CLaRK - an XML-based System for Corpora Development*

Kiril Simov, Alexander Simov, Milen Kouylekov, Krassimira Ivanova
CLaRK Programme & BulTreeBank Project
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@bgcict.acad.bg, adis_78@dir.bg, mkouylekov@dir.bg, krassy_v@abv.bg

May 19, 2002

1 Introduction

This paper will presented the main body of documentation for the CLaRK System. It is based on a paper presented at the Corpus Linguistics 2001 conference in Lancaster. The idea of the paper is to present the main technologies on which the CLaRK System is based.

In this paper we describe the architecture and the intended applications of the CLaRK System. The development of the CLaRK System started under the Tübingen-Sofia International Graduate Programme in Computational Linguistics and Represented Knowledge (CLaRK). The main aim 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 to be the 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 chose the XML technology because of its popularity and its ease for understanding. Very soon the XML technology will be a part of our lives and it will be the 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 finite-state grammars and facilities that search for finite-state patterns, 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 uses for our system:

*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.

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 work.

2. *Dictionary compilation for human users.* The system will support the creation of the actual lexical entries whose structure will be defined via an appropriate DTD. The XML tools will be used also for corpus investigation that provides appropriate examples of the word usage in the available corpora. The constraints incorporated in the system will be used for writing a grammar of the sublanguages 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 rich set of tools for searching over tokens and mark-up in XML corpora, including cascaded grammars, XPath language. Their combinations 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 the context of their use ordered by a user 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** - notions of XML language and XPath querying language; **Cascaded regular grammars** - regular expressions, cascaded grammars, application within XML technology; **Unicode-based tokenizers** - the encoding within the system is based on Unicode and the user can defines a hierarchy of tokenaizer depending on the needs. The third section describes the main components of the CLaRK System and their functionality, the last section outlines some directions for future development.

2 The technologies behind the CLaRK System

2.1 XML technology

2.1.1 eXtendible Markup Language (XML)

XML stands for *eXtendible Markup Language* (see [XML, 2000]) and it emerged as a new generation language for data description and exchange for Internet use. The language is more powerful than HTML and easier to implement than SGML. Starting as a markup language, XML evolved into a technology for structured data representation, exchange, manipulation, transformation, and querying. The popularity of XML and its ease for learning and use made it a natural choice for the basis of the CLaRK System. This section presents in an informal way some of the most important notions of the XML technology. For more rigorous and full presentation the reader is directed to the corresponding literature on the following address: `http://www.w3c.org/XML` defines the notion of structured document in terms of sequences and inclusions of elements in the structure of the document. The whole document is considered as an element which contains the rest of the elements. The elements of the structure of a document are marked-up by means of *tags*. Tags can surround the *content* of an element or tags can mark some points in the document. In the first case the beginning of an element is marked-up by the so called open tag written as `<tagname>` and the end is marked-up by a closing tag written as `</tagname>`. For example, TEI documents include on top level the following two elements:

```
<TEI.2>
    <TeiHeader> ... content of the TEI header element ...
    </TeiHeader>
    <text> ... Content of the text element ... </text>
</TEI.2>
```

The tags of the second kind, so-called empty elements, are usually represented as `<tag/>`. For example, a line break within a sentence can be represented in the following way:

```
<s>...first line of text... <lb/> ...second line of text...</s>
```

Each element can be connected with a set of *attributes* and their *values*. The currently assigned set of attributes of an element is recorded within the open tag of the element or before the closing slash in an empty element. Some of the attribute-value pairs are by default assigned to some tags and thus it is not obligatory to list them.

One important requirement for an XML document is that elements having common content must strictly include one into another. This means that overlap of the elements is forbidden. Such documents are called *well-formed*. For example, the following document *is not well-formed* and thus it is not an acceptable XML document:

```
<doc><el1> ... <el2> ... </el1> ... </el2></doc>
```

The XML technology defines a set of mechanisms for imposing constraints over XML documents. Such kind of a very basic mechanism is the so called DTD (Document Type Definition). A DTD defines the inclusion of elements and the possible sequences of elements within the content of an element. These definitions are given as ELEMENT statements in the DTD. Each ELEMENT statement has the following format:

```
<!ELEMENT tagname content_definition>
```

where *tagname* is the name of the element and *content_definition* is a definition of the content of this kind of elements. Besides some reserved words as `EMPTY` and `ANY`, the content definition is represented as a regular expression over tag names. This regular expression determines the tags and their order in the content of the enclosing element. Additionally, the DTD can contain definitions of the allowed attributes for the elements, entity declarations and others. For more details, the interested reader is directed to the literature on the corresponding notions - see the above address.

An XML document containing elements whose content obeys the restrictions stated in a DTD is said to be *valid* with respect to this DTD.

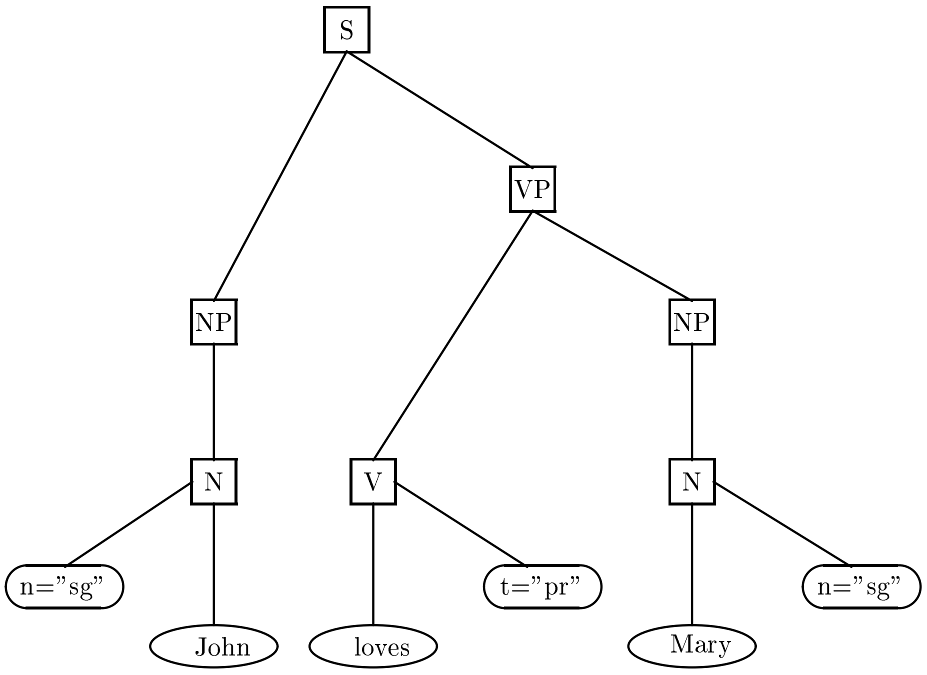
2.1.2 XML Path Language (XPath)

Another important language defined within the XML world and used within the CLaRK System is 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 outer most tag is the root of the tree and the children of a node represent the content of the corresponding element. Attributes and their values of each element are represented as an addition to the tree. This section is based on the following document [XPath, 1999].

For instance, the following XML document:

```
<S>
 <NP><N n="sg"> John </N></NP>
 <VP>
         <V t="pr"> loves </V>
         <NP><N n="sg"> Mary </N></NP>
 </VP>
</S>
```

is represented as the following tree:



In this picture the rectangles correspond to elements in the XML document and they are called *element nodes*, the ovals represent the text elements in the document and they are called *text nodes*, and the rounded rectangles contain the attribute-value pairs and they are called *attribute nodes*. Each rectangle contains the tag of the corresponding element, each text node contains the text from the content of the corresponding element in the document. Each attribute node is attached to the element that contains it. The root of the tree corresponds to the element containing the whole document. The immediate descendant nodes of a given node N represent the content of the node N. The attribute nodes are considered in a different way and they are connected with the corresponding nodes in a different dimension. Thus the attribute nodes are not descendant nodes of any node in the tree.

The XPath language uses the tree-based terminology to point into 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 connected to 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. Some examples: let the `"VP"` node in the above tree be the context node, then its children are the `"V"` node and `"NP"` node that immediately dominates the node `"Mary"`; its descendant nodes (or simply descendants) are the nodes: `"V"`, `"loves"`, `"NP"`, `"N"`, `"Mary"`; its parent is the node `"S"`; this node is also an ancestor of the node `"VP"`; its sibling is the left node `"NP"` and more precisely this is a preceding sibling. If the context node is `"V"` then its ancestor nodes (or simply ancestors) are the nodes `"VP"` and `"S"`; its attribute node is the node `'t="pr"'`; its following sibling is the right `"NP"` node.

Another important concept of the XPath language is the *location path*. Informally, the location path is a description of how to reach some nodes in the tree from the context node. More formally, the location path is a kind of expression. The location path selects a set of nodes relative to the context node. The result of evaluating a location path expression is a node-set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter sets of nodes. The next section describes the syntax and meaning of location path expressions.

Location Paths

Although location paths are not the most general grammatical construct in the language, they are the most important construct and therefore they will be described first. Every location path can be expressed via a straightforward but rather verbose syntax. There are also a number of syntactic abbreviations specific for the CLaRK System (not 100% conformant to the recommendation) that allow common cases to be expressed concisely. This section will explain the semantics of location paths using the unabbreviated syntax. The abbreviated syntax will then be explained by showing its expansion into the unabbreviated syntax . Here are some examples of location paths using the unabbreviated syntax:

- `child::para` selects the `para` element children of the context node
- `child::*` selects all children (element and text nodes) of the context node (different from the recommendation)
- `child::text()` selects all text node children of the context node
- `child::node()` selects all children (element and text nodes) of the context node (different from the recommendation)
- `attribute::name` selects the name attribute of the context node
- `attribute::*` selects all the attributes of the context node
- `descendant::para` selects the `para` element descendants of the context node
- `ancestor::div` selects all `div` ancestors of the context node
- `ancestor-or-self::div` selects the `div` ancestors of the context node and, if the context node is a `div` element, selects the context node itself as well
- `descendant-or-self::para` selects the `para` element descendants of the context node and, if the context node is a `para` element, selects the context node itself as well
- `self::para` selects the context node if it is a `para` element, and otherwise selects nothing
- `child::chapter/descendant::para` selects the `para` element descendants of the `chapter` element children of the context node

- `child::*/child::para` selects all `para` grandchildren of the context node
- `/descendant::para` selects all the `para` elements in the same document as the context node
- `/descendant::olist/child::item` selects all the `item` elements that have an `olist` parent and that are in the same document as the context node
- `child::para[position()=1]` selects the first `para` child of the context node
- `child::para[position()=last()]` selects the last `para` child of the context node
- `child::para[position()=last()-1]` selects the second last `para` child of the context node
- `child::para[position()>1]` selects all `para` children of the context node excluding the first `para` child of the context node
- `following-sibling::chapter[position()=1]` selects the next `chapter` sibling of the context node
- `preceding-sibling::chapter[position()=1]` selects the previous `chapter` sibling of the context node
- `/descendant::figure[position()=42]` selects the forty-second `figure` element in the document
- `/child::doc/child::chapter[position()=5]/child::section[position()=2]` selects the second `section` of the fifth `chapter` of the `doc` document element
- `child::para[attribute::type="warning"]` selects all `para` children of the context node that have a `type` attribute with value `warning`
- `child::para[attribute::type='warning'][position()=5]` selects the fifth `para` child of the context node that has a `type` attribute with value warning
- `child::para[position()=5][attribute::type="warning"]` selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`
- `child::chapter[child::title='Introduction']` selects the `chapter` children of the context node that have one or more `title` children with *string-value* equal to `Introduction`
- `child::chapter[child::title]` selects the `chapter` children of the context node that have one or more `title` children
- `child::*[self::chapter or self::appendix]` selects the `chapter` and `appendix` children of the context node
- `child::*[self::chapter or self::appendix][position()=last()]` selects the last `chapter` or `appendix` child of the context node

There are two kinds of location path: relative location paths and absolute location paths.

A relative location path consists of a sequence of one or more location steps separated by `/` (slash). The steps in a relative location path are composed together from left to right. Each step in turn selects a set of nodes relative to a context node. An initial sequence of steps is composed together with a following step as follows. The initial sequence of steps selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are unioned together. The set of nodes identified by the composition of the steps is this union. For example, `child::div/child::para` selects the `para` element children of the `div` element children of the context node, or, in other words, the `para` element grandchildren that have `div` parents.

An absolute location path consists of `/` followed by a relative location path. The slash `/` selects the root node of the document as a context node for the following relative location path. The slash itself is not a location path in the CLaRK System. If the root node has to be selected, the expression `/self::*` is to be used.

Location Steps

A location step has three parts:

- an axis, which specifies the tree relationship between the nodes selected by the location step and the context node,
- a node test, which specifies the node type or the name of the nodes selected by the location step, and
- zero or more predicates, which use arbitrary expressions for further refining the set of nodes selected by the location step.

The syntax for a location step is the axis name and node test separated by two colons, followed by zero or more expressions, each in square brackets. For example, in `child::para[position()=1]`, *child* is the name of the axis, *para* is the node test and *[position()=1]* is a predicate.

The node-set selected by the location step is the node-set that results from generating an initial node-set from the axis and node-test, and then filtering that node-set by each of the predicates in turn.

The initial node-set consists of the nodes having the relationship to the context node specified by the axis, and having the node type or name specified by the node test. For example, a location step `descendant::para` selects the `para` element descendants of the context node: `descendant` specifies that each node in the initial node-set must be a descendant of the context; `para` specifies that each node in the initial node-set must be an element named `para`. The available axes are described in **Axes**. The available node tests are described in **Node Tests**.

The initial node-set is filtered by the first predicate to generate a new node-set; this new node-set is then filtered using the second predicate, and so on. The final node-set is the node-set selected by the location step. The axis affects the evaluation of the expression in each predicate and so the semantics of a predicate is defined with respect to an axis.

Axes

The following axes are available in the implementation of XPath in the CLaRK System:

- the **child** axis contains the children of the context node
- the **descendant** axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes
- the **parent** axis contains the parent of the context node, if there is one
- the **ancestor** axis contains the ancestors of the context node; the ancestors of the context node consist of the *parent* of the context node and the parent's parent and so on; thus, the ancestor axis will always include the root node, unless the context node is the root node
- the **following-sibling** axis contains all the following siblings of the context node; if the context node is an attribute node, the `following-sibling` axis is empty
- the **preceding-sibling** axis contains all the preceding siblings of the context node; if the context node is an attribute node, the `preceding-sibling` axis is empty
- the **following** axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and attribute nodes

- the **preceding** axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and attribute nodes
- the **attribute** axis contains the attributes of the context node; the axis will be empty unless the context node is an element
- the **self** axis contains just the context node itself
- the **descendant-or-self** axis contains the context node and the descendants of the context node
- the **ancestor-or-self** axis contains the context node and the ancestors of the context node; thus, the `ancestor-or-self` axis will always include the root node

NOTE: The `ancestor`, `descendant`, `following`, `preceding` and `self` axes partition the document (ignoring attributes): they do not overlap and together they contain all the nodes in the document.

Node Tests

The node tests are divided into two categories: node type tests and node name tests.

Here are the node type tests implemented in the system:

- `text()` From the initial node-set it preserves only the text nodes.
- `text(<text>)` From the initial node-set it preserves only the text nodes, which contain the `<text>` as a substring of their text content. For example `child::text("play")` will return all text nodes which are children of the context node and contain the token ”play” in them. This is specific for the CLaRK System[1].

  `<text>` in each of the above uses is, in fact, a token description and could include wildcard symbols as it is described on page 21.
- `node()` This node test does not filter the initial node-set. It is used when all the nodes selected from the axis are needed for further evaluation. For short, `"*"` can be used, i.e. `child::*` is the same as `child::node()`. This is specific for the CLaRK System.
- `element()` From the initial node-set only the element nodes remain.
- `attribute()` From the initial node-set only the attribute nodes remain. It is possible for the initial nodes to contain not only attributes.
- `attribute(<attributeName>)` Filters only for element nodes which have an attribute named `<attributeName>`. Example: `child::attribute(id)` will take only the element nodes, children of the context node, which have an attribute id. This is specific for the CLaRK System.

[1]Additional possibilities in the system are:

`text(<mode>,<text>)`

From the initial node-set preserves only the text nodes, which contain the `<text>` in their text content. Four modes are available. Mode 1 - The text node content is starting with the `<text>`. Mode 2 - The text node contains the `<text>`. Mode 3 - The text node content ends with the text. Mode 4 The text node content is the same as the `<text>`. For example `child::text(3,"play")` will return all text nodes which are children of the context node and contain the token “play” in the end of their content.

`text(<mode>,<y|n>, <"(" regular expression ")">)`

From the initial node-set preserves only the text nodes, which contain text that matches the `< regular expression >`. The text is tokenized by the tokenizer for the element parent of this text node or by the default tokenizer for the DTD if the first is not presented. The second parameter is for normalization. The user must select either `"y"` (yes) or `"n"` (no). The modes are equivalent to the previous node test. For example `child::text(3,y,"play")` will return all text nodes which are children of the context node and contain the token “play” or “Play” or “pLAY” or ... in the end of their content.

- `attribute(<attributeName> = "<attributeValue>")` This node test is almost the same as the preceding node test `attribute(<attributeName>)`, but in addition it also has a restriction on the value of the attribute. Example: `child::attribute(id="243")` will take only these element nodes which have an attribute `id` set to value `243`. This is specific for the CLaRK System.

- `comment()` From the initial node-set only comment nodes remain. The XPath in the CLaRK System supports comments, but the document model for XML documents in the CLaRK System doesn't support them at the moment.

- `processing-instruction()` From the initial node-set only processing-instruction nodes remain. The XPath in the CLaRK System supports processing-instructions, but the document model for XML documents in the CLaRK System doesn't support them at the moment.

The name node tests are used to filter the initial node-set for element nodes with a given name. All other non-element nodes and element nodes with other names are removed from the set. For example, `child::para` takes only the *para* element node children of the context node.

Predicates

An axis is either a forward axis, or a reverse axis. An axis that only contains the context node or nodes that follow the context node in *document order* is a forward axis. An axis that only contains the context node or nodes that precede the context node in *document order* is a reverse axis. Thus, the `ancestor`, `ancestor-or-self`, `preceding`, and `preceding-sibling` axes are reverse axes; all other axes are forward axes. Since the `self` axis always contains only one node, it makes no difference whether it is a forward or reverse axis. The `proximity position` of a member of a node-set with respect to an axis is defined in the following way: the position of the node in the node-set is ordered in document order if the axis is a forward axis and it is ordered in reverse document order if the axis is a reverse axis. The first position is 1.

A predicate filters a node-set with respect to an axis in order to produce a new node-set. For each node in the node-set which is to be filtered, the predicate is evaluated with the node in question as the context node, with the number of nodes in the node-set as the context size, and with the *proximity position* of the node in the node-set with respect to the axis as the context position; if the predicate evaluates to true for that node, the node is included in the new node-set; otherwise, it is not included.

The predicate is evaluated as an expression and the result is converted to a boolean. If the result is a number, the result will be converted to true if the number is equal to the context position and will be converted to false otherwise; if the result is not a number, then the result will be converted as if by a call to the `boolean()` function. Thus a location path `para[3]` is equivalent to `para[position()=3]`. In other words if the context node has 4 child nodes *para*, then only for the third *para* child the predicate `[3]` (or `[position()=3]`) will be true. So the new node-set will contain only the third *para* child.

Abbreviated Syntax

Here are some examples of location paths using abbreviated syntax:

- `para` selects the `para` element children of the context node

- `*` selects all children of the context node

- `text()` selects all text node children of the context node

- `@name` selects the `name` attribute of the context node

- `@*` selects all the attributes of the context node

- `para[1]` selects the first `para` child of the context node

- `para[last()]` selects the last `para` child of the context node

- `*/para` selects all `para` grandchildren of the context node
- `/doc/chapter[5]/section[2]` selects the second `section` of the fifth `chapter` of the `doc`
- `chapter//para` selects the `para` element descendants of the `chapter` element children of the context node
- `//para` selects all the `para` descendants of the document root and thus selects all `para` elements in the same document as the context node
- `//olist/item` selects all the `item` elements in the same document as the context node that have an `olist` parent
- `.` selects the context node
- `.//para` selects the `para` element descendants of the context node
- `..` selects the parent of the context node
- `../@lang` selects the `lang` attribute of the parent of the context node
- `para[@type="warning"]` selects all `para` children of the context node that have a `type` attribute with value `warning`
- `para[@type="warning"][5]` selects the fifth `para` child of the context node that has a `type` attribute with value `warning`
- `para[5][@type="warning"]` selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`
- `chapter[title="Introduction"]` selects the `chapter` children of the context node that have one or more `title` children with *string-value* equal to `Introduction`
- `chapter[title]` selects the `chapter` children of the context node that have one or more `title` children
- `employee[@secretary and @assistant]` selects all the `employee` children of the context node that have both a `secretary` attribute and an `assistant` attribute

The most important abbreviation is that `child::` can be omitted from a location step. In effect, `child` is the default axis. For example, the location path `div/para` is short for `child::div/child::para`.

There is also an abbreviation for attributes: `attribute::` can be abbreviated to `@`. For example, a location path `para[@type="warning"]` is short for `child::para[attribute::type="warning"]` and therefore it selects `para` children with a `type` attribute with value equal to `warning`.

`//` is short for `/descendant-or-self::node()/`.

For example, `//para` is short for `/descendant-or-self::node()/child::para` and so will select any `para` element in the document; `div//para` is short for `div/descendant-or-self::node()/child::para` and therefore it will select all `para` descendants of `div` children.

NOTE: The location path `//para[1]` does not mean the same as the location path `/descendant::para[1]`. The latter selects the first descendant `para` element; the former selects all descendant `para` elements that are the first `para` children of their parents.

A location step of `.` is short for `self::node()`. This is particularly useful in conjunction with `//`. For example, the location path `.//para` is short for `self::node()/descendant-or-self::node()/child::para` and therefore it will select all `para` descendant elements of the context node.

Similarly, a location step of `..` is short for `parent::node()`.

For example, `../title` is short for `parent::node()/child::title` and therefore it will select the `title` children of the parent of the context node.

2.1.3 XSL Transformations (XSLT)

Here the main ideas about XSLT are described and a list of the implemented and unimplemented features is given.

XSLT offers a way to transform a document by applying transformation rules. For complete information about the XSLT standard see ([XSLT, 1999]). Not everything stated in the standard is implemented in the CLaRK System, and some things work differently. For the application of a transformation two things are necessary:

1. A source document that can be any well-formed XML document.
2. A transformation document (TD) that contains the transformation rules.

The rest of this section will be concerned with the transformation document. In the CLaRK System every transformation document is an XML document.

Basic structure of the transformation document

As we said each transformation document in the CLaRK System is a well-formed XML document which can be edited by the system. In part the tag names used in transformation documents are reserved tag names defined in the XSL standard. In order they to be separate from the user-defined tag names, they usually may be preceded by an *xsl* prefix followed by a colon (:) (so a tag named *template* may be also written as *xsl:template*) or in other sense a namespace prefix for XSL. In the following the syntax of the TD is given. For details of how the XSLT processor work see the next section.

The root

The root element of every TD must be marked with either the *transform* or *stylesheet* tag.

Children of the root

Child elements of the root element are marked with a *template* tag. Each of them contains exactly one template rule.

Templates

Attributes of the *template* tag

match attribute

The `match` attribute contains a string that is interpreted as an XPath expression. The pattern matches a node A in the source document iff there exists a node C in the source document such that when the XPath expression in the match attribute is evaluated with C as a context node, the result is a node list and A is member of this list.

mode attribute

The `mode` attribute contains a string that determines the mode in which the XSLT processor must be in order to use this rule. The string is completely arbitrary. The default is the empty string.

priority attribute.

The `priority` attribute determines the priority of the rule. Priorities are used to determine which rule to use if more than one rule matches a given node. If no value is given for this attribute, 0 is substituted. If more than one template still remain, the last defined template is used.

Children of the `template` tag

The children elements of a template element could be of any kind, but some tag names have special meaning.

`apply-template` **tag**

This tag tells the XSLT processor to continue applying the rules.

select attribute

This attribute contains a XPath expression, that is evaluated with the node that the template has matched as a context node. Then each node in the resulting node set is processed. If no value is given, `child::*` is assumed.

mode attribute

This attribute sets the XSLT processor in the given mode. The `apply-template` tag can have optional `sort` children.

`for-each` **tag**

This tag applies its content to every node in the node set it selects. Its content is similar in every way to the content of a `template` tag and is similarly used, but it may have `sort` tag children that determine the order of processing of the nodes.

Attributes:

select attribute

A XPath that selects the nodes to be processed with the body of the `for-each` tag.

`sort` **tag**

The `sort` tag determines the key for sorting of the lists, resulting from XPath expression. When no such tag occurs, the list is processed in document order. Otherwise, it is first sorted. The first sort child determines the first key, the second sort child the second key and so on.

select attribute

This attribute holds an XPath expression which is evaluated for every node in the list and is then converted either to string or to number. It determines the key value. The default value is "`.`".

data-type attribute

This attribute determines whether the key is converted to a string or to a number. It can have the values a `string` or a `number`. The default is `string`.

order attribute

Specifies the order of the resulting list. Valid values are `ascending` and `descending`. Default is `ascending`.

The `sort` tags may appear as children of `for-each` and `apply-templates` tags. Elsewhere it is considered as a normal tag without special meaning.

`if` **tag**

This tag contains a XPath that is evaluated (or casted) to boolean. If the answer is true, then the processor applies the content of the tag to the current node. Otherwise it does nothing.

test attribute

The XPath that is evaluated to boolean. It behaves exactly as if the XPath expression was the following expression `boolean(original_expression)`.

`copy-of` **tag**

This tag contains a XPath, that is evaluated either to a list of nodes, or to a boolean, to a number or a string. In the first case each of the selected nodes together with its subtree is attached to the result. In the othe cases the XPath is converted to string and is then added to the result tree.

select attribute

Selects the nodes (or data) to be copied.

`copy` **tag**

This tags copies the current node and its attributes. The subtree is not copied. No attributes are defined for this tag.

`value-of` **tag**

This tag behaves like `copy-of` tag with `select` attribute converted to `string(original_expression)`.

select attribute

Selects the nodes (or data) to be converted and copied.

These are the special meaning tags. Every other tag or character data is directly copied to the result tree.

XSLT processor

The XSLT processor takes as input a source document (SD) and a transformation document (TD) and returns a new document that is called the result document (RD). Its works as follows :

1. Finds the template that matches the document element of the SD and has mode `""` and highest priority.
2. Applies that template.

In the template there may be "apply-template" tags that run the processor recursively. Every node that has no special meaning is attached to the result tree in the right position node (RP). At first this RP is the document node of the RD itself. For more information see the examples below.

As one may see, the resulting document may not have only one root. To avoid this, only the first child of the Document node is actually used. This is not the case when you apply the transformation inside the document rather than to the entire document.

Examples

The "books" example.

Lets assume that we have a set of "books", and every "book" tag has zero or more "author" tags. We want to extract all the authors and put them in a new document. The following XSLT document will do the job.

Example:

```
<books>
  <book><title>Alice in Wonderland</title><author>Lewis Carrol</author></book>
  <book><title>HPSG-based syntactic treebank of Bulgarian (BulTreeBank).</title>
    <author>Simov</author><author>Popova</author><author>Osenova</author></book>
  <book><title>Creatures of Light and Darkness</title><author>Roger Zelazny</author></book>
  <book><title>The Chronicles of Amber </title><author>Roger Zelazny</author></book>
</books>
```

Note: We will use the "//" for comments. They are not part of the TD.

```
<transform> <template match="*"> // the only template that matches every tag
      <authors>    // this tag has no special meaning
        <for-each select="child::book/child::author">
                    // for every author of every book
                <sort/> // sort the authors
                <copy-of select="."/> // copy its "author"s tags
        </for-each> // end cycle
      </authors>
</template> // end template
</transform> // end the TD
```

The result will be

```
<authors>
 <author>Lewis Carrol</author>
 <author>Osenova</author>
 <author>Popova</author>
 <author>Roger Zelazny</author>
 <author>Roger Zelazny</author>
 <author>Simov</author>
</autohrs>
```

As one may see, we will have authors that are mentioned more than once in the RD. Lets fix this! The new SD will be the old RD, and the transformation is this :

```
<transform>
 <template match="authors"> // match the root
   <copy> // copy the root
     <apply-templates/> // apply-templates to its children
   </copy>
 </template>
 <template match="author"> // match every author
   <if test="(. != following-sibling::author[1])">
              // check if it's the same as its next sibling
              // note that the authors are sorted
       <copy-of select="."/>
   </if>
 </template>
</transform>
```

And the final result is

```
<authors>
 <author>Lewis Carrol</author>
 <author>Osenova</author>
 <author>Popova</author>
 <author>Roger Zelazny</author>
 <author>Simov</author>
</autohrs>
```

Besides XSLT, the CLaRK System also provides additional means for transformations of a document via the XPath Replace menu. It is described in another part of the documentation.

2.2 Cascaded Regular Grammars

The CLaRK System is equipped with a finite-state engine which is used for several tasks in the system. In this 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 underlaying 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 feature of cascaded grammars is that each grammar tries to recognise only a particular category in the string but not the whole string. Before going into detail of how to apply grammars in the CLaRK System some basic notions about regular expressions are given.

2.2.1 Regular Expressions and Grammars

This is not a completely formal presentation of the topic. Our purpose here is to clarify the terminology that will be used further in the document.

Letter

We call a *letter* each designate symbol. Symbol here is understood in a general way and not as a symbol code in the computer. Symbol can be a word from a natural language as 'love' in English.

Alphabet

An *alphabet* is a set of letters.

Word

A *word* is a finite sequence of letters over some alphabet. The empty sequence of letters will be called the empty word and we will write it as ϵ. The set of all words over the alpabet A is denoted by $A*$. A basic operation over words is *concatenation* – $\oplus$. The concatenation is a binary operation. The concatenation of two words is the word which consist of the letters of the first word followed by the letters of the second word. For example:

$grand \oplus mother = grandmother$

Sometimes we simplify the notation and we write **l** for the letter **l** and for the word that consists of this letter.

Language

A *language* is a set of words over some alphabet.

Regualar expression

A *regualar expression* over an alphabet is an expression that can be interpreted as a set of words over this alphabet. This set of words is the language that is recognised by the regular expression. Before giving the syntax and semantics of the regular expressions we will enrich our formal inventory with *letter descriptions.* A letter description is an expression ld which denotes a set of letters from a given alphabet. Each letter from a given alphabet is a description of itself. The set of letters denoted by the letter description ld is represented as $A(ld)$, i.e. `[a-z]`, `\p`.

Syntax of the regualar expressions:

1. (Basic case)	If ld is a letter description, then ld is a regular expression.
2. (Concatenation)	If r_1 and r_2 are regular expressions, then (r_1, r_2) is a regular expression.
3. (Union)	If r_1 and r_2 are regular expressions, then $(r_1 \mid r_2)$ is a regular expression.
4. (One or zero)	If r is a regular expression, then $r?$ is a regular expression.
5. (Zero or more)(Kleene star)	If r is a regular expression, then $r*$ is a regular expression.
6. (One or more)(Kleene plus)	If r is a regular expression, then $r+$ is a regular expression.

Semantics of the regular expressions:

The semantics of the regular expressions is defined by an interpretation function $L(.)$ which maps each regular expression into the set of words denoted by this regular expression. The function $L(.)$ is total and meets the following equations:

1. (Basic case)

   If ld is a letter description, then $L(ld) = \{l \mid l \in A(ld)\}$

Thus each letter description ld denotes the set of words corresponding to the letters described by ld.

2. (Concatenation)

   If r_1 and r_2 are regular expressions, then $L((r_1, r_2)) = \{w_1 \oplus w_2 \mid w_1 \in L(r_1) \text{ and } w_2 \in L(r_2)\}$

Thus the interpretation of a concatenation expression is the set of words so that each word in the set is a concatenation of two words, where the first word is from the interpretation of the first regular expression and the second one is from the interpretation of the second expression.

3. (Union)

If r_1 and r_2 are regular expressions, then $L((r_1 \mid r_2)) = \{w \mid w \in L(r_1) \text{ or } w \in L(r_2)\}$

Thus the interpretation of a union expression is the set of words so that each word in the set is either from the interpretation of the first regular expression or from the interpretation of the second expression.

Another way to define the interpretation of a union expression is by $L((r_1 \mid r_2)) = L(r_1) \cup L(r_2)$.

4. (One or zero)

If r is a regular expression, then $L(r?) = \{\epsilon\} \cup L(r)$.

Thus the interpretation of an one-or-zero expression is the interpretation of the expression unioned with the set of the empty word.

5. (Zero or more)(Kleene star)

If r is a regular expression, then $L(r*) = \{\epsilon\} \cup \{w_1 \oplus w_2 \oplus \dots w_n \mid w_I \in L(r) \text{ and } n > 0 \text{ is a natural number }\}$.

Thus the interpretation of a zero-or-more expression is the set of the empty word unioned with the set of all words in the interpretation of the expression r and all words that are concatenations of two or more words from the interpretation of the expression r.

6. (One or more)(Kleene plus)

If r is a regular expression, then $L(r+) = \{w_1 \oplus w_2 \oplus \dots w_n \mid w_I \in L(r) \text{ and } n > 0 \text{ is a natural number }\}$.

Thus the interpretation of an one-or-more expression is the set of all words in the interpretation of the expression r and all words that are concatenations of two or more words from the interpretation of the expression r.

Some simplifications of the syntax are allowed. When only the operations concatenation and union are used, the internal brackets can be omitted. For instance, $((a, b), (c, d))$ can be written as (a, b, c, d).

The set of words in the interpretation of a regular expression is called the language recognised by the expression. Each language that can be recognised by a regular expression is called a *regular language*. Notice that different regular expressions can recognise the same language. Also a word can belong to different languages.

Regular Grammars

A regular expression can be used to determine whether a given word belongs to the language recognised by the expression, but sometimes we want to recognise the internal structure of a word. This can be done by segmenting the word in subwords and by assigning some categories to these subwords. For instance, let we write a regular expression for a recognition of dates in the format `dd.mm.yyyy` (10.11.2002):

```
( ( 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)*))
```

This regular expression successfully recognises 10.11.2002 as a word over the alphabet {0,1,2,3,4,5,6,7,8,9,.}.

It also accepts some words that are not dates as 31.02.1999, but this is not important in our example.

Sometimes we would like to segment the date in a day part, a month part and a year part. In order to do this we use rules of the following kind:

```
        C -> R
```

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

A *regular grammar* is a set of rules.

A regular grammar works over a word (called input word) and it tries to segment this word in a sequence of subwords in such a way that each of these subwords be recognised by a regular expression of the rules in the grammar. The result of the grammar's work is a new word representing the sequence of the categories assigned to each subword of the input word. The segmentation of the input word and the result word of categories is called an analysis with respect to the grammar. For instance, let us represent the above regular expression as a grammar:

```
D -> ( ( 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)) )
R -> .
M -> ( (0,(1|2|3|4|5|6|7|8|9)) | (1,(0|1|2)) )
Y -> (( (1|2|3|4|5|6|7|8|9), (0|1|2|3|4|5|6|7|8|9)*))
```

Then for the date 10.11.2002 some of the possible analyses are:

```
 D  R    M  R      Y
10  .   11  .   2002

 D  R    D  R      Y
10  .   11  .   2002

 Y  R    Y  R      Y
10  .   11  .   2002

 D  R   Y   Y   R     Y
10  .   1   1   .   2002
```

where the categories are written above the corresponding subwords. As it is obvious from the example processing this way of defining a grammar is not very useful because it produces too many wrong analyses. For a better processing with such kind of grammars one would like to be able to impose additional constraints over the analyses produced by the grammar. In the CLaRK System we made two extensions:

- cascaded application of regular grammar [Abney, 1996]; and
- extension of the rules with descriptions of the left and the right contexts of a subword recognized by the regular expression of the rule.

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 of categories, the second grammar works over the output word of the first grammar, produces a new word of categories and so on. If one of the grammars in the sequence can produce more that one analysis, then these analyses are considered one by one in producing the analyses of the next grammar. The process of application of the cascaded regular grammar stops when all possibilities are finished. The output of the last grammar constitutes the analyses made by the cascaded regular grammar.

Example:

Let us take the previous grammar as a first grammar:

```
D -> ( ( 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)) )
R -> .
M -> ( (0,(1|2|3|4|5|6|7|8|9)) | (1,(0|1|2)) )
Y -> (( (1|2|3|4|5|6|7|8|9), (0|1|2|3|4|5|6|7|8|9)*))
```

And as a second grammar the following grammar with one rule:

```
Date -> (D,R,M,R,Y)
```

Applying the second grammar to the possible analyses of the first grammar, the only analysis that can be constructed is:

```
          Date
--------------------------
 D  R    M  R      Y
10  .   11  .   2002
```

and all other analyses from the first grammar will be ruled out as unacceptable.

One relaxation of the application scenario is that a given regular grammar in the sequence need not necessary recognise all letters in the input word. This means that the segmentation of the input word is such that the grammar recognises some of the subwords, but not all. 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 recognised subwords 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 subwords that can be recognised by the grammar and so on to the end of the word. The output word in this case consists of a sequence of unrecognised subwords of the input word and the categories of the recognised subwords in the order in which the subwords are found in the input word. In the CLaRK System we have implemented also *shortest match* and allow the user to choose the most appropriate strategy for the task at hand.

The second extension of the regular grammars realised in the CLaRK System concerns the description of the left and the right context of a given subword recognised by the regular expression of a rule. Thus the rules in the grammar will have the format:

```
C -> LC : R : RC
```

where `C` is the category, `LC` is a regular expression describing the left context of the words recognisable by the rule, `R` is a regular expression describing the set of words recognisable by the rule, `RC` is a regular expression describing the right context of the words recognisable by the rule. The regular expressions `LC` and `RC` can be empty and then there are no constraints over the left and the right context. We envisage the use of such rules for tasks as sentence boundary recognition where one has to take a look at the word after the full stop in order to determine whether this full stop is marking the end of the sentence or not. The user can control the strategy for applying each of these rules by specifying longest or shortest match for them.

To demonstrate this kind of rules we rewrite the above grammar in the following way:

```
G1:
   M -> . : ( (0,(1|2|3|4|5|6|7|8|9)) | (1,(0|1|2)) ) : .
```

(the left context is a dot and the right context is a dot)

```
G2:
   D -> ( ( 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)) ) : .
```

(the left context is empty)

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

(the right context is empty)

```
G3:
   R ->  .
```

(both contexts are empty)

```
G4:
   Date -> (D,R,M,R,Y)
```

(both contexts are empty)
Longest match is required in all rules of this grammar although in some it is not obligatory.
The analysis will be in four steps:

```
Step1:

             M
    10  .   11  .   2002

Step2:

     D       M         Y
    10  .   11  .   2002

Step3:

     D  R    M  R      Y
    10  .   11  .   2002

Step4:

            Date
    ---------------------
     D  R    M  R     Y
    10  .   11  .   2002
```

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

2.2.2 Cascaded Regular Grammars in the CLaRK System

The basic the CLaRK mechanism for linguistic processing of text corpora is the regular grammar processor. 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 (excluding the EMPTY elements on which regular grammar cannot be applied) is either a sequence of XML elements, or text, or both.

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

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-overlaping chunks (or in usual terminology tokens) and treat them as 'letters' of the grammars.

In the CLaRK System we have adopted 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
```

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

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 as 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 is matching 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 tokeniser. If the string doesn't 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 for `#`. This is not a problem in the system because the token description is always matched by a token in the input word. In a similar way the other wildcard symbol is treated, 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"` and to `"who"`

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

The wildcard symbols `#, @` and the double quote symbols `"` can be used in the token expressions by using the symbol `^` as a escape symbol. Thus "^#" matches the token `"#"`. The symbol `^` is represented as `^^`.

The second kind of token description is represented simply as a *string* which is not enclosed in double quotes or angle brackets - `< >`. The string is 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 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 turns out to be more sophisticate as the following:

```
<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 the context node. The sequence defined by a XPath expression is called *element value* in the CLaRK System. Thus each element in the content of the element is replaced by a sequence of text elements. For the above example a possible element value for tag `w` could be: **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"`. Besides such text elements, by using of XPath expressions one can point to an arbitrary node in the document, so that the element value is determined differently. At the moment the nodes returned 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 that this is a tag;
2. if the returned node is an attribute node, then the value of the attribute is:
   (a) tokenised by an appropriate tokeniser if the attribute value is declared as CDATA in the DTD;
   (b) taken to be 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 tokenised by an appropriate tokeniser.

Within the regular expressions we use the angle brackets in order to denote the tags. 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 we to incorporate the category assigned to a given rule. In general we can accept that the category has to be encoded as XML mark-up in the document and that this mark-up could be very different depending on the appropriate DTD we are using. For instance, if we have a simple tagger (example is based on [Abney, 1996]):

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

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

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

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 recognised word. Since in most cases we would also like to save the recognised word, we use the variable `\w` for the recognised word. For instance, the above example will be:

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

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:

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

2.3 Unicode for multilingual text representation (Tokenization)

The representation of the XML documents inside the system is based on UNICODE. This allows the representation of texts in different languages in a natural way. In XML each text nodes is considered as a sequence of letters. In order to treat these text elements

XML considers the content of each text element as a whole string that is unacceptable for corpus processing where one usually requires to distinguish wordforms, punctuation and other tokens in the text. In order to cope with 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, so called parent tokenizer. In the system tokens are used in different processing modules. For each tokenizer an alphabetical order over the token types is defined. This order is used for operations as comparing two tokens, sorting and similar.

Sometimes in different parts of one document the user will want to apply different tokenizers. For instance in a multilingual corpus the sentences in different languages will need to be tokenized by different tokenizers. In order to allow this functionality, the system allows for attaching tokenizers to the documents via the DTD of the document. To each DTD the user can attach a tokenizer which will be used for tokenization of all textual elements of the documents corresponding to the DTD. Additionally the user can overwrite the DTD tokenizer for some of the elements attaching to them other tokenizers.

The system supports definition of different keyboards for supporting of different languages. At the moment we support the standard American keyboard and Bulgarian keyboard for entering Cyrillic letters

2.4 Constraints

Several mechanisms for imposing constraints over XML documents are available. The constraints cannot be stated by the standard XML technology. The following types of constraints are implemented in the 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 building of a corpus. 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 minimisation of the human labour.

General syntax of the constraints in the CLaRK system is the following:

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

where the selector defines in which node(s) in the document the constraint is applicable; the condition defines the state of the document 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. .

At the moment the following constraints are implemented in the system:

Regular expression constraints

In this kind of constraints the action is defined as a regular expression which is evaluated over the content of each element selected by the selector. If the word formed by the content of the element can be recognized as belonging to the language of the regular expression then the constraint is evaluated as true. Otherwise it is evaluated as false and an appropriate message is given. The content of the element is treated as the content of the element when a grammar is applied to it (see above).

This constraints are useful when the content of a element is text or MIXED and the user wants to impose some constraints over the content or when the DTD defines the content of some kind of elements as a disjunction and the different disjuncts are realised in different contexts which can be determined by a XPath expression.

Number Constraints

This kind of constraints are defined in terms of an XPath expression, which is evaluated to a list of nodes, and MIN and MAX values where MIN and MAX are natural numbers. The constraint is satisfied (evaluated as true) if the length of the list returned by the XPath expression is between MIN and MAX.

Value Constraints

These constraints determine the possible children or the parent of an element in a document. These constraints apply when the user enters a new child or a new parent of an element. In both cases a list of possible children or parents are determined by the DTD, but depending on the context in the document an additional reduction of these lists is possible. In case the only possible child of an element is a text then these constraints determine the possible text values for the element. Let us take as an example the following definitions in a DTD:

```
<!ELEMENT np ((np, pp) | ...) >
<!ELEMENT vp ((vp, pp) | ...) >
```

which in part define that a PP can be attached to a NP or a VP. Then let us take the partially marked-up sentence:

```
<s> <np>The man</np><v>saw</v><np>the boy</np><pp>in the
garden</pp> </s>
```

For the `PP` "in the garden" there are still two possibilities for a parent - a `NP` or a `VP`. But if the user enters a new information than "saw the boy" is a `VP` then for the `PP` "in the garden" there is only one possible parent - a `VP`. This information can be encoded in the system as a value constraint for the parent of `PP` elements. In future versions of the system we envisage such kind of constraints to be compiled from grammar represented in a grammar development environment.

In the next paragraphs 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 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.

This kind of constraints is very useful for filling of the content of elements with predetermined set of values as, for example, in dictionary construction. Within BulTreeBank, we use these constraints for manual disambiguation of morpho-syntactic tags of wordforms in the text. For each wordform we encode the appropriate morpho-syntactic information from the dictionary as two elements: `<aa>` element which contains a list of morpho-syntactic tags for the wordform separated by a semicolon, and `<ta>` element which contains the actual morpho-syntactic tag for this use of the wordform. Obviously the value of `<ta>` element has to be among the values in the list presented in the element `<aa>` for the same wordform. "Some Children" constraints are very appropriate in this case. Using different conditions and filters on the values we implemented and used more than 50 constraints during the manual disambiguation of wordforms in the "golden standard" of the project. It is important to mention that when the context determines only one possible value for some word, it is added automatically to the content of `<ta>` element and thus the constraint becomes a rule.

Similar constraints are constraints of the type "Some Attributes" which are working for the values of some attribute.

3 The CLaRK System

At the heart of the CLaRK System is the XML technology as a set of utilities for structuring, manipulation and management of data. 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 of single XML documents but of an integrated set of documents and constraints over them. The main goal of this development is to allow the user to add to the XML documents a desirable semantics reflecting the user's goals. Inside the system, the core structure of the representation of the XML documents follows the DOM Level1 specification - see [DOM, 1998]. When an XML document is imported (or created) in the system it is stored in this internal representation and in this way the user has access to it only via the facilities of the system. This restriction allows us to support the consistency of the data represented in the system. We plan to exploit this feature of the system even further in future for automatic support of construction of XML documents that reflect the content of a corpus or a set of documents.

The CLaRK System includes the following components: *XML Engine*, *XML Editor*, *Database*, *Document*

Transformation, *Tokenizer*, *Constraints Engine*, *XPath Engine* and *Regular Grammar Engine*. In this section we describe each of these components in turn. Some of these components are not directly accessible by the user and they are used in the other components to support the corresponding functionality. The most important components of this type are the *XPath Engine* and the *Regular Grammar Engine*. The first is a module which evaluates XPath expressions over a document and the second is a module dealing with compilation of regular expressions into finite-state automata, determinization and minimization of the compiled automata.

3.1 XML Engine

XML Engine offers a full set of facilities for processing XML documents. This includes:

DTD compiler

The DTD compiler which compiles the element, entities and attribute definitions in a DTD and represents them in an internal format. For the elements the internal format is a set of finite-state automata corresponding to the content definition of the elements. These automata are determined and minimized during the compilation. Attributes and entities are stored as hash-tables. Additionally in the internal format the user can write comments on the element, attribute and entity declarations.

Functions connected with the DTD compiler: **Compile DTD**, **Remove DTD**, **View DTD**, **Edit Layout**

XML parser

This parser transforms an XML document into internal for the system DOM representation. During the parsing process the parser checks the well-formedness of the document and reports the corresponding errors.

The third component is the *Validator*. This module checks the validity of the document with respect to a DTD. Each document which is loaded in the system has to be attached to a DTD. Once a document is parsed to the internal representation of the system, it can be saved in this internal representation and the next time when it is used it will not be necessary to be parsed again. The Validator is active for the currently loaded document in the editor and when the user is modifies the document the Validator reports the changes in the validity of the document with pointers to the corresponding wrong elements of the document.

3.2 XML Editor

The access to the system is via a structure-driven editor which allows the user to edit and manipulate XML documents. Each loaded into the editor document is presented to the user in two or more views. One of these views reflects the tree structure of the document as described in the previous section. The other views of the document are textual. Each textual view shows the tags and the text content of the document. The tags in the textual view are separate elements from the rest of the text and can not be edited. The user has the possibility to attach to each textual view a filter which determines the tags and the content of which elements to be displayed in the view. This option allows the user to hide some of the information in the document and to concentrate on the rest of the information. With different textual views of the same document the user can attach different filters.

The editor supports a full set of editing operations as copy, cut, paste and so on. These operations are consistent with the XML structure of the document. Thus the user can copy or delete a whole subtree of the document. Some of these operations as search and replace are defined in terms of XPath expressions. This allows the user to search not only in textual content of the document but also with respect to the XML markup. The most powerful operation here is the XPath replace. This operation is used for various commands for restructuring the document. Generally, the scenario is the following: (1) a list of nodes (subtrees, text elements) is chosen by the Source XPath expression. In this way the elements which will be copied or moved in the document are defined; (2) a list of nodes is chosen by the Target XPath expression. In this way the place(s) where the source elements will be copied or moved are defined; (3) the elements from source list are attached to the elements of the target list. There are several options defining the way of performing of the above action. These concern such things as whether the elements of the source are copied or cut from the document before they are attached to the target, the mapping between the source and target elements - it is possible for the source elements to be attached to each element of the target, or each element of the source to be attached to the corresponding element of the target. Via different options this operation becomes very powerful means for document modification or entering new information in case the source is given not as an XPath expression but as a fragment of XML document or text. In future we plan to allow an evaluation of the source and the target expressions over different documents and thus to allow their merging.

The editor allows editing of the document textual content or its structure. The editing of the structure is supported by the attached to the document DTD. When the cursor is located at some point in the document structure, the user can enter a child, a sibling or a parent of the pointed element. In both cases the DTD is consulted and the list of the allowed for this position tags is offered to the user.

3.3 Document Transformation

The system offers a general mechanism for the transformation of some XML documents into other XML documents. This is done by implementing XSLT language - see [XSLT, 1999]. The transformations can be applied in two modes: globally and locally. When a transformation is applied globally it is applied to the whole document. In future we plan some transformation to be applied to a set of documents. When the user wants to apply a transformation locally he/she first selects an appropriate fragment of the document and then the transformation is applied only to this fragment. This last option provides a mechanism for construction of a set of transformations which the user applies depending on the current task and thus avoiding the necessity to write very specific conditions on the applicability of the transformation.

3.4 Additional Facilities

Besides the basic tools presented so far, there is a great number of additional facilities for supporting corpus development implemented in the CLaRK system:

Extractor

The Extractor is a tool for extracting elements from XML documents in a new document. For instance this can be used to extract all sentences in certain documents that meet some condition. The condition is stated as an XPath expression.

Remove tool

The Remove tool allows the user to delete some elements that meet a certain condition stated again as an XPath expression.

Sort

The Sorting tool is concerned with sorting elements of a document according to some keys defined over these elements. The sorting is defined in terms of two XPath expressions. The first expression determines which elements will be sorted. This expression is evaluated with respect to the root of the document as a context node. The second XPath expression defines the key for each element and it is evaluated for each node returned by the first XPath expression. The list of nodes returned by the first expression is sorted according to the keys of the nodes. Afterwards the nodes are returned in the document in the new order.

Concordancer

A concordance tool is implemented on the basis of the XPath engine, regular grammar engine and a sorting module. The concordance tool is useful for searching of some kind of units within some bigger units. For instance, a word with a sentence, a phrase within a paragraph and similar. The bigger element is called here a *context* and the smallest element is called *item*. The context is defined by an XPath expression. The item could be defined by an XPath expression or by a regular grammar. There are additional possibilities to restrict the context by regular grammars or XPath expressions. The found units are stored in a new document and presented to the user in a table format. The user could also to open the document as ordinary XML document and use all the tools available in the system in order to process further this document.

For example, in case of appropriately marked-up corpus one can extract all verbs and order them with respect to the first noun on the right hand side of the verb (not necessarily the first word on the right hand side).

4 Future development

The CLaRK system will be very intensively used within the BulTreeBank Project which just has started at the Linguistic Modelling Laboratory - see [Simov, Popova and Osenova, 2001] and [Simov et al., 2002]. We plan to extend the system in the following directions:

External programs. A mechanism for calling external programs which receive as input fragments of an XML document and returns also fragments of XML document. We envisage the actual communication to the external programs to be implemented via transformations of fragments of documents to and from special interface XML documents. In this way an external program will be declared within the system only once and the user will be able to use the program with XML documents with different structure.

Schemes of dependencies between elements in several documents. This is in connection with databases. We can consider each XML DTD as a conceptual scheme over XML documents. Then we can use a set of DTDs to describe interconnected schemes. We plan to implement support for such schemes. Because the task can prove to be very hard we will start with one basic DTD and auxiliary DTDs defining interconnections in table format.

We plan to extend the set of events and actions available to the user for defining the constraints. The target here will be a macro language for definitions of actions. Also we plan to make the constraints more active and as an activating event will be used the result from evaluation of some other constraint. In this way we will have mechanisms for propagation of information from one constraint to others.We also plan to add statistical facility for evaluating the quantity characteristic of the documents. The result of this facility will be a table of the relative frequency of some mark-up to some other mark-up in the document. Also we plan to add some other views over documents that are not naturally represented in textual or tree view of XML document such as graph view reflecting the ID references inside a document or other interpretations of the content of a document.

4.1 Acknowledgements

We would like to thank to Atanas Kiryakov and Marin Dimitrov from OntoText Lab (http://www.OntoText.com) for their participation in the initial stages of the development of the CLaRK System.

Also we would like to Zdravko Peev for his great work during early stages of the implementation of the system.

We would like to thank to our colleagues from BulTreeBank Project: Petya Osenova, Milena Slavcheva, Sia Kolkovska, Elisaveta Balabanova and Dimitar Dojkov for their patience during the use of the system in their work.

We would like to thank Petya Osenova and Gergana Popova for their help with the English.

We would like to thank Tylman Ule for his reading of the documentation and pointing us to some inconsistencies.

All errors in the text are ours of course.

References

[Abney, 1991] Steve Abney. 1991. *Parsing By Chunks.* In: *Robert Berwick, Steven Abney and Carol Tenny (eds.), Principle-Based Parsing.* Kluwer Academic Publishers, Dordrecht.

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

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

[DOM, 1998] 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

[Simov, Popova and Osenova, 2001] K. Simov, G. Popova, P. Osenova. 2001. *HPSG-based syntactic treebank of Bulgarian (BulTreeBank).* In: *Proceedings of Corpus linguistics 2001,* Lancaster, UK.

[Simov et al., 2002] K.Simov, P.Osenova, M.Slavcheva, S.Kolkovska, E.Balabanova, D.Doikoff, K.Ivanova, A.Simov, M.Kouylekov. 2002. *Building a Linguistically Interpreted Corpus of Bulgarian: the BulTreeBank.* In: Proceedings from the LREC conference, Canary Islands, Spain.

[TEI, 2001] Text Encoding Initiative. 1997. *Guidelines for Electronic Text Encoding and Interchange.* Sperberg-McQueen C.M., Burnard L (eds).

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

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

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