GGS. This grammar requires part of speech tags and lemmas for the tokens given as input.

The grammar contains a structure of graphs which match different types of NPs. First, the NPs are classified into direct and oblique (Figure 5) cases based on their center noun. NP_direct and NP_oblic are each jumping to NPs centered around common nouns, proper nouns and pronouns (Figure 6). This manner of structuring the grammar allows to easily define a different behavior for each case. For the cases of NPs centered around a direct noun, the problem is further fine grained into 4 different cases (all combinations of feminine/masculine and singular/plural) (Figure 7).

There are thus 16 cases of noun based NPs for which the grammar is behaving differently. The number of cases handled differently for pronoun centered NPs is 28. This is due to various types of pronoun that can constitute NPs (in this grammar) each having its own particular behavior in Romanian: the personal, demonstrative, possessive, indefinite and negative pronoun.

For most of the parallel paths in the grammar there are priorities set up. This is for efficiency. Once the GGS matching process matches a type of NP it is useless to have it search for some other longer matches (the default priority policy finds the longest match which implies testing all the possible matching paths, in a depth first manner). Even though the grammar handles both pronoun and noun centered NPs, this paper focuses on the latter.
In Figure 7 each type of NP is handled by two iterative nodes. The first column is composed of jumps to graphs which match the noun of the NP and eventual pre modifiers (Figure 8). The nodes from the second column (Figure 7) jump to graphs which match post modifiers. These can contain other NPs; recursive jumps are thereby present. These graphs are quite complex. They were designed to successfully annotate test cases which cover various linguistic phenomena of interest for the problem of NP chunking. Figure 9 shows the graph which matches post modifiers for NPs centered on a feminine, singular, direct case, common nouns. To reduce the number of visual elements, the path priorities are not visible in this picture.

Examples of linguistic phenomena covered by the post modifiers matching graph:

- A post modifying structure of a noun can be composed of a sequence of adjectival and prepositional phrase post modifiers, in any order.
- The possessive post modifier (matched by the bottom most branch) can be only the last in a sequence of post modifiers.
The grammar contains in total 122 graphs and 1288 nodes from which 212 are token matching nodes and 466 are jump nodes. It contains in total 1621 arcs. The grammar has been applied on the 1984 corpus (6726 sentences, 118334 tokens) created in the Multext East Project\(^2\) (Erjavec, 2004) and matched 28308 NPs. Unfortunately, due to the lack of a rigorous testing corpus, an evaluation score can’t be presented for now.

4. Implication of semantic information in NP chunking

There is an empirical rule which is obeyed by the grammar; i.e., in the case of a recursive NP sequence which ends with a post modifier, it is considered that this modifies the rightmost NP possible (in case of adjectival post modifiers, grammatical agreement must be satisfied). In the case of a prepositional phrase modifier it will be considered to modify the rightmost NP, because no grammatical agreement must be satisfied. This rule usually works well and seems to be the default manner in which the Romanian speaker understands NP post modifiers. Yet there are exceptions which suggest that human generation and parsing of NP structure also involves a semantic understanding.

The constructed grammar considers only morphological information but this is not sufficient for solving cases where semantic information is involved. The following example illustrates this in English:

- [Beds for [children with [iron legs]]] – incorrect (the output of the NP chunker)
- [Beds for [children] with [iron legs]] - correct

An idea is suggested towards making the grammar correctly solve this type of confusion: just like the adjectival post modifier is considered to modify the rightmost NP which satisfies grammatical agreement, in the same manner all post modifiers should be considered to modify the rightmost NPs which also satisfy a semantic agreement. The manner in which such a semantic agreement would be formalized or the detail of an actual implementation, remain the subject of future research.

GGS would still be a great solution for implementing such rules which also consider semantic agreement. A preprocessing module would be required to annotate the input with semantic annotation which would serve this purpose.

Another semantic implication was identified by observing failed cases of the NP chunker in which a prepositional phrase is actually modifying a verb, but because it is positioned immediately to the right of another NP (which usually has the role of object or complement) the NP chunker considers it a post modifier of this. Below is an example of such confusion illustrated in English.

- He greeted [the man with [the hammer]] – correct
- He broke [the stone with [the hammer]] – incorrect

The semantic implication is obvious and is deeply infiltrated in the logic of correctly parsing the second sentence. This simple example suggests that semantic information

---

\(^2\) http://nl.ijs.si/ME
about the verbs, the prepositions of their objects and even the syntactic category of these might be required for correctly parsing the example.

5. Conclusions

The first part of this paper presented recursive transitional networks to introduce Graphical Grammar Studio as a NLP tool for finding and annotating sequences of tokens which can easily be integrated in processing chains.

The second part presents the details behind writing a complex grammar in general and more specifically creating rules for a Romanian NP chunker.

The NP chunking grammar created is quite impressive. All the 16 parallel noun centered NP matching graphs are very similar in structure, with only a few differences meant to check for agreement with various post and pre modifiers. The question would naturally rise: “wouldn’t it be easier to have less such graphs and use some sort of parameters to check for agreement between different tokens attributes (gender, number etc.)”? The answer is yes; this will be possible using variables once they will be implemented in GGS. Another obvious question might be: “Nooj has support for variables, why not use it instead?” Unfortunately, Nooj doesn’t provide the possibility to define the visibility of variables. All variables are global. Any value modification of any variable from a recursive level is visible from all its superior recursion levels.

Finally, a conclusion on the involvement of semantic understanding of language when it comes to NP chunking is underlined. By creating a morphologic based NP chunker and by observing the failed cases, situations of semantic information implication can be extracted. The paper suggests future research ideas in this area.

Bibliography


Silberztein, Max. Nooj : an Object-Oriented Approach. INTEX pour la Linguistique et le Traitement Automatique des Langues, C. Muller, J. Royauté M. Silberztein
RADU SIMIONESCU
