Towards a Simple and Full-Featured Treebank Query Language

Jiří Mírovský
Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics
Malostranské nám. 25, 118 00 Prague 1, Czech Republic
mirovsky@ufal.mff.cuni.cz

Abstract

Netgraph query language is a query system for linguistically annotated treebanks that aims to be sufficiently powerful for linguistic needs and yet simple enough for not requiring any programming or mathematical skill from its users. We provide an introduction to the system along with a set of examples how to search for some frequent linguistic phenomena. We also offer a comparison to the querying power of TGrep – a traditional and well known treebank query system.

1 Introduction

Searching in a linguistically annotated treebank requires a sophisticated tool, the more so the more complex the annotation is. Many users require (quite understandably) a simple and easy-to-learn tool, and yet they expect it to be satisfactorily powerful. It is obvious that there is a trade-off between simplicity of a query language and its searching power.

Netgraph has been designed to perform the searching with maximum comfort and minimum requirements on its users. Although it has been developed primarily for the Prague Dependency Treebank 2.0 (Hajič et al. 2006), it can be used with other treebanks too, both dependency and constituent-structure types.

In this paper, we present Netgraph query language and show how it can be used to search for some frequent linguistic phenomena. Afterwards, we try to compare the searching power of Netgraph query system to the power of traditional TGrep (Pito 1994), in order to check if it is at least the same. Thus, we set a lower boundary to the power of Netgraph query language. Therefore we concentrate on showing that TGrep does not outperform Netgraph and only mention what TGrep's flaws are, also because we know that there exist TGrep2, TigerSearch and other more recent tools. But we consider the power of TGrep the first step on the way of Netgraph towards "a full-featured searching tool". We plan to offer a comparison with the more recent tools in some future paper.

In *section 1* (after this introduction) we very briefly describe the Prague Dependency Treebank 2.0, just to make the examples in the subsequent text more understandable. Anyone familiar with this treebank may safely skip this subsection. In the next subsection we also mention in a few words the history of Netgraph and its properties as a tool.

In *section 2* we offer an introduction to the query language of Netgraph along with the idea of meta-attributes and what they are good for, and present several linguistically motivated examples of queries in the Prague Dependency Treebank. We also list all available meta-attributes.

In *section 3* we compare Netgraph query language to TGrep by translating TGrep predicates to Netgraph.

Finally, in *section 4* we offer some concluding remarks.

1.1 Prague Dependency Treebank 2.0

The Prague Dependency Treebank 2.0 (PDT 2.0, see Hajič et al. 2006, Hajič 2004) is a manually annotated corpus of Czech. It is a sequel to the

Prague Dependency Treebank 1.0 (PDT 1.0, see Hajič et al. 2001a, Hajič et al. 2001b).

The texts in PDT 2.0 are annotated on three layers - the morphological layer, the analytical layer and the tectogrammatical layer. The corpus size is almost 2 million tokens (115 thousand sentences), although "only" 0.8 million tokens (49 thousand sentences) are annotated on all three layers. By 'tokens' we mean word forms, including numbers and punctuation marks.

On the morphological layer (Hana et al. 2005), each token of every sentence is annotated with a lemma (attribute `m/lemma`), keeping the base form of the token, and a tag (attribute `m/tag`), keeping its morphological information. Sentence boundaries are annotated here, too.

The analytical layer roughly corresponds to the surface syntax of the sentence; the annotation is a single-rooted dependency tree with labeled nodes (Hajič et al. 1997, Hajič 1998). The nodes on the analytical layer (except for technical roots of the trees) also correspond 1:1 to the tokens of the sentences. The order of the nodes from left to right corresponds exactly to the surface order of tokens in the sentence. Non-projective constructions (that are quite frequent both in Czech (Hajičová et al. 2004) and in some other languages (see Havelka 2007)) are allowed. Analytical functions are kept at nodes (attribute `a/afun`), but in fact they are names of the dependency relations between a depending node (son) and its governing node (father).

The tectogrammatical layer captures the linguistic meaning of the sentence in its context. Again, the annotation is a dependency tree with labeled nodes. The correspondence of the nodes to the lower layers is more complex here. It is often not 1:1, it can be both 1:N and N:1. It was shown in detail in Mírovský (2006) how Netgraph deals with this issue.

Many nodes found on the analytical layer disappear on the tectogrammatical layer (such as functional words, prepositions, subordinating conjunctions, etc.). The information carried by these nodes is stored in attributes of the remaining (autosemantic) nodes and can be reconstructed. On the other hand, some nodes representing for example obligatory positions of verb frames, deleted on the surface, are regenerated on this layer.

The tectogrammatical layer goes beyond the surface structure of the sentence, replacing notions such as Subject and Object by notions like Actor, Patient, Addressee etc (see Hajičová 1998).

Attribute `functor` describes the dependency between a depending node and its governor and again is stored at the son-nodes. A tectogrammatical lemma (attribute `t_lemma`) is assigned to every node. Grammatemes are rendered as a set of 16 attributes grouped by the "prefix" `gram` (e.g. `gram/verbmod` for verbal modality).

The total of 39 attributes are assigned to every non-root node of the tectogrammatical tree, although (based on the node type) only a certain subset of the attributes is necessarily filled in.

Topic and focus (Hajičová et al. 1998) are marked (attribute `tfa`), together with so-called deep word order reflected by the order of nodes in the annotation (attribute `deepord`). It is in general different from the surface word order, and all the resulting trees are projective by the definition of deep word order.

To be complete (as much as possible in this short description), let us add that coreference relations between nodes of certain category types are captured (Kučová et al. 2003), distinguishing also the type of the relation (textual or grammatical). Each node has an identifier (attribute `id`) that is unique throughout the whole corpus. Attributes `coref_text.rf` and `coref_gram.rf` contain `id`s of coreferential nodes of the respective types.

1.2 Netgraph as a Tool

The development of Netgraph started in 1998 as a topic of Roman Ondruška's Master's Thesis (Ondruška 1998), and has been proceeding along with the ongoing annotations of the Prague Dependency Treebank 1.0 and later the Prague Dependency Treebank 2.0. Now it is a fully functional tool for complex searching in PDT 2.0.

Netgraph is a client-server application that allows multiple users to search the treebank on-line and simultaneously through the Internet. The server (written in C) searches the treebank, which is located at the same computer or local network. The client (written in Java2) serves as a very comfortable graphical user interface and can be located at any node in the Internet. It sends user queries to the server and receives results from it. Both the server and the client also can, of course, reside at the same computer. Authentication by the means of login names and passwords is provided. Users can have various access permissions.

A detailed description of the inner architecture of Netgraph and of the communication between the server and the client was given in Mírovský, Ondruška and Průša (2002).

2 Netgraph Query Language

In this section we give an introduction to the Netgraph query language. We show on a series of examples how some frequent linguistic phenomena can be searched for.

2.1 The Query Is a Tree

The query in Netgraph is a tree that forms a subtree in the result trees. The treebank is searched tree by tree and whenever the query is found as a subtree of a tree (we say the query and the tree match), the tree becomes part of the result. The result is displayed tree by tree on demand. The query can also consist of several trees joined either by `AND` or `OR` relation. In that case, all the query trees at the same time (or at least one of the query trees, respectively) are required to match the result tree.

The query has both a textual form and a graphical form. For lack of space, we will use its textual form in this paper. However, each textual query has its full graphical counterpart, which is always much more transparent.

The syntax of the language is very simple. In the textual form, square brackets enclose a node, attributes (pairs `name=value`) are separated by a comma, quotation marks enclose a regular expression in a value. Parentheses enclose a subtree of a node, brothers are separated by a comma. In multiple-tree queries, each tree is on a new line and the first line contains only a single `AND` or `OR`. Alternative values of an attribute, as well as alternative nodes, are separated by a vertical bar. It almost completes the description of the syntax, only one thing (references) will be added in the following subsection.

The simplest possible query (and probably of little interest on itself) is a simple node without any evaluation: `[]`. It matches all nodes of all trees in the treebank, each tree as many times as how many nodes there are in the tree. Nevertheless, we may add conditions on its attributes, optionally using regular expressions in values of the attributes. Thus we may search e.g. for all nodes that are Subjects and nouns but not in first case:

```
[afun=Sb, m/tag="N...[^1].*"].
```

We may notice here that regular expressions allow the first (very basic) type of negation in queries.

More interesting queries usually consist of several nodes, forming a tree structure. The following example query searches for trees containing a Predicate that directly governs a Subject and an Object:

```
[afun=Pred]([afun=Sb],[afun=Obj]).
```

Please note that there is no condition in the query on the order of the Subject and the Object, nor on their left-right position to their father. It does not prevent other nodes to be directly governed by the Predicate either.

2.2 Meta-Attributes

This simple query language, described briefly in only a few examples, is quite useful but not powerful enough. There is no possibility to set a real negation, no way of restricting the position of the query in the result tree or the size of the result tree, nor the order of nodes can be controlled. To allow these and other things, meta-attributes have been added to the query system.

Meta-attributes are not present in the corpus but they pretend to be ordinary attributes and the user uses them the same way like normal attributes. Their names start with an underscore. There are eleven meta-attributes, each adding some power to the query language, enhancing its semantics, while keeping the syntax of the language on the same simple level. We present several of the meta-attributes in this subsection, some others will be presented in the subsequent section, when they are needed. A list of all meta-attributes is presented in the next subsection.

Coordination is a frequent phenomenon in languages. In PDT (and in most other treebanks, too) it is represented by a coordinating node. To be able to skip (and effectively ignore) the coordination in the queries, we have introduced the meta-attribute `_optional` that marks an optional node. The node then may but does not have to be in the result. If we are interested, for example, in Predicates governing Objects, we can get both cases (with coordination and without it) in one query using this meta-attribute:

```
[afun=Pred]([afun=Coord,_option-
al=1]([afun=Obj])).
```

The Coordination becomes optional. If there is a node between the Predicate and its Object in the result tree, it has to be the Coordination. But the Object may also be a direct son of the Predicate, omitting the optional Coordination.

There is a group of meta-attributes of rather technical nature. They allow setting a position of the query in the result tree, restricting the size of the result tree or its part, and restricting number of direct sons of a node. Meta attribute `_depth` controls the distance of a node from the root (useful when searching for a phenomenon in subordinated clauses, for example), `_#descendants` controls number of nodes in the subtree of a node (useful e.g. when searching for „nice“ small examples of something), `_#sons` controls number of (direct) sons of a node.

Controlling number of direct sons (mainly in its negative sense) is important for studying valency of words (Hajičová and Panevová 1984). The following example searches on the tectogrammatical layer of PDT. We want a Predicate that governs directly an Actor and a Patient and nothing else (directly):

```
[functor=PRED,_#sons=2]([func-
tor=ACT],[functor=PAT]).
```

If we replaced PAT with ADDR, we might search for errors in the evaluation, since the theory forbids Actor and Addressee being the only parts of a valency frame.

So far, we could only restrict number of nodes. But we often want to restrict a presence of a certain type of node. We want to specify that there is not a node of a certain quality. For example, we might want to search (again on the tectogrammatical layer) for an Effect without an Origo in a valency frame. The meta-attribute that allows this real type of negation is called `_#occurrences`. It controls the exact number of occurrences of a certain type of node, in our example of Origos:

```
[functor=PRED]([functor=EFF],[fu
nctor=ORIG, _#occurrences=0]).
```

It says that the Predicate has at least one son – an Effect, and that the Predicate does not have an Origo son.

There is still one important thing that we cannot achieve with the meta-attributes presented so far. We cannot set any relation (other than dependency) between nodes in the result trees (such as order, agreement in case, coreference). All this can be done using the meta-attribute `_name` and a system of references. The meta-attribute `_name` simply names a node for a later reference from other nodes.

Curly brackets enclose a reference to a value of an attribute of the other node (with a given name) in the result tree. This, along with the dot-referencing inside the reference and some arithmetic possibilities, completes our description of the syntax of the query language from subsection 2.1.

In the following example (back on the analytical layer and knowing that attribute `ord` keeps the order of the node (~ token) in the tree (~ sentence)), we search for a Subject that is on the right side from an Object:

```
[afun=Pred]([afun=Sb,ord>{N1.ord
}],[afun=Obj,_name=N1]).
```

We have named the Object node `N1` and specified that ord of the Subject node should be bigger than `ord` of the `N1` node. If we used `ord>{N1.ord}+5`, we would require them to be at least five words apart.

2.3 List of All Meta-Attributes

To complete our description of Netgraph query language, we present all available meta-attributes in one list, along with a short description:

`_transitive`

This meta-attribute defines a transitive edge. It has two possible values: `true` means that a node may appear anywhere in the subtree of its query-father, `exclusive` means, in addition, that the transitive edge cannot share nodes in the result tree with other exclusively transitive edges.

`_optional`

It defines an optional node. It may but does not have to appear in the result. However, if there is a node in the result at this particular place (father in grandfather-father-son hierarchy), it must be the one defined in the query. Depending on the value of this meta-attribute, one or more nodes may be skipped. A special value `true` skips an unlimited chain of the specified nodes.

`_#sons`

It defines an exact number of sons of a query-node in the result tree.

`_#hsons`

It defines an exact number of hidden sons of a query-node in the result tree. Hidden nodes are especially marked nodes in the tree that provide connection to the information on the lower layers of annotation. They are useful when the relation between nodes at different layers is not 1:1. A detailed description of the system of hidden nodes was given in Mírovský (2006).

`_#descendants`

This meta-attribute defines an exact number of all descendants of a node (number of nodes in its subtree), excluding the node itself.

`_#lbrothers`

This meta-attribute defines an exact number of left brothers of a node.

`_#rbrothers`

Similarly, it defines an exact number of right brothers of a node.

`_depth`

It defines a distance between a node and a root in the result tree.

`_#occurrences`

This meta-attribute specifies an exact number of occurrences of a particular node at a particular place in the result tree. It controls how many nodes of the kind can occur in the result tree as sons of the father of the node (including the node itself). It can be combined with meta-attribute `_transitive` for transitive meaning of the above definition.

`_name`

It names a node for references to values of its attributes in the result tree.

`_sentence`

The value of this meta attribute is simply the sentence the result tree belongs to in its linear form. It can be used for linear searching in the sentence (using regular expressions).

3 Comparison to TGrep

In this section, we compare the query language of Netgraph to the query language of TGrep, in order to show that the power of Netgraph query language is at least the same as the power of TGrep. We also show at the end that Netgraph has a greater power.

In subsection 3.1 we compare the ability of expressing an evaluation of a node. In the next two subsections (3.2 and 3.3) we translate TGrep positive and negative predicates to Netgraph expressions. In subsection 3.4 we give an example of Netgraph expressions that cannot be searched for in TGrep.

3.1 Node Evaluation

TGrep is a one-attribute searcher. Each node is supposedly labeled only either by a non-terminal symbol or a token. Netgraph, on the other hand, can deal with multiple attributes and set conditions on them separately and even form groups of them that are labeled differently (so called "alternative nodes"). Leaving this aside, we can say that Netgraph has (at least) the same expressing power in the sense of node values as TGrep does, as both tools allow using regular expressions and set alternative values. Thus, we can almost simply repeat the example of a search pattern from TGrep manual:

in TGrep:

```
/^[Cc]hild.*$/|kid|youngster
```

in Netgraph:

```
"[Cc]hild.*"|kid|youngster
```

Netgraph regular expressions are automatically anchored and are enclosed in quotation marks. The complete query in Netgraph in the text form would then be (it also has to be "escaped" in the text form, though not in the graphical form):

```
[token="\[Cc\]hild.*"|kid|young-
ster]
```

The wildcard represented by two underscores in TGrep is reproducible in Netgraph by not specifying any attribute at the node: `[]`.

3.2 Tree Structure

The close similarity between Netgraph and TGrep in expressing node evaluations disappears completely when it comes to defining relations between nodes. Here, these two tools have quite a different approach. The main difference is that TGrep uses predicates to express dependency between nodes, while Netgraph expresses dependency directly in the syntax of the query. In this subsection, we try to match TGrep positive predicates with similar constructions in Netgraph. We take predicates (relationships between nodes) from

TGrep manual one by one and translate them to equivalent Netgraph expressions.

The first line of each example (starting with `T`) always shows the expression in TGrep, while the second line (starting with `N` and occasionally followed by other lines) shows the equivalent expression in Netgraph.

A immediately dominates B:

```
T: A < B
N: [A]([B])
```

B is the X-th son of A:

```
T: A <X B
N: [A]([B,_#lbrothers=X-1])
```

We use meta-attribute `_#lbrothers` here, which specifies how many left brothers a node has. X-th to last son is similar, we only use meta-attribute `_#rbrothers` (number of right brothers).

A dominates B (A is dominated by B similarly):

```
T: A << B
N: [A]([B,_transitive=true])
```

Meta-attribute _transitive defines the father edge as transitive.

B is the leftmost (rightmost) descendant of A:

```
T: A <<, B
N:
[A]([B,_transitive=true,_name=N1],
[_transitive=true,ord<{N1.ord},
_#occurrences=0]).
```

B is a transitive descendant of A and there is no transitive descendant of A that has smaller ord than B. Rightmost descendant is similar (`ord<{N1.ord})`.

A immediately precedes B:

```
T: A . B
N: AND
[A,_name=N1]
[B,ord={N1.ord+1}]
```

Since we generally do not know what dependency relation between the two nodes is, we must define them as two separate trees in a multiple-tree query (another possibility is to use a wildcard and two transitive sons). A precedes B is similar, we only use a different expression in the second tree:

```
N: [B,ord>{N1.ord}]
```

A and B are brothers:

```
T: A $ B
N: []([A],[B])
```

We use the wild card here since we generally do not know anything about the father (we only know that there must be one).

A and B are brothers and A immediately precedes B:

```
T: A $. B
N: []([A,_name=N1][B,
_#brothers={N1._#brothers}+1])
```

We have to use meta-attribute `_#brothers` here instead of attribute `ord`, because there may be other nodes (not brothers of A and B) in between them in left-right order of nodes. On the other hand, if we wanted to take the other nodes into account, we might use attribute `ord`.

Of course, things get more complex when we start combining these expressions. We believe that in Netgraph the complex expressions remain well readable. Sometimes we may be lucky and have a convenient meta-attribute at our disposal, just like in the following example, taken again from TGrep manual, which specifies all nodes A that dominate either two or three sons:

```
T: A <2 __ !<4 __
N: [A,_#sons=2|3]
```

3.3 Negation

Netgraph's way of specifying relations between nodes, especially their dependency, is primarily positive and it has some difficulty expressing negative relations. For this reason, it is sometimes not easy or even possible to match directly and exactly TGrep negative expressions without “saying” something positive about the nodes, too.

A does not immediately dominate B:

```
T: A !< B
N: [A]([B,_#occurrences=0]).
```

B is not the X-th son of A:

```
T: A !<X B
N: A([B,_#lbrothers!=X-1])
```

But note that it also means that B is a son of A. Using meta-attribute _#occurrences again, we may have another try on this example with a different meaning:

```
N: [A]([B,_#lbrothers=X-1,_#occur
rences=0])
```

Here, B still may be a son of A, but not necessarily, and in any case not the X-th one.

A does not dominate B:

```
T: A !<< B
N: [A]([B,_transitive=true,_#occur
rences=0])
```

B is not the leftmost descendant of A:

```
T: A !<<,B
```

This again must be considered in two separate cases: positive and negative. If we only want to say that the leftmost descendant of A has another property than B, the query in Netgraph would be:

```
N: [A]([!B,_transitive=true,
_name=N1],[_transitive=true,
ord<{N1.ord},_#occurrences=0]).
```

On the other hand, if we want to say that B is a descendant of A that is not the leftmost one, the query would be:

```
N: [A]([B,_transitive=true,
_name=N1],[ord<{N1.ord},_#occur-
rences>=1,_transitive=true])
```

A does not immediately precede B:

```
T: A !. B
N: AND
[A,_name=N1]
[!B,ord={N1.ord+1}]
```

Which is very similar to the positive case from the previous subsection. Note that it also means that there is a directly subsequent node !B in the result tree (a node that does not have B-property).

A does not precede B:

```
T: A !.. B
```

Just like before, two possible interpretations of this expression lead to two different realizations in Netgraph. The positive meaning is quite simple – A does not precede B is equal to B precedes A (since nodes cannot have the same left-right order). The negative meaning (there is A that is not followed by B) would be translated:

```
AND
[A,_name=N1]
[B,ord>{N1.ord},_#occurrences=0]
```

A is not a brother of B:

```
T: A !$ B
N: []([A],[B,_#occurrences=0])
```

If we also wanted to use B positively in the query, we might add another tree of a multiple-tree query.

It is not true that A $. B (similarly A !$.. B)

```
T: A !$. B
```

Many possible interpretations of this expression lead to many different realizations of the equivalent Netgraph query. We will not show all of them (they are all similar to the previous queries) but only choose the most direct one, B is a brother of A but does not immediately follow A:

```
N: []([A,_name=N1],
[B,_#lbrothers!={N1._#lbrothers}+1
])
```

3.4 The Other Way

Since TGrep always searches for one pattern only, it cannot reproduce multiple-tree queries from Netgraph, combined with expression OR. Meta-attribute `_optional` also represents a type of OR-expression on the tree structure and even the simple example given in subsection 2.2 cannot be reproduced in TGrep:

```
[afun=Pred]([afun=Coord,_option-
al=1]([afun=Obj])).
```

4 Conclusion

We have presented Netgraph query language on a set of linguistically motivated examples. We have compared Netgraph query power to the power of TGrep query language in order to show that it is not lesser, by translating all TGrep predicates to expressions in Netgraph. We have also shown that some Netgraph expressions cannot be translated to TGrep.

Many constructions in Netgraph seem more complicated than respective expressions in TGrep. The reason is that we matched TGrep predicates. It is clear that any other system that uses a different set of predicates cannot be as straightforward as TGrep in mimicking these predicates. It is sufficient for our purpose that the translation is possible.

We can conclude that Netgraph query language is at least as strong as TGrep query language. The impossibility of translating OR-expressions from Netgraph to TGrep shows that Netgraph query language is stronger than TGrep query language.

Acknowledgment

The research reported in this paper was supported by the Grant Agency of the Academy of Sciences of the Czech Republic, project IS-REST (No. 1ET101120413).

References

Hajič J. et al. 2006. Prague Dependency Treebank 2.0. *CD-ROM LDC2006T01, LDC, Philadelphia, 2006.*

Pito R. 1994. TGrep Manual Page. *Available from http://www.ldc.upenn.edu/ldc/online/treebank/*

Hajič J. 2004. Complex Corpus Annotation: The Prague Dependency Treebank. *Jazykovedný ústav Ľ. Štúra, SAV, Bratislava, 2004.*

Hajič J., Vidová-Hladká B., Panevová J., Hajičová E., Sgall P., Pajas P. 2001a. Prague Dependency Treebank 1.0 (Final Production Label). *CD-ROM LDC2001T10, LDC, Philadelphia, 2001.*

Hajič J., Pajas P. and Vidová-Hladká B. 2001b. The Prague Dependency Treebank: Annotation Structure and Support. *In IRCS Workshop on Linguistic databases, 2001, pp. 105-114.*

Hana J., Zeman D., Hajič J., Hanová H., Hladká B., Jeřábek E. 2005. Manual for Morphological Annotation, Revision for PDT 2.0. *ÚFAL Technical Report TR-2005-27, Charles University in Prague, 2005.*

Hajič J. et al. 1997. A Manual for Analytic Layer Tagging of the Prague Dependency Treebank. *ÚFAL Technical Report TR-1997-03, Charles University in Prague, 1997.*

Hajič J. 1998. Building a Syntactically Annotated Corpus: The Prague Dependency Treebank. *In Issues of Valency and Meaning, Karolinum, Praha 1998, pp. 106-132.*

Hajičová E., Havelka J., Sgall P., Veselá K., Zeman D. 2004. Issues of Projectivity in the Prague Dependency Treebank. *MFF UK, Prague, 81, 2004.*

Havelka J. 2007. Beyond Projectivity: Multilingual Evaluation of Constraints and Measures on Non-Projective Structures. *In Proceedings of ACL 2007, Prague, pp. 608-615.*

Hajičová E, Panevová J. 1984. Valency (case) frames. *In P. Sgall (ed.): Contributions to Functional Syntax, Semantics and Language Comprehension, Prague, Academia, 1984, pp. 147-188.*

Mírovský J. 2006. Netgraph: a Tool for Searching in Prague Dependency Treebank 2.0. *In Proceedings of TLT 2006, Prague, pp. 211-222.*

Hajičová E. 1998. Prague Dependency Treebank: From analytic to tectogrammatical annotations. *In: Proceedings of 2nd TST, Brno, Springer-Verlag Berlin Heidelberg New York, 1998, pp. 45-50.*

Hajičová E., Partee B., Sgall P. 1998. Topic-Focus Articulation, Tripartite Structures and Semantic Content. *Dordrecht, Amsterdam, Kluwer Academic Publishers, 1998.*

Kučová L., Kolářová-Řezníčková V., Žabokrtský Z., Pajas P., Čulo O. 2003. Anotování koreference v Pražském závislostním korpusu. *ÚFAL Technical Report TR-2003-19, Charles University in Prague, 2003.*

Ondruška R. 1998. Tools for Searching in Syntactically Annotated Corpora. *Master Thesis, Charles University in Prague, 1998.*

Mírovský J., Ondruška R., Průša D. 2002. Searching through Prague Dependency Treebank - Conception and Architecture. *In Proceedings of The First Workshop on Treebanks and Linguistic Theories, Sozopol, 2002, pp. 114--122.*