Cascaded Regular Grammars over XML Documents*

Kiril Simov, Milen Kouylekov, Alexander Simov
CLaRK Programme & BulTreeBank Project
http://www.BulTreeBank.org
Linguistic Modelling Laboratory - CLPPI, Bulgarian Academy of Sciences
Acad. G.Bonchev Str. 25A, 1113 Sofia, Bulgaria
Tel: (+3592) 979 28 25, (+3592) 979 38 12, Fax: (+3592) 70 72 73
kivs@bultreebank.org, mkouylekov@dir.bg, adis_78@dir.bg

Abstract

The basic mechanism of CLaRK for linguistic processing of text corpora is the cascade regular grammar processor. The main challenge to the grammars in question is how to apply them on XML encoding of the linguistic information. The system offers a solution using an XPath language for constructing the input word to the grammar and an XML encoding of the categories of the recognized words.

1 Introduction

This paper describes a mechanism for definition and application of cascaded regular grammars over XML documents. The main problems are how to define the input words for a grammar and how to incorporate back in the document the grammar categories returned by the rules of the grammar. The presented solutions are implemented within the CLaRK System – an XML-based System for Corpora Development (Simov et. al., 2001).

The main goal behind the design of the system is the minimization of human intervention during the creation of corpora. Creation of corpora is still an important task for the majority of languages like Bulgarian, where the invested effort in such development is very modest in comparison with more intensively studied languages like English, German and French. We consider the corpora creation task as editing, manipulation, searching and transforming of documents. Some of these tasks will be done for a single document or a set of documents, others will be done on a part of a document. Besides efficiency of the corresponding processing in each state of the work, the most important investment is the human labor. Thus, in our view, the design of the system has to be directed to minimization of the human work. For document management, storing and querying we have chosen the XML technology because of its popularity and its ease for understanding. The XML technology becomes a part of our lives and a predominant language for data description and exchange on the Internet. Moreover, a lot of already developed standards for corpus descriptions like (XCES, 2001) and (TEI, 2001) are already adapted to the XML requirements. The core of the CLaRK System is an XML Editor which is the main interface to the system. With the help of the editor the user can create, edit or browse XML documents. To facilitate the corpus management, we enlarge the XML inventory with facilities that support linguistic work. We added the following basic language processing modules: a tokenizer with a module that supports a hierarchy of token types, a finite-state engine that supports the writing of cascaded regular grammars and facilities for regular pattern search, the XPath query language which is able to support navigation over the whole set of mark-up of a document, mechanisms for imposing constraints over XML documents which are applicable in the context of some events. We envisage several usages of our system:

1. *Corpora markup.* Here users work with the XML tools of the system in order to mark-up texts with respect to an XML DTD. This task usually requires an enormous human effort and comprises both the mark-up itself and its validation afterwards. Using the available grammar resources such as morphological analyzers or partial parsing, the system can state local constraints reflecting the characteristics of a particular kind of texts or mark-up. One example of such constraints can be as follows: a PP according to a DTD can have as parent an NP or VP, but if the left sister is a VP then the only possible parent is VP. The system can use such kind of constraints in order to support the user and minimize his/her work.
2. *Dictionary compilation for human users.* The system supports the creation of the actual lexical entries whose structure is defined via a DTD. The XML tools will be used also for corpus investigation that provides appropriate examples

* This work is funded by the Volkswagen Stiftung, Federal Republic of Germany under the Programme "Cooperation with Natural and Engineering Scientists in Central and Eastern Europe" contract I/76 887. The authors are grateful to Tylman Ule from SfS, University of Tübingen, Germany and the three anonymous reviewers for their comments on the earlier drafts of the paper. Needless to say, all remaining errors in the text are ours.

of the word usage in the available corpora. The constraints incorporated in the system are used for writing a grammar of the sub-languages of the definitions of the lexical items, for imposing constraints over elements of lexical entries and the dictionary as a whole.

3. *Corpora investigation.* The CLaRK System offers a set of tools for searching over tokens and mark-up in XML corpora, including cascaded grammars, XPath language. The combinations between these tools are used for tasks such as: extraction of elements from a corpus - for example, extraction of all NPs in the corpus; concordance - for example, give me all NPs in contexts, ordered by a user's defined set of criteria.

The structure of the paper is as follows: in the next section we give a short introduction to the main technologies on which the CLaRK System is built. These are: XML technology; Cascaded regular grammars; Unicode-based tokenizers; and Constraints over XML documents. The third section describes the definition of cascaded regular grammars. The fourth section presents an approach for applying cascaded regular grammars over XML documents. The last section concludes the paper.

2 The technologies behind the CLaRK System

CLaRK is an XML-based software system for corpora development implemented in JAVA. It incorporates several technologies:

- XML technology;
- Unicode;
- Regular Grammars (they are presented in the next sections);
- Constraints over XML Documents.

2.1 XML Technology

The XML technology is at the heart of the CLaRK system. It is implemented as a set of utilities for structuring, manipulation and management of data. We have chosen the XML technology because of its popularity, its ease of understanding and its already wide use in description of linguistic information. Besides the XML language (see (XML, 2000)) processor itself, we have implemented an XPath language (see (XPath, 1999)) engine for navigation in documents and an XSLT language (see (XSLT, 1999)) engine for transformation of XML documents. The documents in the system are represented as DOM Level1 trees (see (DOM, 1998)). We started with basic facilities for creation, editing, storing and querying of XML documents and developed further this inventory towards a powerful system for processing not only single XML documents but an integrated set of documents and constraints over them. The main goal of this development is to allow the user to add the desirable semantics to the XML documents.

In the implementation of cascaded regular grammars within the CLaRK System, a crucial role plays the XPath language. XPath is a powerful language for selecting elements from an XML document. The XPath engine considers each XML document as a tree where the nodes of the tree represent the elements of the document, the document's most outer tag is the root of the tree and the children of a node represent the content of the corresponding element. The content nodes can be *element* nodes or *text* nodes. Attributes and their values of each element are represented additionally to the tree.

The XPath language uses the tree-based terminology to point to some direction within the tree. One of the basic notions of the XPath language is the so called *context node*, i.e. a chosen node in the tree. Each expression in the XPath language is evaluated with respect to some context node. The nodes in the tree are categorized with respect to the context node as follows: the nodes immediately under the context node are called *children* of the context node; all nodes under the context node are called *descendant* nodes of the context node; the node immediately above the context node is called *parent* of the context node; all nodes that are above the context node are called *ancestor* nodes of the context node; the nodes that have the same parent as the context node are called *sibling* nodes of the context node; siblings of the context node are divided into two types: *preceding siblings* and *following siblings*, depending on their order with respect to the context node in the content of their parent - if the sibling is before the context node, then it is a preceding sibling, otherwise it is a following sibling. Attribute nodes are added with the context node as *attribute* nodes and they are not children, descendant, parent or ancestor nodes of the context node. The context node with respect to itself is called *self* node.

2.2 Tokenization

XML considers the content of each text element a whole string that is usually unacceptable for corpus processing. For this reason it is required for the wordforms, punctuation and other tokens in the text to be distinguished. In order to solve this problem, the CLaRK system supports a user-defined hierarchy of tokenizers. At the very basic level the user can define a tokenizer in terms of a set of token types. In this basic tokenizer each token type is defined by a set of UNICODE symbols. Above this basic level tokenizers the user can define other tokenizers for which the token types are defined as regular expressions over the tokens of some other tokenizer, the so called parent tokenizer. For each tokenizer an alphabetical order over the token types is defined. This order is used for operations like the comparison between two tokens, sorting and similar.

2.3 Constraints on XML documents

Several mechanisms for imposing constraints over XML documents are available. The constraints cannot be stated by the standard XML technology (even by the means of XML Schema (XML Schema, 2000)). The following types of constraints are implemented in CLaRK: 1) finite-state constraints - additional constraints over the content of given elements based on a document context; 2) number restriction constraints - cardinality constraints over the content of a document; 3) value constraints - restriction of the possible content or parent of an element in a document based on a context. The constraints are used in two modes: checking the validity of a document regarding a set of constraints; supporting the linguist in his/her work during the process of corpus building. The first mode allows the creation of constraints for the validation of a corpus according to given requirements. The second mode helps the underlying strategy of minimization of the human labor.

The general syntax of the constraints in the CLaRK system is the following:

```
(Selector, Condition, Event, Action)
```

where the selector defines to which node(s) in the document the constraint is applicable; the condition defines the state of the document at the time when the constraint is applied. The condition is stated as an XPath expression which is evaluated with respect to each node selected by the selector. If the evaluation of the condition is a non-empty list of nodes then the constraints are applied; the event defines some conditions of the system when this constraint is checked for application. Such events can be: the selection of a menu item, the pressing of key shortcut, some editing command as enter a child or a parent and similar; the action defines the way of the actual application of the constraint.

Here we present constraints of type "Some Children". This kind of constraints deal with the content of some elements. They determine the existence of certain values within the content of these elements. A value can be a token or an XML mark-up and the actual value for an element can be determined by the context. Thus a constraint of this kind works in the following way: first it determines to which elements in the document it is applicable, then for each such element in turn it determines which values are allowed and checks whether in the content of the element some of these values are presented as a token or an XML mark-up. If there is such a value, then the constraint chooses the next element. If there is no such a value, then the constraint offers to the user a possibility to choose one of the allowed values for this element and the selected value is added to the content as a first child. Additionally, there is a mechanism for filtering of the appropriate values on the basis of the context of the element.

3 Cascaded Regular Grammars

The CLaRK System is equipped with a finite-state engine which is used for several tasks in the system such as validity check for XML documents, tokenizers, search and cascaded regular grammar. In this and the next section we present the use of this engine for cascaded regular grammars over XML documents along the lines described in (Abney, 1996). The general idea underlying cascaded regular grammars is that there is a set of regular grammars. The grammars in the set are in particular order. The input of a given grammar in the set is either the input string if the grammar is first in the order or the output string of the previous grammar. Another specific feature of the cascaded grammars is that each grammar tries to recognize only a particular category in the string but not the whole string. The parts of the input word that are not recognized by the grammar are copied to the output word. Before going into detail of how to apply grammars in the CLaRK System some basic notions about regular expressions are given.

Regular grammars standardly are formalized as a set of rewriting rules of the following kinds

```
A -> b C
A -> B
A -> b
```

where A, B, C stand for non-terminal symbols and b stands for terminal symbols. Such grammars are applied over a word of terminal symbols in order to parse it to a special goal symbol S. Each language accepted by such a grammar is called regular. Using such a formalization one could situated the regular languages within the families of other languages like context free languages, context sensitive languages and so on. In practice this formalization is rarely used. Other formal devices for dealing with regular languages are regular expressions and finite-state automata with the well know correspondence between them. Although regular grammars are not expressive enough in order to be a good model of natural languages they are widely used in NLP. They are used in modelling of inflectional morphology (see (Koskenniemi, 1983)), tokenization and Named Entity recognition (Grover et. al., 2000), and many others.

In our work we modify the definition of regular grammars along the lines of (Abney, 1996). We use rewriting rules of the following kind:

```
C -> R
```

where `R` is a regular expression and `C` is a category of the words recognized by `R`. We can think of `C` as a name of the language recognized by `R`.

A *regular grammar* is a set of rules such that the regular expressions of the rules recognize pairwise disjoint languages. The disjointness condition is necessary in order the grammar to assign a unique category to each word recognized by it.

A regular grammar works over a word of letters, called *input word*. The grammar scans the input word from left to right trying to find the first sub-word such that it belongs to the language of the regular expression presented by some of the rules of the grammar. If there is no such word starting with the first letter in the input word, then the grammar outputs the first letter of the word and prolongs the scanning with the second letter. When the grammar recognizes a sub-word then it outputs the category of the corresponding rule and prolongs the scanning with the letter after the recognized sub-word. The grammar works deterministically over the input word. The result of the application of the grammar is a copy of the input word in which the recognized sub-words are substituted with the categories of the grammar. The result word is called *output word* of the grammar. In this respect such kind of regular grammars could be considered a kind of finite-state transducers.

An additional requirement suggested by (Abney, 1996) is the so-called *longest match*, which is a way to choose one of the possible analyses for a grammar. The longest match strategy requires that the recognized sub-words from left to right have the longest length possible. Thus the segmentation of the input word starts from the left and tries to find the first longest sub-words that can be recognized by the grammar and so on to the end of the word.

An example of such a regular grammar is the grammar Γ_1 for recognition of dates in the format `dd.mm.yyyy` (10.11.2002) defined by the following rule:

```
Date ->
    ( (0,(1|2|3|4|5|6|7|8|9)) |
      ((1|2),(0|1|2|3|4|5|6|7|8|9)) |
      (3,(0|1))
    )
    ,
    .
    ,
    ((0,(1|2|3|4|5|6|7|8|9))|(1,(0|1|2)))
    ,
    .
    ,
    (((1|2|3|4|5|6|7|8|9),
      (0|1|2|3|4|5|6|7|8|9)*))
```

Application of this grammar on the following input word

```
The feast is from 12.03.2002 to 15.03.2002.
```

will produce the output word

```
The feast is from Date      to Date.
```

A *cascaded regular grammar* (Abney, 1996) is a sequence of regular grammars defined in such a way that the first grammar works over the input word and produces an output word, the second grammar works over the output word of the first grammar, produces a new output word and so on.

As one example of a cascaded regular grammar let us consider the sequence of Γ_1 as defined above and the grammar Γ_2 defined by the following rule:

```
Period -> from, Date, to, Date
```

Application of the grammar Γ_2 on output of the grammar Γ_1:

```
The feast is from Date      to Date.
```

will produce the following output word

```
The feast is Period.
```

In the next section we describe how cascaded regular grammars can be applied to XML documents.

4 Cascaded Regular Grammars over XML Documents

The application of the regular grammars to XML documents is connected with the following problems:

- how to treat the XML document as an input word for a regular grammar;
- how should the returned grammar category be incorporated into the XML document; and
- what kind of 'letters' to be used in the regular expressions so that they correspond to the 'letters' in the XML document.

The solutions to these problems are described in the next paragraphs.

First of all, we accept that each grammar works on the content of an element in an XML document. The content of each XML element[1] is either a sequence of XML elements, or text, or both (MIXED content). Thus, our first task is to define how to turn the content of an XML element into an input word of a grammar. We consider the two basic cases - when the content is text and when it is a sequence of elements.

When the content of the element to which the grammar will be applied is a text we have two choices:

[1]Excluding the EMPTY elements on which regular grammar cannot be applied.

1. we can accept that the 'letters' of the grammars are the codes of the symbols in the encoding of the text; or
2. we can segment the text in meaningful non-overlapping chunks (in usual terminology tokens) and treat them as 'letters' of the grammars.

We have adopted here the second approach. Each text content of an element is first tokenized by a tokenizer[2] and is then used as an input for grammars. Additionally, each token receives a unique type. For instance, the content of the following element

```
<s>
 John loves Mary who is in love with Peter
</s>
```

can be segmented as follows:

```
"John"  CAPITALFIRSTWORD
" "     SPACE
"loves" WORD
" "     SPACE
"Mary"  CAPITALFIRSTWORD
" "     SPACE
"who"   WORD
" "     SPACE
"is"    WORD
" "     SPACE
"in"    WORD
" "     SPACE
"love"  WORD
" "     SPACE
"with"  WORD
" "     SPACE
"Peter" CAPITALFIRSTWORD
```

Here on each line in double quotes one token from the text followed by its token type is presented.

Therefore when a text is considered an input word for a grammar, it is represented as a sequence of tokens. How can we refer now to the tokens in the regular expressions in the grammars? The most simple way is by tokens. We decided to go further and to enlarge the means for describing tokens with the so called *token descriptions* which correspond to the letter descriptions in the above section on regular expressions. In the token descriptions we use strings (sequences of characters), wildcard symbols `#` for zero or more symbols, `@` for zero or one symbol, and *token categories*. Each token description matches exactly one token in the input word.

We divide the token descriptions into two types - those that are interpreted directly as tokens and others that are interpreted as token types first and then as tokens belonging to these token types.

The first kind of token descriptions is represented as a *string* enclosed in double quotes. The string is interpreted as one token with respect to the current tokenizer. If the string does not contain a wildcard symbol then it represents exactly one token. If the string contains the wildcard symbol `#` then it denotes an infinite set of tokens depending on the symbols that are replaced by `#`. This is not a problem in the system because the token description is always matched by a token in the input word. The other wildcard symbol is treated in a similar way, but zero or one symbol is put in its place. One token description may contain more than one wildcard symbol.

Examples:

`"Peter"` as a token description could be matched only by the last token in the above example.

`"lov#"` could be matched by the tokens `"loves"` and `"love"`

`"lov@"` is matched only by `"love"`

`"@"` is matched by the token corresponding to the intervals `" "`

`"#h#"` is matched by `"John"`, `"who"`, and `"with"`

`"#"` is matched by any of the tokens including the spaces.

The second kind of token description is represented by the dollar sign `$` followed by a *string*. The string is interpreted as either a token type or a set of token types if it contains wildcard symbols. Then the type of the token in the input word is matched by the token types denoted by the string. If the token type of the token in the text is denoted by the token description, then the token is matched to the token description.

Examples:

`$WORD` is matched to `"loves"`, `"who"`, `"is"`, `"in"`, `"love"`, `"with"`

`$CAP#` is matched to `"John"`, `"Mary"` and `"Peter"`

`$#WORD` is matched to `"John"`, `"loves"`, `"Mary"`, `"who"`, `"is"`, `"in"`, `"love"`, `"with"`, and `"Peter"`

`$#` is matched to any of the tokens including the spaces.

Now we turn to the case when the content of a given element is a sequence of elements. For instance the above sentence can be represented as:

```
<s>
 <N>John</N><V>loves</V><N>Mary</N>
 <Pron>who</Pron><V>is</V><P>in</P>
 <N>love</N><P>with</P><N>Peter</N>
</s>
```

At first sight the natural choice for the input word is the sequences of the tags of the elements: `<N>` `<V>` `<N>` `<Pron>` `<V>` `<P>` `<N>` `<P>` `<N>`, but when the encoding of the grammatical features is more sophisticated (see below):

[2]There are built-in tokenizers which are always available and there is also a mechanism for defining new tokenizers by the user.

```
<s>
  <w g="N">John</w>
  <w g="V">loves</w>
  <w g="N">Mary</w>
  <w g="Pron">who</w>
  <w g="V">is</w>
  <w g="P">in</w>
  <w g="N">love</w>
  <w g="P">with</w>
  <w g="N">Peter</w>
</s>
```

then the sequence of tags is simply `<w> <w> ...`, which is not acceptable as an input word. In order to solve this problem we substitute each element with a sequence of values. This sequence is determined by an XPath expression that is evaluated taking the element node as context node. The sequence defined by an XPath expression is called *element value*. Thus each element in the content of the element is replaced by a sequence of text segments of appropriate types as it is explained below.

For the above example a possible element value for tag `w` could be defined by the XPath expression: "**attribute::g**". This XPath expression returns the value of the attribute `g` for each element with tag `w`. Therefore a grammar working on the content of the above sentence will receive as an input word the sequence: `"<" "N" ">" "<" "V" ">" "<" "N" ">" "<" "Pron" ">" "<" "V" ">" "<" "P" ">" "<" "N" ">" "<" "P" ">" "<" "N" ">"`. The angle brackets `"<" ">"` determine the boundaries of the element value for each of the elements.

Besides such text values, by using of XPath expressions one can point to arbitrary nodes in the document, so that the element value is determined differently. In fact, an XPath expression can be evaluated as a list of nodes and then the element value will be a sequence of values.

For example, if the element values for the above elements are defined by the following XPath expression: "**text() | attribute::g**"[3], then the input word will be the sequence: `"<" "John" "N" ">" "<" "loves" "V" ">" "<" "Mary" "N" ">" "<" "who" "Pron" ">" "<" "is" "V" ">" "<" "in" "P" ">" "<" "love" "N" ">" "<" "with" "P" ">" "<" "Peter" "N" ">"`.

Using predicates in the XPath expressions one can determine the element values on the basis of the context. For example, if in the above case we want to use the grammatical features for verbs, nouns and pronouns, but the actual words for prepositions, we can modify the XPath expression for the element value in the following way: "**text()[../attribute::g="P"] | attribute::g[not(../attribute::g="P")]**". In this case the XPath expression will select the textual content of the element if the value of the attribute `g` for this element has value `"P"`. If the value of the attribute `g` for the element is not `"P"`, then the XPath expression will select the value of the attribute `g`. The input word then is: `"<" "N" ">" "<" "V" ">" "<" "N" ">" "<" "Pron" ">" "<" "V" ">" "<" "in" ">" "<" "N" ">" "<" "with" ">" "<" "N" ">"`.

[3]The meaning of this XPath is *point to the text child of the element and the value of the attribute g.*

The element value is a representation of the important information for an element. One can consider it conceptual information about the element. From another point of view it can be seen as a textual representation of the element tree structure.

At the moment in the CLaRK system the nodes selected by an XPath expression are processed in the following way:

1. if the returned node is an element node, then the tag of the node is returned with the additional information confirming that this is a tag;
2. if the returned node is an attribute node, then the value of the attribute is:
   (a) tokenized by an appropriate tokenizer if the attribute value is declared as CDATA in the DTD;
   (b) text if the value of the attribute is declared as an enumerated list or ID.;
3. if the returned node is a text node, then the text is tokenized by an appropriate tokenizer.

Within the regular expressions we use the angle brackets in order to denote the boundaries of the element values. Inside the angle brackets we could write a regular expression of arbitrary complexity in round brackets. As letters in these regular expressions we use again token descriptions for the values of textual elements and the values of attributes. For tag descriptions we use strings which are neither enclosed in double quotes nor preceded by a dollar sign. We can use wildcard symbols in the tag name. Thus
`<p>` is matched with a tag `p`;
`<@>` is matched with all tags with length one.
`<#>` is matched with all tags.

The last problem when applying grammars to XML documents is how to incorporate the category assigned to a given rule. In general we can accept that the category has to be encoded as XML markup in the document and that this mark-up could be very different depending on the DTD we use. For instance, let us have a simple tagger (example is based on (Abney, 1996)):

```
Det -> "the"|"a"
N -> "telescope"|"garden"|"boy"
Adj -> "slow"|"quick"|"lazy"
V -> "walks"|"see"|"sees"|"saw"
Prep -> "above"|"with"|"in"
```

Then one possibility for representing the categories as XML mark-up is by tags around the recognized words:

```
the boy with the telescope
```

becomes

```
<Det>the</Det><N>boy</N>
<Prep>with</Prep>
<Det>the</Det><N>telescope</N>
```

This encoding is straightforward but not very convenient when the given wordform is homonymous like:

```
V -> "move"
N -> "move"
```

In order to avoid such cases we decided that the category for each rule in the CLaRK System is a custom mark-up that substitutes the recognized word. Since in most cases we would also like to save the recognized word, we use the variable `\w` for the recognized word. For instance, the above example will be:

```
<Det>\w</Det> -> "the"|"a"
<N>\w</N> -> "telescope"|"garden"|"boy"
<Adj>\w</Adj> -> "slow"|"quick"|"lazy"
<V>\w</V> -> "walks"|"see"|"sees"|"saw"
<Prep>\w</Prep> -> "above"|"with"|"in"
```

The mark-up defining the category can be as complicated as necessary. The variable `\w` can be repeated as many times as necessary (it can also be omitted). For instance, for "move" the rule could be:

```
<w aa="V;N">\w</w> -> "move"
```

Let us give now one examples in which element values are used. For instance, the following grammar recognizes prepositional phrases:

```
<PP>\w</PP> -> <"P"><"N#">
```

Generally it says that a prepositional phrase consists of an element with element value "P" followed by an element with element value which matches the token description "N#". Usage of the token description "N#" ensures that the rule will work also for the case when a preposition is followed by an element with element value "NP". The application of this grammar on the above example with appropriate definition of element values will result in the following document:

```
<s>
  <w g="N">John</w>
  <w g="V">loves</w>
  <w g="N">Mary</w>
  <w g="Pron">who</w>
  <w g="V">is</w>
  <PP>
     <w g="P">in</w>
     <w g="N">love</w>
  </PP>
  <PP>
     <w g="P">with</w>
     <w g="N">Peter</w>
  </PP>
</s>
```

Here is another example (based on a grammar developed by Petya Osenova for Bulgarian noun phrases) which demonstrates a more complicated regular expressions inside element descriptions:

```
    <np aa="NPsn">\w</np> ->

<("An#"|"Pd@@@sn")>,<("Pneo-sn"|"Pfeo-sn")>
```

Here "An#" matches all morphosyntactic tags for adjectives of neuter gender, "Pd@@@sn" matches all morphosyntactic tags for demonstrative pronouns of neuter gender, singular , "Pneo-sn" is a morphosyntactic tag for the negative pronoun, neuter gender, singular, and "Pfeo-sn" is a morphosyntactic tag for the indefinite pronoun, neuter gender, singular. This rule recognizes as a noun phrase each sequence of two elements where the first element has an element value corresponding to an adjective or demonstrative pronoun with appropriate grammatical features, followed by an element with element value corresponding to a negative or an indefinite pronoun. Notice the attribute `aa` of the category of the rule. It represents the information that the resulting noun phrase is singular, neuter gender. Let us now suppose that the next grammar is for determination of prepositional phrases is defined as follows:

```
<pp>\w</pp> -> <"R"><"N#">
```

where "R" is the morphosyntactic tag for prepositions. Let us trace the application of the two grammars one after another on the following XML element:

```
<text>
  <w aa="R">s</w>
  <w aa="Ansd">golyamoto</w>
  <w aa="Pneo-sn">nisto</w>
</text>
```

First, we defined the element value for the elements with tag `w` by the XPath expression: "**attribute::aa**". Then the cascaded regular grammar processor calculated the input word for the first grammar: `"<" "R" ">" "<" "Ansd" ">" "<" "Pneo-sn" ">"`. Then the first grammar is applied on this input words and it recognizes the last two elements as a noun phrase. This results in two actions: first, the markup of the rule is incorporated into the original XML document:

```
<text>
  <w aa="R">s</w>
  <np aa="NPsn">
      <w aa="Ansd">golyamoto</w>
      <w aa="Pneo-sn">nisto</w>
  </np>
</text>
```

Second, the element value for the new element `<np>` is calculated and it is substituted in the input word of the first grammar and in this way the input word for the second grammar is constructed: `"<" "R" ">" "<" "NPsn" ">"`. Then the second grammar is applied on this word and the result is incorporated in the XML document:

```
<text>
  <pp>
     <w aa="R">s</w>
     <np aa="NPsn">
         <w aa="Ansd">golyamoto</w>
         <w aa="Pneo-sn">nisto</w>
     </np>
  </pp>
</text>
```

Because the cascaded grammar only consists of this two grammars the input word for the second grammar is not modified, but simply deleted.

5 Conclusion

In this paper we presented an approach to application of cascaded regular grammars within XML. This mechanism is implemented within the CLaRK System (Simov et. al., 2001). The general idea is that an XML document could be considered as a "blackboard" on which different grammars work. Each grammar is applied on the content of some of the elements in the XML document. The content of each of these elements is converted into input words. If the content is text then it is converted into a sequence of tokens. If the content is a sequence of elements, then each element is substituted by an element value. Each element value is defined by an XPath expression. The XPath engine selects the appropriate pieces of information which determine the element value. The element value can be considered as the category of the element, or as textual representation of the tree structure of the element and its context. The result of the grammar is incorporated in the XML document as XML markup.

In fact in the CLaRK System are implemented more tools for modification of an XML document, such as: constraints, sort, remove, transformations. These tools can write some information, reorder it or delete it. The user can order the applications of the different tools in order to achieve the necessary processing. We call this possibility **cascaded processing** after the cascaded regular grammars. In this case we can order not just different grammars but also other types of tools.

References

Steve Abney. 1996. *Partial Parsing via Finite-State Cascades.* In: *Proceedings of the ESSLLI'96 Robust Parsing Workshop.* Prague, Czech Republic.

DOM. 1998. *Document Object Model (DOM) Level 1. Specification Version 1.0.* W3C Recommendation. http://www.w3.org/TR/1998/REC-DOM--Level-1-19981001

Claire Grover, Colin Matheson, Andrei Mikheev and Marc Moens. 2000. *LT TTT - A Flexible Tokenisation Tool.* In: *Proceedings of Second International Conference on Language Resources and Evaluation (LREC 2000).*

K. Koskenniemi. 1983. *Two-level Model for Morphological Analysis.* In: *Proceedings of IJCAI-83* , pages: 683-685, Karlsruhe, Germany.

Kiril Simov, Zdravko Peev, Milen Kouylekov, Alexander Simov, Marin Dimitrov, Atanas Kiryakov. 2001. *CLaRK - an XML-based System for Corpora Development.* In: Proc. of the Corpus Linguistics 2001 Conference, pages: 558-560. Lancaster, UK.

Text Encoding Initiative. 1997. *Guidelines for Electronic Text Encoding and Interchange.* Sperberg-McQueen C.M., Burnard L (eds).

Corpus Encoding Standard. 2001. *XCES: Corpus Encoding Standard for XML.* Vassar College, New York, USA.
http://www.cs.vassar.edu/XCES/

XML. 2000. *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation.
http://www.w3.org/TR/REC-xml

XML Schema. 2001. *XML Schema Part 1: Structures.* W3C Recommendation.
http://www.w3.org/TR/xmlschema-1/

XPath. 1999. *XML Path Lamguage (XPath) version 1.0.* W3C Recommendation.
http://www.w3.org/TR/xpath

XSLT. 1999. *XSL Transformations (XSLT) version 1.0.* W3C Recommendation.
http://www.w3.org/TR/xslt