Oraculum, a System for Complex Linguistic Queries

Vladimír Ljubopytnov, Petr Němec, Michaela Pilátová, Jakub Reschke, and Jan Stuchl

Center for Computational Linguistics, MFF UK, Prague
{s.ljubo|s.nemec|s.pilat|s.reschke|s.stuchl}@ufal.mff.cuni.cz

Abstract. A system for complex linguistic queries is proposed. Data of multiple resources are converted into uniform representation of logical programming language predicates. Query is expressed as the programming language code, too.
Users may use the interactive web interface to generate the query code automatically and to view the result in the desired form.
The system database presently contains the set of tectogramatically annotated data of the Prague Dependency Treebank [1, 6] and part of the German corpus Negra [4].

1 Introduction

Nowadays, there are several annotated language corpora available worldwide containing linguistic data of various nature. For complex linguistic query purposes it would be advantageous to engage a system that stores the data of these resources in uniform way and allows to query them via appropriate user interface. The user should then be able to search for various linguistic phenomena in the corpora involved, even across these resources.

In this paper we present a linguistic query system, Oraculum. Oraculum is able to perform query operations on the corpora it encapsulates.

Data of each corpus involved are converted into uniform representation of logical programming language predicates. Above these stem predicates there is a set of extending predicates[1] designed for each corpus. These perform expected types of query operations and the user can combine them in his/her particular query.

There are several possibilities to form a query, most of them connected with the interactive web interface (which has been made available at http://slunicko.ms.mff.cuni.cz). The user can generate the query code via graphical interface, using a specially designed query language or write it directly in the internal logical language predicate form. It is possible to combine the methods in one particular query.

[1] In the paper we will often refer to a predicate in its procedural rather than the usual declarative sense.

Mária Bieliková (Ed.): SOFSEM 2002 Student Research Forum, pp. 27–34, 2002.

Oraculum presently works over the set of tectogramatically annotated data of the Prague Dependency Treebank [1, 6] and the part of German corpus Negra [4], which has been automatically transformed into tectogrammatic trees [8]. Other tectogramatically annotated corpora as well as other selected resources from EuroWordNet [7] will be incorporated into the system soon.

The goal of the project is to create a fully functional application that can be used by the community of linguists.

In this paper we aim to describe the system in greater details. Section 2 gives the system overview in form of data flow diagram. Section 3 describes the predicate set structure with respect to PDT tectogrammatical data. Sections 4 and 5 briefly describe the query mechanism together with the user interface functionality. Sections 6 gives a detailed query example and section 7 summarizes the crucial predicate performance test results. Finally, in section 8 conclusions and future work are pointed out.

2 System overview

The data flow diagram (Figure 1) gives an overview of the system functionality. The selected linguistic resources are automatically converted into predicate form and stored in the database. Users set the queries via the web interface and the code performs the operation on the database. Results are then displayed by the interface.

3 The Predicate Set

The set of predicates for the given corpus can be basically divided into three layers:

1. The *stem layer* representing the sole data of the original resource.
2. The *extending layer* consisting of predicates performing basic operations on the stem layer data regardless of their semantics.
3. The *language layer* made up of predicates that perform the expected types of linguistic queries. This layer represent the high-level database interface.

User defined predicates (forming queries) can be though of as a separate layer.

The stem layer predicates of the respective resources are visible for all the extending and language layer predicates. That allows to perform (if it makes sense) the designed operation on any data included.

3.1 The Representation of a Tectogrammatic Tree

The structure of tectogrammatic tree (TGTS [2]) representing the deep structure of a sentence is that of an ordinary tree. Unlike the regular tree, however, the order of daughter nodes is fixed and these are furthermore divided into left and

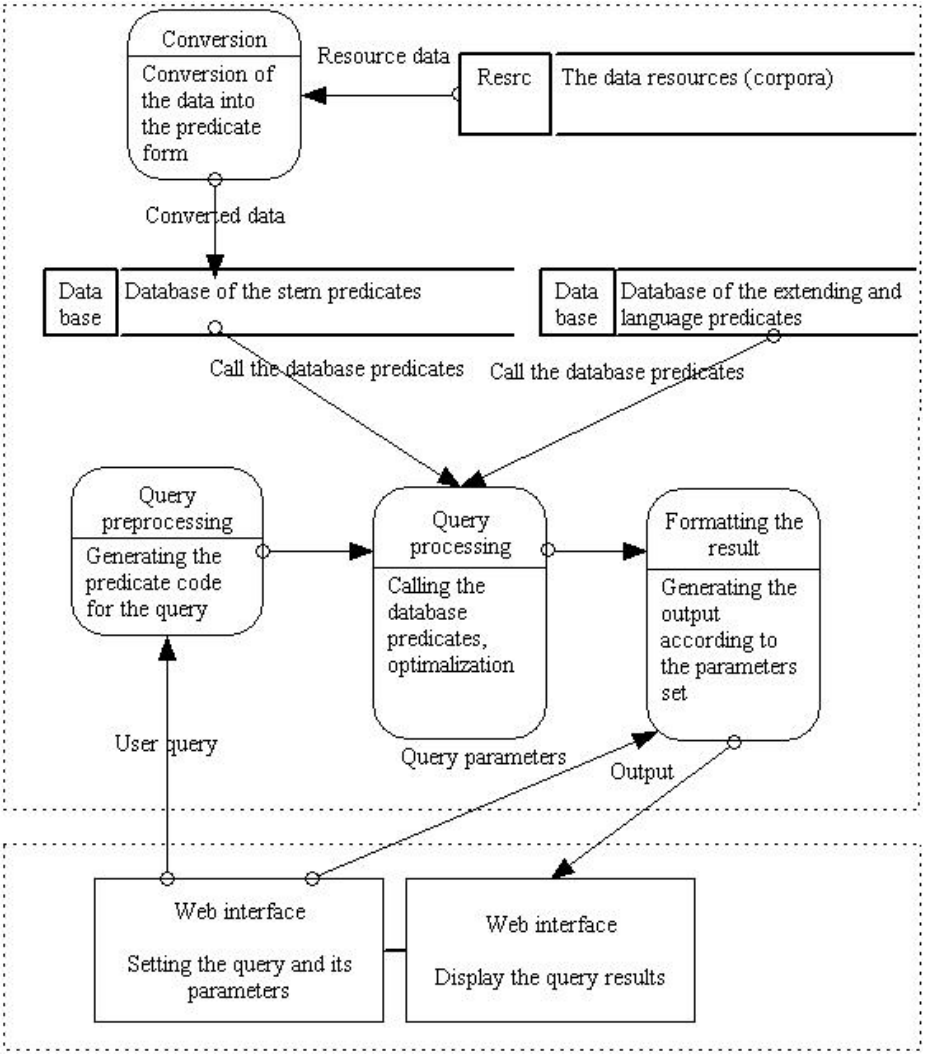

Fig. 1. Data Flow Diagram

right daughter nodes respective to their parent node. Each node of a TGTS is labelled by given set of attribute - value pairs. The following set of stem predicates has been designed to represent a TGTS[2]:

- node(Tree_id,Node_id,Parent_node_id,List_of_children_nodes) represents a node of a tree
- attribute(Name,Tree_id,Node_id,Value) specifies the value for an attribute of the given node
- type(Attribute_name,Type_of_value) determines the type of attribute value

3.2 The Extending and Linguistic Layer for a Tectogrammatic Corpora

Although many predicates have been designed to operate over the stem representation, most of them serve only as a basis for a small number of much more general ones. Only the most important predicates of the extending and linguistic layer are presented below:

STRUCT (in combination with the path predicate) is the universal search predicate. It allows to define a subgraph structure to be searched for in great detail. Nodes in a subgraph may be restricted by a rich system of conditions. These include logical expressions the node attributes must fulfill, the restrictions imposed on presence of other nodes surrounding the searched subgraph etc. (See section 6 for an example.)

STAT performs various statistical operations over the query (it counts the percentage of the matching structures etc.)

PATH checks whether the path in the tree between two specified nodes satisfy the given conditions. Together with the STRUCT predicate it represents the essential searching predicate equipment.

SENTENCE_DEPENDENCY_TREE creates a sentence dependency tree (nodes represent the sentences and edges the respective dependencies) for the given complex sentence

3.3 Advantages over NetGraph Query Tool

To query the PDT data, NetGraph tool has been created [5]. Until the Oraculum system has been developed, NetGraph was the only application available for this purpose. NetGraph is a client-server application, which allows the user to browse and view on-line analytic and tectogramatic trees.

[2] An abbreviated form of the predicate arguments is presented, the actual form of the predicates contains few more technical details.

As NetGraph has been designed for simple query purposes, it has limited capabilities in comparison with Oraculum PDT query predicates. It only allows to define a tree subgraph together with simple conditions for node labelling to be searched for and display the matching trees. There is no possibility to perform any other operation on the corpus.

It is also not possible to process the result further or to form a complex query as there are no logical variables or language designed for this purpose. Moreover, the sole subgraph structure cannot be defined in such detailed and flexible way as the Oraculum logical programming engine together with the searching predicates offers.

4 Query

Every query in Oraculum is either written in the internal logical language (Mercury [3]) form or is converted to that form by the user interface preprocessor. A query is therefore just a user defined predicate that is temporary linked with the main predicate set. As soon as the query is completed, the query predicate is retracted from the database (memory). The logical programming language approach allows the user to combine the built-in predicates in a complex logical expression (which may involve negation, for instance) that forms the resulting query.

4.1 Data types

For the tectogrammatical data, the result of a predicate may be the one and only one of the following four data types (at the logical language level, all of them except for the last one are list types):

- set of scalars
- set of node attributes
- set of nodes (possibly forming a tree)
- fail or proved status (no data are returned)

The return type is explicitly stated for each predicate. The query code may contain variables representing different data types, but only variables of one type may be selected to form the query result, which is then properly displayed by the user interface.

4.2 Caching and Optimization

Multiple user access, large data volumes and mainly exponential computing complexities of some operations (predicate STRUCT, for instance) require the overall optimization of the system.

Currently, caching of the STRUCT predicate on the user interface level is implemented. The results of queries and subqueries are stored and before an attempt to call a time consuming predicate is made the cache is searched for the potential result.

Moreover, the Mercury compiler itself performs a lot of code optimization [5]. In the future, an automatic optimizing procedure based on the corpus semantics will be implemented.

5 User Interface

Web user interface serves two main purposes: to help the user to write a query code and to display the results in the appropriate form.

5.1 Forming a Query

There are three ways to form a query: to write the pure logical language code, to use predefined macros to generate parts of the code, or to design the query via graphic interface. It is possible to combine these approaches in one query.

Graphic interface is mostly used to generate the STRUCT and PATH predicate code, which is otherwise quite uncomfortable to write. The user visually creates the structure to be searched for and sets the desired conditions via set of dialog windows. The code is then generated automatically and can be further modified and extended by the user.

5.2 Result Formatting

As far as the result formatting is concerned, Oraculum web interface offers either textual or graphical output of the result. Any query output variable can be displayed as a text regardless of its semantics. Special structures (such as trees or parts of trees) can be displayed graphically and exported to a defined format (XML, for instance) if requested.

6 Example

The usage of the search predicates STRUCT and PATH is demonstrated. The following code[3] finds all the tectogrammatic trees, whose head clause is a verb having either "agens" or "patiens" valency actant and an actant, whose morphological tag is not the same as of some descendant of the "agens"/"patiens" actant :

```
query([],[]).

query([Tree|Trees],Output) :-
     (struct(Tree,
       [[x, central, [left-any-eq-y, y-any-eq-z, z-any-eq-right],
           [y-z], [('tag','V*')]],
       [y, [left-any-eq-right], [('afun','agens'),(or),
```

[3] The code has been abbreviated and the syntax has been altered for sake of simplicity.

```
    ('afun','patiens')]],
[z, [left-any-eq-right], [('tag',V)]], Matching_struct ]),
not ( struct([u, ('tag',V)]),
      path(u, y, ['vu',(1,INF)])
    )
-> Output = [Tree, NextTrees] ; Output = [NextTrees]),
query(Trees, NextTrees).
```

Here, [Tree—Trees] is the input set of trees and Output is the resulting set. The first STRUCT predicate defines a subgraph of three nodes x,y,z. x is defined to be a central node and a verb (matching regular expression 'V*', it has to have at least two daughter nodes being the given actants (the first one is specified by the 'afun' attributte, the second one matches any daughter node, its morphological tag afterwards unifies with the V variable). Both daughters are mutually interchangeable and any number of other daughters may be present. The second struct predicate call is present in the negated clause and finds all the nodes with the V morphological tag. The successive path predicate call succeeds only for a descendant of the first actant. Each tree is searched for the occurrence of such structure and is added to the Output list on success.

The code is apparently quite cumbersome and hard to construct for a non-programmer. The query language allows to avoid predicate definition and recursion call by placing preprocessor directive in front of struct predicate call and the rest of the code is automatically generated.

7 Benchmark

The performance of exponentially complex STRUCT predicate has been tested in order to confirm acceptable running time of the query engine. The benchmarks were run on an Intel Pentium III 1 GHz processor machine with 256 megabytes of memory running Windows 2000.

The input data set consists of 5 query types (A,B,C,D,E) of descending complexity. The class A input represents the most complex query (the task was to find all paths -dependencies of length 3), the class E input represents the least complex query (to find all nodes representing adjectives).

Similarly, the corpus data are divided into 3 sets. Set 1 consists of large trees (above 20 nodes), set 2 of medium size trees (20 to 10 nodes) and set 3 of small trees (less than 10 nodes).

Each query type has been performed on the 3 groups. Table 1 shows the average response times for each query type and group.

Because the stem layer predicate set contains thousands of trees, the measured response times show acceptable performance of the crucial STRUCT predicate.

Table 1. The average response times for one tree of the given set in miliseconds.

	Query A	**Query B**	**Query C**	**Query D**	**Query E**
Set 1	0,447	0,197	0,082	0,064	0,053
Set 2	0,145	0,072	0,034	0,022	0,021
Set 3	0,011	0,030	0,013	0,008	0.006

8 Conclusions

The Oraculum query system has been presented. The logical programming background of the system offers great possibilities of querying the data of various corpora. On the other hand, this approach may be more difficult for a user without programming experience, although the system offers as much help as possible.

The performance of crucial part of the system has been tested and the response times showed to be acceptable.

As future work, new data resources will be incorporated to the system and the optimizing procedures will be enhanced. The set of predicates will also be enlarged.

References

1. Hajič J., Hajičová E., Hladká B., Holub M., Pajas P., Řezníčková V., Sgall P.: The Current Status of Prague Dependency Treebank. In: Text, Speech and Dialogue, ed. by V. Matoušek, P. Mautner, R. Mouček, K. Taušer, Plzeň: University of West Bohemia (2001) 11-20
2. Hajičová E., Panevová J., Sgall P.: A Manual for Tectogrammatic Tagging of the Prague Dependency Treebank. ÚFAL/CKL Technical Report TR-2000-09, Charles University, Czech Republic (2000)
3. Mercury programming language homepage: http://www.cs.mu.oz.au/research/mercury
4. Negra Corpus on-line resources: http://www.coli.uni-sb.de/sfb378/negra-corpus
5. NetGraph main page: http://shadow.ms.mff.cuni.cz/~mirovsky/netgraph/indexEn.html
6. PDT on-line resources: http://ufal.ms.mff.cuni.cz/pdt
7. Vossen P. (eds.) EuroWordNet: a multilingual database with lexical semantic networks for European Languages. Kluwer Academic Publishers, Dordrecht, 2002.
8. Zabokrtsky,Z. Transforming Negra-Millenium Treebank into (Praguian) Tectogrammatic Tree Structures: http://ckl.mff.cuni.cz/~zabokrtsky/saarbrucken2002/negra2tgts.ps