A Concise Query Language with Search and Transform Operations for Corpora with Multiple Levels of Annotation

Anil Kumar Singh

School of Computer Engg.
KIIT University, Bhubaneswar, India
nlprnd@gmail.com

Abstract

The usefulness of annotated corpora is greatly increased if there is an associated tool that can allow various kinds of operations to be performed in a simple way. Different kinds of annotation frameworks and many query languages for them have been proposed, including some to deal with multiple layers of annotation. We present here an easy to learn query language for a particular kind of annotation framework based on 'threaded trees', which are somewhere between the complete order of a tree and the anarchy of a graph. Through 'typed' threads, they can allow multiple levels of annotation in the same document. Our language has a simple, intuitive and concise syntax and high expressive power. It allows not only to search for complicated patterns with short queries but also allows data manipulation and specification of arbitrary return values. Many of the commonly used tasks that otherwise require writing programs, can be performed with one or more queries. We compare the language with some others and try to evaluate it.

Keywords: Corpus query language, Annotated corpus, Search and transform operations

1. Introduction

Representation of annotated corpora and mechanisms to access and manipulate data have been a major area of research in Natural Language Processing (NLP) during the last many years. These are difficult problems (even when considering only text), primarily because linguistic annotation can be of various kinds and at multiple levels, e.g. morphological, Part Of Speech (POS) tagging, chunking, phrase structure, dependency relations, semantic relations, discourse and dialog information etc. Merging all such annotation (Suderman and Ide, 2006) for some corpus in one file per document is perhaps not possible or it may not be feasible. However, many of these annotation levels can indeed be merged in one file per document. One of the ways to do this is through a formalism based on threaded trees (Larchevłque, 1995; Ait-Mokhtar et al., 2002), by having different kinds of threads for different kinds of annotation and putting constraints on the threads. The constraints ensure that the problem of 'chasing pointers' (Bird and Liberman, 2000) does not become a serious problem. In the absence of typed threads with well defined constraints, there can be arbitrary pointers that can make the representation as complex as any unconstrained graph.

One specific annotation scheme that uses threaded trees (Begum et al., 2008) encodes dependency trees and some other relations (such as co-reference) over chunked data by the use of constrained threads. It might be possible to translate many other annotation schemes into a threaded tree based scheme, but we leave that for future work, as here we will focus on the query language, not the annotation framework.

Though a lot of query languages have been proposed (Bird et al., 2000; Lai and Bird, 2004), we are not aware of any language for threaded tree based annotated data. Languages for linguistic trees have, however, been well studied and we will try to relate our language with a few of those.

We will first present a short review of the related work (Section-2). Then we briefly discuss how threaded trees can be used for encoding multiple layers of annotation (Section-3). After discussing the requirements of a query language for searching and transforming annotated data in Section-4, we present a brief overview of the syntax of the language in Section-5, followed by some examples (Section-6) and a description of the syntactic elements in Section-7. We then suggest some applications of the language (Section-8), before presenting a comparative assessment of the language and its limitations (Section-9). We also derive some directions for future work based on this.

2. Related Work

In this section we present a short review of related work reported in the literature under two headings: 1) annotation frameworks, and 2) query languages.

2.1. Annotation Frameworks

Annotation frameworks can be divided into two broad categories: graph based and tree based. A comparative study of many annotation frameworks was presented by Bird and Liberman (1998; 2000). Since their work was more focused on speech data, many of the frameworks considered were meant for such data, e.g. TIMIT, CHILDES and MATE. But they also considered some frameworks which are used more for text based data, such as the Penn Treebank corpus. One major framework that was not included in their study was the GATE annotation framework (Cunningham et al., 2002), which uses standoff format (a common way to allow multiple layers of annotation). After considering each of them, they proposed a formal framework for linguistic data that could be used for all those purposes for which these frameworks are used. They called the proposed framework an 'annotation graph'. The nodes in this graph were temporal points, while the edges represented the linguistic objects. They showed how even multiple layers of

annotation could be represented by different tiers of the annotation graph. Maeda et al. (2002) described how the Annotation Graph Toolkit could be used for creating tools for this framework.
As compared to graph based annotation frameworks, tree based frameworks are (for obvious reasons) much simpler. However, adding extra layers of annotation to tree based data is a non-trivial problem. One way is to store such extra information in a separate file and use node identifiers to link it to the data in the tree. Another way is to store the data in multiple trees stored in different files, which are somehow linked together. Yet another way is to use threaded trees, which is the one we will be assuming for our query language.
In one of the major works, Cotton and Bird (2002) had proposed an integrated framework for treebanks and multilayer annotations. This work focused more on tree based data, but it also suggested annotation graphs as the solution.

2.2. Query Languages

Bird et al. (2000) had compared some of the query languages available (at that time) for graph based annotation frameworks. These included Emu and the MATE query language. They then proposed their own query language for annotation graphs. This language used path patterns and abbreviatory devices to provide a convenient way to express a wide range of queries. This language also exploited the quasi-linearity of annotation graphs by partitioning the precedence relation to allow efficient temporal indexing of the graphs. Another such survey was by Lai and Bird (2004), where the authors considered TigerSearch, CorpusSearch, NiteQL, Tgrep2, Emu and LPath (Bird et al., 2005; Bird et al., 2006). From this study, the authors tried to derive the requirements that a good tree query language should satisfy.
Resnik and Elkiss (2005) had reported a search engine for linguists that was meant to be easy to use for linguists who were not versed in the use of computers. This tool allowed linguists to draw patterns in the form of sub-trees, which were then converted into queries and searched. Like almost all such languages, it did not allow manipulation of data and it worked only for certain levels of annotation. It was mainly aimed at searching phrase structure patterns and morphological information.
One of the well known query languages for annotated corpora used for linguistic studies and for NLP is the Corpus Query Language[1] (CQL), very different from the one we are presenting here. It is used in a popular tool called Sketch Engine[2] (Kilgarriff et al., 2004). It provides a wide variety of functionalities to access corpora, such as searching words, lemmas, roots, POS tags of a word, getting the left and right contexts upto a window size of 15.
Another usual practice is to have a query tool for syntactically annotated corpora such that the data is converted internally to relational database and the query is written using SQL (Kallmeyer, 2000). A much earlier work was titled 'A modular and flexible architecture for an integrated corpus query system' (Christ, 1994), which is used by the IMS Corpus Workbench[3]. Another query language called MQL is used in the Emdros database engine for analyzed or annotated text[4]. MQL is a descendant of QL (Doedens, 1994).
The language that we describe here is similar in some aspects to many of these languages, but different in others. The most important differences are the support for threaded trees, its very concise syntax, query-and-action mechanism (data manipulation), arbitrary return values, support for custom commands and the possibility for pipelining results through the source and destination operators. It also has high expressive power generally. Moreover, it can be used for purposes other than NLP because the data that it operates on is similar to the general XML representation and the language has **some** of the power of both XPath[5] based querying and XSLT[6] based transformation.

3. Threaded Trees and Multiple Layers of Annotation

There can be a different kind of 'annotation graph' where nodes are the linguistic objects and edges are relations (which could include hierarchical relations such as parent-child or dominated-by). Threaded trees (Larchevłque, 1995; Ait-Mokhtar et al., 2002) are a subset of this kind of annotation graphs. The base structure is a tree and threads are the crossing edges. If threads are typed and labelled, then each type can be used to represent one layer of annotation. The labels can be used to indicate annotated relations of that type. Depending on the requirements of the annotation framework, strong constraints can be applied to the threads. For example, we could have a constraint that says that the threads of a particular type are required to form a tree such that it has all the leaf nodes that the base tree has. Such threads can then be used to represent, say, dependency relations, assuming that the base tree represents the phrase structure (or just a POS tagged and chunked sentence). Other threads may have different kind of constraints and they can be used to encode, say, the argument structures or semantic relations.
By allowing the nodes to have feature structures associated with them, we can have deeper annotation of the properties of nodes as well as relations. In fact, threads themselves may be marked via the feature structures. We will assume here that these feature structures are sets of attribute-value pairs, but in an extended version, the values could be feature structures, thus allowing nested feature structures.
The Sanchay Corpus Query Language that we present here (now renamed Sandhaan) assumes that the data is in the above representation (Figure 1). The language allows the data to be searched, manipulated and extracted, irrespective of how many levels of annotations are stored in the threaded tree structure.
Threaded trees can be easily implemented using XML as the data format.

[1] http://www.fi.muni.cz/~thomas/corpora/CQL/
[2] http://www.sketchengine.co.uk/
[3] http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/
[4] http://emdros.org/mql.html
[5] http://www.w3.org/TR/xpath
[6] http://www.w3.org/TR/xslt

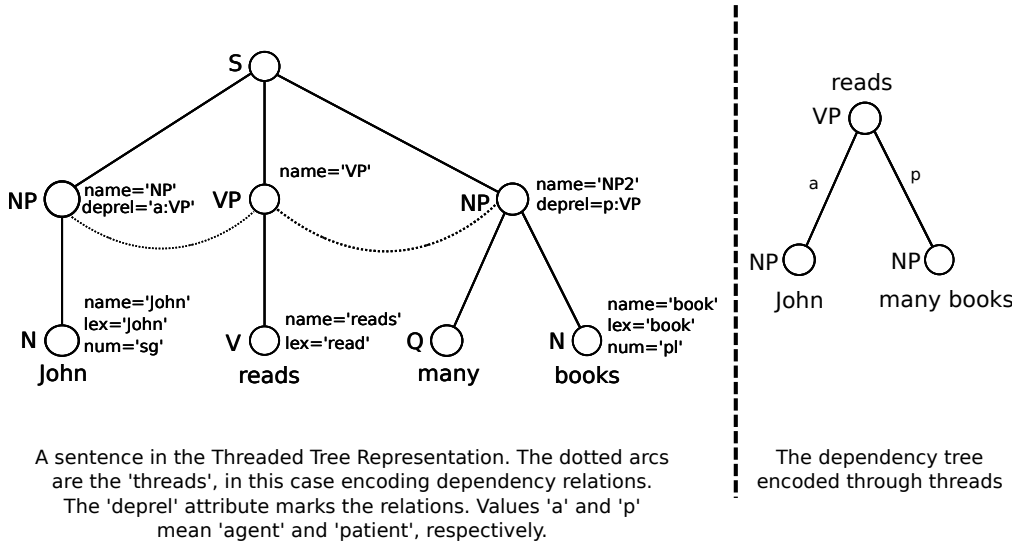

Figure 1: Threaded Tree Representation

Note that we are not following any specific linguistic formalism here and the examples given in this paper are only to demonstrate the syntax of the language. It is up to the users of the language to write linguistically significant queries for their own specific purposes.

4. Requirements of a Search and Transform Language

Lai and Bird (2004) had identified some requirements that a good tree query language should satisfy (in terms of expressive power). These can be summarized under four headings:

- **Tree Navigation**: This will allow operations for subtree matching, returning subtrees, reverse navigation (e.g. 'follows') and non-tree navigation (e.g. over terminals).
- **Closures**: Of basic relations such as dominance, precedence and sibling precedence, as well as of more complex relations such as self-recursive rules using a closure or closures involving more than one step, i.e., things which are expressible using XPath.
- **Beyond-Tree Navigation**: For searching beyond sentence boundaries or for searching over forests. Also, for querying non-tree structures (the threads in our case might form a non-tree structure).
- **Update**: Operations like insertion, deletion, moving and labelling of nodes, subject to the constraint that the base text is preserved.

Apart from these specific requirements, there are some general requirements that any programming or scripting language should satisfy. These include conciseness of the syntax, short learning curve, scope for efficient implementation and possibility of translation to other popular languages. In the following sections and especially in Section-9., we will discuss the degree to which our language meets these requirements.

5. An Overview of the Syntax

One way to express the motivation for developing this is as follows. Suppose there was an exhaustive API (Application Programming Interface) to process, search and manipulate the data in the Threaded Tree Representation[7] (Singh, 2008; Singh and Ambati, 2010). This API allows all kinds of possible operations on the data. Then the proposed query language should be able to allow the same range of operations on this kind of data, just by providing concise queries. What we present in this document is the first draft of such a language, so it does not cover this whole range of operations, but it does cover a fairly large part which would be the most useful for developers as well as other users.
A rudimentary version of the language described in this paper was earlier introduced as part of the Integrated Resource Access tool described by Singh and Ambati (2010). A new parser and query engine has been developed from scratch for this version and it has also been implemented as part of a platform independent open source system called Sanchay[8], which is being developed as a research and development platform for researchers working on languages. The rudimentary version of the language allowed only the simplest queries such as searching for words with a certain POS tag, whereas the language described here is a full-fledged, higly expressive, search-and-transform corpus query language.
A query in this language can consist of four parts, out of which only the second part (conditions) is mandatory:

1. **Sources (src)**: The documents or data streams on which the query has to be executed. (Optional)
2. **Conditions (cnd)**: The search conditions based on which the values will be returned and the actions (if any) will be taken
3. **Actions (act)**: The data manipulation operations which have to be performed on the nodes which

[7] An API of this kind is a part of Sannchay: http://sanchay.co.in
[8] `http://sanchay.co.in`

matched the search conditions. The operations could (optionally) simply specify return values. (Optional)

4. **Destinations (dst)**: The documents or data streams to which the results have to be stored or transferred. (Optional)

The source and the destination fields are optional because if the query language is being used from inside a tool (such as an annotation interface), then an annotated file might already be open and this file can become the implicit source, while the screen can become the implicit destination for the query results.
The the top level description of the syntax of a query would be:

$$[src\ =:]\ cnd\ [->\ act]\ [:=\ dst] \quad (1)$$

6. Example Queries

Table 1 (next page) presents a summary of the syntactic elements of the language (Section-7.). A simple example of a query is:

```
C.t='NN' -> C.t='Noun' and A
```

The above query has only two parts: the conditions (provided on the Left Hand Side or LHS) and the actions (provided on the Right Hand Side or RHS). This query will replace all the 'NN' tags of the tree nodes in the document with the 'Noun' tag and return the parent node (the first ancestor, A[1] or just A). **C** here represents the current node, **t** represents the tag, **.** is the dot operator, **=** is the value assignment operator and **–>** is the action operator. If we want the action to be applied only on the leaf nodes (which will usually hold the actual tokens with some lexical data), we can write:

```
C.t='NN' and C.f='t' -> C.t='Noun'
```

The previous two queries use one of the logical operators (and/or) and the literal value 't', which means 'true'. If we add the source and destination to the query (leaving out the terminal node condition to save space), it will be:

```
xml:src.txt:UTF-8 =: C.t='NN' \
  -> C.t='Noun' := xml:tgt.txt:UTF-8
```

Two more operators are introduced in the above query: the source (**=:**) and the destination (**:=** operators). If, instead of an action, we want the query to return the current, the previous and the next node, then the query would be:

```
xml:src.txt:UTF-8 =: C.t='NN' \
  -> C and P and N := raw:tgt.txt:UTF-8
```

How the result is displayed or stored will depend on the format specified (raw, i.e., simple text, in the above query and xml in the preceding query) as well as on the implementation, e.g. how exactly the multiple values are added to the destination. The current version of the language does provide some control for this through the process of concatenation:

```
C.t='NN'-> C.l+'-'+C.t+'; \
  '+P.l+'-'+P.t';'+N.l+'-'+N.t';'/r
```

The alias (**/**) operator here assigns an alias (a name or a key), viz. r, to the return value. Another new operator above is the concatenation (**+**) operator.
The present implementation will put the concatenation of values on the RHS of the above query on one line (assuming simple text output), preceded by the alias for the return value (if given) or the query term representing the return value followed by ': '. Alternatively, we could write:

```
C.t='NN'-> C.l+'-'+C.t/c \
  and P.l+'-'+P.t/p and N.l+'-'+N.t/n
```

For the above query, the current implementation would put the three return values on three separate lines, each preceded by the respective alias followed by ': '.
One very important feature is that the LHS can be prefixed by a directive such as "TT['deprel']:" that will 'extract' the tree encoded by the thread of type 'deprel' and then all the usual tree operations that can be performed on the base tree can also be performed on this tree (in this case a dependency tree):

```
TT['deprel']: C.t='NP' AND A.t='VP'
```

7. Syntactic Elements

The syntactic elements of the language can be divided into the following categories: objects and members, operators and values, wildcards and ranges, source and destinations, actions and return values, and custom commands. Together, they provide the expressive power that allows the language to fulfil most of the requirements mentioned in Section-4..

7.1. Objects and Members

Objects are the tree nodes or larger (e.g. S or sentence and F or file/document) units of linguistic data on which the queries operate. Query processing (in most cases) happens node-by-node and the centre of this processing is the C or the Current node. For navigation, all the node addresses are based on the current node. Thus, we can have a previous node (P), a next node (N), an ancestor node (A), a descendant node (D), a referred node (R) and a referring node (T), the last two meant for threads. While P is for precedence, there is corresponding node Pr for sibling precedence. Similarly, there is a corresponding node Nx for N. There is another special node (M) to represent the nodes that matched a query condition. Most of the node types (except C) can have multiple candidates for matching, which are specified through integer indices (for P, N, Pr, Nx and D, e.g. N[2] for the next to next node), through string keys (for R, e.g. R['deprel']) or through a string key and an integer key (for T, e.g. T['deprel':2]). The indices are enclosed inside square brackets. The string keys are enclosed in single quotes, except when the key is an alias because aliases are not evaluated whereas other keys can be evaluated as they can consist of variables. Any value (including node indices) is treated as a variable and is evaluated if it is not enclosed within single quotes.

Objects	
Object	**Remarks**
F and **S**	The File (document) and the Sentence
C	The Current node (the centre of the node-by-node query processing)
P and **N**	The Previous and the Next nodes
Pr and **Nx**	The Previous sibling and the Next sibling nodes
A and **D**	The Ancestor (or parent) and the Descendant (or child) nodes
R and **T**	The Referred and the Referring nodes (thread navigation)
M	The node(s) that matched one of the conditions, e.g. M[p], p being the condition alias
Members	
Member	**Remarks**
l	The lexical data for the node
t	The tag (e.g. POS tag) of the node
a	The attribute, with the index specified within square brackets, e.g. a['lex']
v	The level (distance from the root) of a tree node (0 being the root)
f	Boolean value to check if a node is a leaf node
Operators and Values	
Operator/Value	**Remarks**
AND	Conjunction of two or more search conditions
OR	Disjunction of two or more search conditions
()	Parenthesis: Grouping of search conditions for evaluation or nesting
[]	Index: Integer (position) or string (name or alias), e.g. D[2], a['deprel'] etc.
:	Index qualifier, e.g. D[2:3] (grandchild's third child)
.	The dot operator to access the members of an object and to form node addresses
‘’	The literal value specified within single quotes (e.g. 'agent'), usually of members
+	Concatenation: To join together two or more literal values or variables
= and !=	Equal and Not Equal (LHS), based on exact equality of values
∼ and !∼	Similar and Not Similar (LHS), based on similarity, e.g. using regex
=	Value assignment operator (RHS)
–>	Action to be performed on the nodes that matched the conditions
=:	The sources of the data, e.g. the corpus files
:=	The destinations, e.g. the files where the results have to be stored
/	Alias assignment for conditions, return values and sources/destinations
Wildcards and Ranges	
Wildcard/Range	**Remarks**
?	The first node to match
.	The last node to match
*****	Any nodes to match (disjunction)
@	All node(s) that match(es), e.g. N[@], M[@] (conjunction)
0	None (normal indices start from 1)
–	The range of nodes, e.g. N[2-4], P[3-], D[-2] etc. and z is the last node.

Table 1: A Summary of the Query Language

Compound node addresses can be created using the dot operator, e.g. N.A[2].P[3], which will mean the preceding node at a distance of 3 (P[3]) of the grandparent node (A[2]) of the next node (N). Variables can be either node addresses, or member values (e.g. N.t) or a combination of variables and literals formed by using the concatenation operator (e.g. N.t+'-'+N.l).

The four members currently implemented are: **l** or the lexical data, **t** or the tag, **v** or the level in the tree and the boolean member **f** to check if a node is a leaf node. While the object symbols are written in capital letters, the members symbols are written in small letters.

7.2. Operators and Values

Some of these have already been introduced in the preceding section. Here we will add some more information about them. The dot operator (**.**) allows us to form node addresses or to access the members of the nodes. The comparison operators (used on the LHS) currently provided are: equality (**=**), inequality (**!=**), similarity (∼) and not-similarity (**!**∼). The value assignment operator reuses the symbol (**=**) on the RHS. The LHS and the RHS are separated by the action operator (**–>**). The alias assignment operator (**/**) provides an easy way to write concise queries because we can give single character aliases, apart from allowing (more readable) access to objects previously mentioned in the query. The concatenation operator (**+**) has already been explained. Parentheses (**(...)**) are used to group together query conditions for prioritized evaluation and to form nested queries. The logical operators AND, OR and NOT (written as !(...)) operators are also supported to form complex queries.

7.3. Wildcards and Ranges

In many cases, it can be very useful to be able to specify wildcards when there can be multiple candidates (the closure requirement of Section-4.). The language provides three wildcards and also ranges in terms of integer indices:

- **?**: The first one to match, e.g. N[?]
- **.**: The last one to match, e.g. N[.]
- *****: Any node(s) that match(es), e.g. N[*], M[*] (disjunction)
- **@**: All node(s) that match(es), e.g. N[@], M[@] (conjunction)
- **0**: None (normal indices start from 1)
- **-**: From the first index to the second index, e.g. P[2-4], P[2-], P[-2]. The last node is specified by the special index z.

Here is one example of using wildcards, aliases and concatenation:

```
P[*].t/p='XC' and C.t!='XC' \
 -> M[p:*].t=C.t+'C'
```

Suppose there are tag sequences of the form XC XC NN, XC XC JJ etc. and they have to be converted to NNC NNC NN and JJC JJC JJ, respectively, then the above query will do that.

7.4. Sources and Destinations

In the current implementation, the sources and destinations can only be files, but they could, in principle, be streams too, e.g. for building pipelines of queries. In the case of files, a source or a destination can be specified in terms of four parts:

- **Format**: Could be simple text or XML or something else, depending on the implementation
- **Location**: The URL or the URI or the path of the document or file
- **Charset**: The charset or encoding of the document (the default is UTF-8)
- **Name**: The object alias

An example of a source specification is:

```
xml:src.txt:UTF-8/s
```

In the above query, xml is the format, src.txt is the location, UTF-8 is the charset and s is the name or the alias. Aliases allow multiple sources and destinations to be specified and also accessed from other parts of the query. For example the following query uses two sources:

```
xml:src1.txt:UTF-8/s1 \
  and xml:src2.txt:UTF-8/s2 \
  =: F[s1].C.t='NN' and F[s2].C.t='Noun'
```

7.5. Actions and Return Values

Multiple transformations can be performed by using the AND operator on the RHS (Right Hand Side), including on nodes other than the current node by using the same notation as for the LHS.

On the RHS, we can also specify return values by using the node symbols and the dot notation (e.g., C, $N.A$). For this purpose, another symbol S can be used to return sentences for the nodes which match. The syntax is intuitive and easy to remember. If we don't provide an assignment value, then the node address, variable or the concatenated value is treated as the return value (e.g. $N.l$ will return the next node's lexical data, whereas N will return the next node).

If an assignment expression has nodes on both sides, it will be interpreted as a node insertion, deletion or move operation. For example, $Nx = A.N.D$ will take the next sibling node and move it so that it is dominated by the node which is next to the parent of the current node. At present there is no way to ensure that base text is preserved (the user is expected to ensure that), but we will introduce a mechanism for this in a future version.

7.6. Creating and Navigating Threads

Threads (which represent multiple layers of annotation) can be created by providing a query like the following:

```
C.l='reads' AND C.f='t' \
   AND A.N[?].t/q='NP' \
  -> M[q].a['deprel']=''a':A.a['name']'
```

This query will look for a leaf node with the lexical data 'reads' such that its parent node is followed by an NP (the first one encountered). Then, according to the action specified on the RHS, it will create a thread from this NP to the parent node (VP) of 'reads' by adding a value for the $deprel$ attribute as a concatenation of the relation label ('a' or agent) and the unique name of the VP, separated by a colon.

To navigate threads, the node symbols R (referred node) and T (referring node) can be used. For the dependency tree in the example given in Figure 1, VP is the referred node and the two NPs are the referring nodes. Thus the following query:

```
C.t='NP' AND R['deprel'].t='VP'
```

will find all the NPs for which the referred node is a VP. Note that we may have to specify the attribute used for the kind of thread we are searching. The most commonly used of these attributes could be used as a default, so that there is no need to specify it. Since there can be more than one referring nodes, we need to specify the index as well in the case of such nodes:

```
C.t='VP' AND T['deprel':2].t='NP'
```

The above query will search for all the VPs whose second referring node (e.g. the second argument) is an NP.

7.7. Commands

The language also allows us to specify commands to be executed on the data. For example, if we want to ensure that all the nodes have unique names before we start providing transformation actions, we can give a command like:

```
reallocateNames
```

where 'names' are the unique node identifiers which are used for marking and navigating threads.
Note that the language supports custom commands, so there is no exhaustive list of commands. The current implementation has some commands that have been found useful so far, but more can be easily added. Commands can also be executed subject to some conditions, i.e., if we write a query with LHS giving the conditions and the RHS giving the command:

```
C.a['name']='' -> reallocateNames
```

This query will ensure that the nodes have unique names when we start marking threads for any kind of extra annotation layer.

8. Applications

Only a few examples have been given in the preceding sections. The query language allows many other kinds of operations using the constructs that have been mentioned. These operations can be used for many purposes, apart from searching. Queries can be written to perform sanity checks on the annotated data. They can be used to automatically mark information that is very predictable (to reduce manual work). They can be used to bring the old annotated data in tune with the new annotation specifications without having to write programs for that purpose, even if the changes require more complex operations than simple global replacement, as the XC XC NN example given above shows. Queries can also be written to easily extract complex features for, say, machine learning algorithms. They can be used to make the task of an annotation adjudicator easier.

9. Comparative Assessment

Lai and Bird (2004) had used seven syntactic queries to compare various tree query languages. These are given in Table 2. The evaluation criterion was at least two fold. First, how many of these queries are expressible in a language. Second, how concise are those queries.
All seven of these queries can be expressed in our language, but a feature required for the fifth query ('@' index for all nodes to match: conjunction, rather than disjunction, as expressed by the '*' index) has not yet been fully implemented. These queries are given in Table 3.
Only NiteQL could express all these queries, others (TigerSearch, Emu, CorpusSearch, Tgrep2 and LPath) could not express at least one of these seven queries. However, NiteQL does not have a concise syntax. Thus, in terms of expressive power, our language compares favourably with these languages. It also has comparable conciseness of syntax. Moreover, there are a range of queries that cannot be expressed in other languages because they are not meant for threaded trees and have no equivalents to the R and T nodes that we have.

Q1.	Find sentences that include the word 'saw'.
Q2.	Find sentences that do not include the word 'saw'.
Q3.	Find noun phrases whose rightmost child is a noun.
Q4.	Find verb phrases that contain a verb immediately followed by a noun phrase that is immediately followed by a prepositional phrase.
Q5.	Find the first common ancestor of sequences of a noun phrase followed by a verb phrase.
Q6.	Find a noun phrase which dominates a word 'dark' that is dominated by an intermediate phrase that bears an L-tone.
Q7.	Find an noun phrase dominated by a verb phrase. Return the subtree dominated by that noun phrase only.

Table 2: Syntactic Queries for Comparing Tree Query Languages (Lai and Bird, 2004)

Q1.	C.l='saw' $-$ > S
Q2.	C.l='0' AND C.D[*:0].l/p='saw' $-$ > S
Q3.	C.t='NP' AND C.D[z].t='NN'
Q4.	C.t='VP' AND C.D[*].t 'V*'/p AND M[p].N.t='NP' AND M[p].N[2].t='NP'
Q5.	P[*].t/p='NP' and C.t='VP' AND M[p:@].A[*]=C.A[*]/q $-$ > M[q:1]
Q6.	C.t='NP' AND D[*].l='dark'/p AND M[p].A[*].a['tone']='LTone'/q AND C.l>M[q].l
Q7.	C.t='NP' AND C.A[*].t='VP'

Table 3: Comparison Queries in Our Language

But a look at the queries in Table 3 also shows some scope for improvement. For example, there is need for 'dominates' and 'is-dominated-by' operators which will make these queries even more concise. As our language was developed initially for annotators working on data that was not huge in quantity, there seem to be some problems from the efficiency point of view too.
Other directions for future work include a study of the formal properties of the language and a more rigorous evaluation based on various criteria. We are also developing a graphical query designer for this language.
Even in the present condition, the language is being used by many annotators and annotation adjudicators to make their work easier, as well as by a few developers to build substitutes for previously used programs for tasks like sanity checks, validation (Agarwal et al., 2012) etc. Users with non-computational background have found it easy to learn at least the basics of it, though we have not yet performed a proper evaluation of the ease of learning. Our non-empirical experience is that power users can learn it within a few hours, or even less in some cases.

10. Conclusion

We presented a concise yet expressive query language for data that is in a tree-like format such that one node can have links ('threads') to any other node in the tree, allowing for additional trees or graphs to be encoded in the core tree. The language uses simple elements and constructs

like objects, members, operators, variables, values, nesting, aliases, wildcards etc. to allow writing queries that can perform fairly complex operations without the need to write programs for this purpose. Multiple conditions can be given, multiple transformation actions can be specified, multiple return values can be specified and so can be multiple sources and destination. The language can, in fact, be used as a scripting language for annotated data with multiple levels of annotation, where multiple levels are encoded through 'typed' threads. We compared the language to some other query languages and suggested some directions for future work.

11. References

Rahul Agarwal, Bharat Ram Ambati, and Anil Kumar Singh. 2012. A GUI to Detect and Correct Errors in Hindi Dependency Treebank. In *The 8th International Conference on Language Resources and Evaluation (LREC)*, Istanbul, Turkey, May. The European Language Resources Association (ELRA).

S. Ait-Mokhtar, J.P. Chanod, and C. Roux. 2002. Robustness beyond shallowness: incremental deep parsing. *Natural Language Engineering*, 8(2-3):121–144, January.

R. Begum, S. Husain, A. Dhwaj, D.M. Sharma, L. Bai, and R. Sangal. 2008. Dependency annotation scheme for Indian languages. *Proceedings of IJCNLP-2008*.

Steven Bird and Mark Liberman. 1998. Towards a formal framework for linguistic annotations. In *Proceedings of the International Conference on Spoken Language Processing*, Sydney.

Steven Bird and Mark Liberman. 2000. A formal framework for linguistic annotation. *Speech Communication*, 33(1,2):23–60.

Steven Bird, Peter Buneman, and Wang-Chiew Tan. 2000. Towards a query language for annotation graphs. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 807–814, Athens, Greece.

Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. 2005. Extending xpath to support linguistic queries. In *Proceedings of Programming Language Technologies for XML*, pages 35–46, Long Beach, California. Association for Computing Machinery.

Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. 2006. Designing and evaluating an xpath dialect for linguistic queries. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 52–61, Atlanta, USA.

Oli Christ. 1994. A modular and flexible architecture for an integrated corpus query system. In *Proceedings of COMPLEX'94*, Budapest.

Scott Cotton and Steven Bird. 2002. An integrated framework for treebanks and multilayer annotations. In *Proceedings of the Third International Conference on Language Resources and Evaluation*, pages 1670–1677, Las Palmas, Spain.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the ACL*.

Crist-Jan Doedens. 1994. *Text Databases. One Database Model and Several Retrieval Languages (Language and Computers)*. Editions Rodopi Amsterdam, Amsterdam and Atlanta.

Laura Kallmeyer. 2000. A query tool for syntactically annotated corpora. In *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora*, pages 190–198.

Adam Kilgarriff, Pavel Rychly, Pavel Smrz, and David Tugwell. 2004. A query tool for syntactically annotated corpora. In *Proceedings of Euralex*, pages 105–116.

Catherine Lai and Steven Bird. 2004. Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasian Language Technology Workshop*, pages 139–146, Sydney, Australia.

J.M. Larchevłque. 1995. Optimal incremental parsing. *ACM Transactions on Programing Languages and Systems*, 17(1):1–15, January.

Kazuaki Maeda, Steven Bird, Xiaoyi Ma, and Haejoong Lee. 2002. Creating annotation tools with the annotation graph toolkit. In *Proceedings of the Third International Conference on Language Resources and Evaluation*, pages 1914–1921, Las Palmas, Spain.

Philip Resnik and Aaron Elkiss. 2005. The linguist's search engine: an overview. In *ACL '05: Proceedings of the ACL 2005 on Interactive poster and demonstration sessions*, pages 33–36, Morristown, NJ, USA. Association for Computational Linguistics.

Anil Kumar Singh and Bharat Ambati. 2010. An integrated digital tool for accessing language resources. In *The Seventh International Conference on Language Resources and Evaluation (LREC)*, Malta, May. The European Language Resources Association (ELRA).

Anil Kumar Singh. 2008. A mechanism to provide language-encoding support and an nlp friendly editor. In *Proceedings of the Third International Joint Conference on Natural Language Processing (IJCNLP)*, Hyderabad, India.

Keith Suderman and Nancy Ide. 2006. Layering and merging linguistic annotations. In *Proceedings of the 5th Workshop on NLP and XML: Multi-Dimensional Markup in Natural Language Processing*, pages 89–92, Trento, Italy.