Dependency-Based Semantic Interpretation for Answer Extraction

Diego Mollá
Centre for Language Technology
Department of Computing
Macquarie University
Sydney, NSW 2109, Australia
diego@ics.mq.edu.au

Ben Hutchinson
Division of Informatics
University of Edinburgh
Edinburgh EH8 9LW, United Kingdom
B.Hutchinson@sms.ed.ac.uk

Abstract

This paper focuses on the use of two algorithms for the integration of two broad-coverage dependency-based parsing systems into ExtrAns, an answer extraction system that operates over Unix manual pages. In the process, ExtrAns uses a semantic interpreter that converts the output of the parser into flat logical forms. The semantic interpreter for the first parsing system, Link Grammar, uses a top-down algorithm, whereas the semantic interpreter for the second parsing system, Conexor FDG, uses a bottom-up algorithm. After showing the differences between the structures returned by both parsing systems, the paper compares the algorithms for semantic interpretation and comments on their adequacy for the production of flat logical forms.

1 Introduction

Dependency Theory is not new. Early concepts of syntactic dependency were developed in the 1930s (Tesnière, 1934; Peškovskij, 1934). The current surge of dependency-based parsers (Sutcliffe et al., 1996; Sleator and Temperley, 1993; Järvinen and Tapanainen, 1997) has spawned a revival of research in dependency theory and applications (Kahane and Polguère, 1998; Kahane, 2000).

The application chosen for the current study is Answer Extraction (AE). The fundamental goal of AE is to locate those exact phrases within text documents that answer a query worded in natural language. AE can be seen as a type of information retrieval (IR) that retrieves "answer passages" (O'Connor, 1975) and it has received much attention recently, as the increasingly active Question Answering track in TREC demonstrates (Voorhees, 2001b; Voorhees, 2001a).

Given the short size of the text to be returned, AE requires more thorough linguistic processing than document retrieval does. It is telling that purely corpus-based and bag-of-word approaches performed worse on 50-byte answers than on 250-byte answers in the Question Answering track of TREC9 (Voorhees, 2000). Several answer extraction systems, including the best system in TREC9, FALCON, used some sort of flat logical forms in the process (Harabagiu et al., 2000).

In this paper we compare two different approaches for the semantic interpretation of the output of two broad-coverage dependency-based parsers and grammars for ExtrAns, an answer extraction system that uses flat logical forms. We decided to build the semantic interpreters on top of the output of the chosen parsers so that the parsers do not need to be modified. Such a two-stage approach enables the use of commercial parsers without having to access to their source codes. The first parser chosen is Link Grammar (Sleator and Temperley, 1993), a publicly available parser and grammar developed at Carnegie Mellon University. We use a top-down approach where the dependencies are traversed

starting from the head and following the dependencies until all the words are covered. The other parser chosen is Conexor FDG (Tapanainen and Järvinen, 1997), a commercial parser and grammar developed at the University of Helsinki. In this case a bottom-up approach is devised where the words are traversed and the relations with their heads are explored.

Section 2 introduces ExtrAns and the flat logical forms used by ExtrAns. Section 3 focuses on the top-down and the bottom-up semantic interpreters implemented in ExtrAns, before reaching the final conclusion in Section 4.

2 Flat Logical Forms for Answer Extraction

ExtrAns is an answer extraction system that operates over UNIX manual pages (Mollá et al., 2000b). The general architecture of ExtrAns is shown in Figure 1. In an off-line stage, the manual sentences are processed and the logical forms produced. These logical forms are converted into Prolog-like Horn clauses and stored in a database. In an on-line stage, the user query is processed similarly to produce its logical form and subsequent set of Horn clauses. ExtrAns finds the answers to the questions by converting the Horn clauses of the questions into Prolog queries and then running Prolog's default resolution mechanism to find those Horn clauses in the database that can prove the question. This default search procedure is called the *synonym mode* since the logical forms convert all the synonyms into a synonym representative, according to a small WordNet-style thesaurus (Fellbaum, 1998). ExtrAns also has a *hyponym mode* that expands the logical form of the query with the first-level hyponyms, and an *approximate mode*, which returns the sentences with the highest overlap in the logical forms.

In order to evaluate the impact of the parsers and semantic interpreters in ExtrAns, we used a collection of 500 UNIX manual pages and a test set of 26 queries with their answers, and ran ExtrAns in the approximate mode to maximise the likelihood of ExtrAns returning an answer to a query. The results (Tables 1 and 2) indicate that the choice of parser did not cause a major change in the performance of the system. A more detailed evaluation is described in (Mollá and Hutchinson, 2002).

Parser	*Precision*[1]	*Recall*	*F-score*
Conexor FDG	28.3%	21.9%	0.177
Link Grammar	31.8%	15.8%	0.150

Table 1: Average precision, recall and F-score (with equal weighting for precision and recall) per query

Parser	*No results returned*	*No relevant results returned*
Conexor FDG	0	8
Link Grammar	1	11

Table 2: Numbers of times no relevant answers were found from a total of 26 queries

2.1 The Logical Forms

An important feature of the logical forms used by ExtrAns is that they do not have nested expressions. The ability of these flat logical forms to underspecify makes them good candidates for NLP applications, specially when the applications benefit from the semantic comparison of sentences (Copestake et al., 1997; Mollá, 2001). In the case of ExtrAns, the logical forms only encode the dependencies between verbs and their arguments, plus modifier and adjunct relations and they are called *minimal logical forms* (MLFs). Ignored information includes complex quantification, tense and aspect, temporal relations, plurality, and modality. We have argued elsewhere that too detailed a logical form may interfere with the answer extraction mechanism and, if necessary in a subsequent stage, additional information can be added incrementally (Mollá et al., 2000b).

The MLFs of ExtrAns use *reification* to achieve flat expressions, very much in the line of (Davidson, 1967; Hobbs, 1985; Copestake et al., 1997). The MLFs do not reify all predicates, as opposed, say, to (Hobbs, 1985; Copestake et al., 1997; Mollá, 2001). In the current implementation only reification of objects, eventualities (events or states), and properties is carried out. The MLFs

[1] Average precision over queries for which precision is defined, i.e. when the number of returns is non-zero. In cases with non-defined precision the F-score was set to zero, and included in the results.

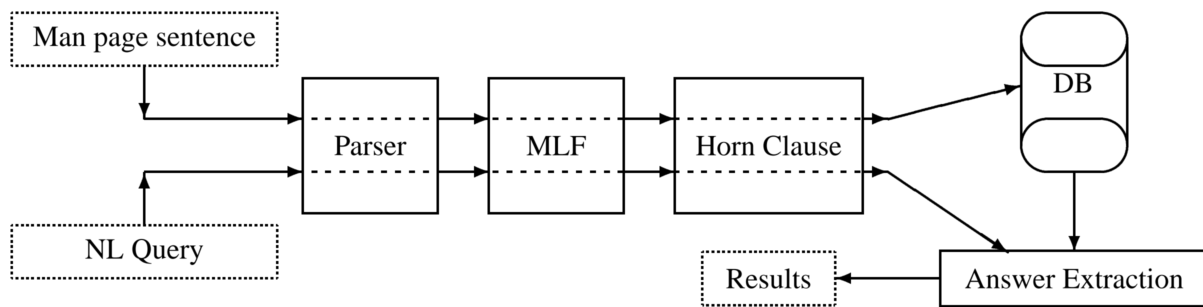


Figure 1: ExtrAns architecture

are expressed as conjunctions of predicates with all the variables existentially bound with wide scope. For example, the MLF of the sentence *cp will quickly copy files* is:

```
holds(e4),
object(cp,o1,[x1]),
object(s_command,o2,[x1]),
evt(s_copy,e4,[x1,x6]),
object(s_file,o3,[x6]),
prop(quickly,p3,[e4]).
```

In other words, there is an entity `x1` which represents an object of type `command`[2] (the `s_` prefix indicates the synonym class of `command`); there is an entity `x6` (a file); there is an entity `e4`, which represents a copying event where the first argument is `x1` and the second argument is `x6`; there is an entity `p3` which states that `e4` is done quickly, and the event `e4`, that is, the copying, holds. The reification of the event, `e4`, has been used to express that the event is done quickly. The entities `o1`, `o2`, `o3`, and `p3` are not used in this MLF, but other more complex sentences may need to refer to the reification of objects (required for non-intersective adjectives *the alleged murder*) or properties (required for adjective-modifying adverbs *very bright*).

Answer extraction is performed by finding those sentences whose logical form predicates form a superset of the logical form predicates of the question. More specifically, a Prolog call is produced with all variables in the logical form converted into Prolog variables. The default Prolog resolution mechanism suffices to find the answers. For example, the Prolog call generated for the question *which command copies files?* is (simplified):

```
?-
object(s_command,O1,[X1]),
evt(s_copy,E4,[X1,X6]),
object(s_file,O3,[X6]).
```

Given that the MLFs are simplified logical forms converted into flat structures, ExtrAns may find sentences that, logically speaking, are not exact answers but are still related to the user's question. Thus, given the question above, ExtrAns may also find sentences such as *cp copies files*, *cp does not copy files*, or *if the user types y, then cp copies the files* (Mollá et al., 2000b).

3 Logical Form Generation from Link Grammar and Conexor FDG

The generation of the MLFs is done in two stages. First a parser and grammar produces a syntactic analysis of each sentence in the form of a dependency structure. A logical form generator then produces logical forms from this dependency structure. Whereas former versions of ExtrAns used Link Grammar for the syntactic analysis, the current version is able to use either Link Grammar or the Conexor FDG parser.

3.1 The Outputs of Link Grammar and Conexor FDG

Example dependency structures returned by Link Grammar and Conexor FDG are shown in Figures 2 and 3 respectively. As the differences between Link Grammar and Conexor FDG are discussed elsewhere (Mollá and Hutchinson, 2002), here we just summarise the main differences.

[2]ExtrAns uses additional domain knowledge to infer that *cp* is a command.

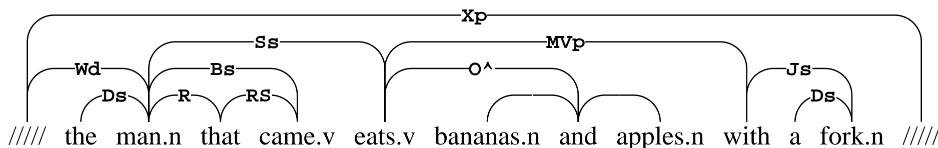


Figure 2: Output of Link Grammar

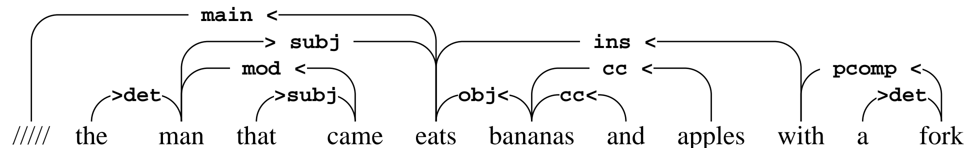


Figure 3: Dependency returned by Conexor FDG

Direction of dependency: In contrast with Conexor FDG, Link Grammar's 'links' do not indicate the direction of dependence.

Clausal heads: While Conexor FDG follows the orthodox convention of choosing the main verb as the head of the clause, Link Grammar generally chooses the subject.

Graphs: Link Grammar's linkages are not always tree structures (e.g. linkages of sentences with relative clauses).

Conjunctions: Link Grammar analyses a coordinating conjunction as the head of a coordinated phrase.[3] However in Conexor FDG's analyses the head will be either the first or the last conjunct, depending on the position of the coordinated phrase within the sentence.

Dependency types: Link Grammar uses a set of about 90 link types, with many subtypes, while Conexor FDG uses a set of 32 dependency relations. Different distinctions are made in the two sets of dependency types.

Non-dependency information: Conexor FDG returns the base form and morphological information for each word, along with a "functional" tag and "shallow syntactic" tag (this information has been removed from Figure 3). Link Grammar returns suffixes on some words indicating which word class they belong to.

[3]The version used in ExtrAns differs from the Link Grammar default behaviour that returns a separate parse for each conjunct.

3.2 Logical Form Generation from Link Grammar

The Link Grammar parser returns several dependency structures when the sentences are ambiguous, and the dependency structures include inflected forms of the words. As a consequence, ExtrAns uses a disambiguation process and a third-party lemmatiser. An additional module resolves pronominal anaphora. In the evaluation described in Section 2 the anaphora resolution module was disabled and the first parse was chosen in order to enable a fair comparison between Link Grammar and Conexor FDG.

The construction of the flat logical forms out of Link Grammar's dependency structures follows two steps (Mollá et al., 2000a). In the first step, the direction of the dependence is added to the links returned by Link Grammar. In most cases this is done by looking up the link type in a table. There are some specific cases (for example the link type `B`, a rather complex link type used in a number of situations involving relative clauses and questions) where the direction of the dependency depends on the context and a more elaborate algorithm is necessary for these cases.

In the second step, the logical form is constructed by a top-down procedure that starts from the head of the main sentence (The dependency labels below correspond to those of the example in Figure 2):

1. Starting from the sentence root, find the head of the main sentence. The sentence root is indicated by '/////' on the left in Figure 2.

The head of the sentence `eats` is found after following the links `Wd` (the link that connects the sentence with the root or "wall") and `Ss` (connecting a subject with the verb).

2. Follow the subject dependency (`Ss`) to find the head of the subject and build its logical form `object(man,o1,[x1])`. The subject dependency is attached to the main verb or the leftmost auxiliary verb if there is any.

3. Follow the dependencies of the subject modifiers (`R`) and build their corresponding logical form `evt(come,e1,[x1])`.

4. Build the logical forms of the other verb arguments (`O^`). In our example the object is a conjunction that introduces an object lattice `object(banana,o2,[x2])`, `object(apple,o3,[x3])`, $x2 \sqsubseteq x4$, $x3 \sqsubseteq x4$.

5. Create an entity for the main eventuality, `e2`.

6. Build the logical forms of the sentence modifiers and adjuncts (`MVp`). In our example they are `with(e2,x5)`, `object(fork,o4,[x5])`.

7. Add the logical form of the main event (`e2`) and the `holds` predicate `holds(e2)`, `evt(eat,e2,[x1,x4])`.

Most of the above steps can become very complex, sometimes involving recursive applications of the algorithm. For example, the logical form of the relative clause *that came* is generated by applying the general algorithm recursively. On top of this, specific particularities of the dependency structures returned by Link Grammar (summarised in Section 3.1) add complexity to the process (Mollá et al., 2000a).

This algorithm would ignore complete sentence constituents whenever an unexpected link appeared. As a result, a special recovery treatment has been implemented to find the unprocessed words, locate their heads, and recursively apply the top-down algorithm. The logical forms of the resulting islands are added to the logical form of the top sentence. The result is a logical form that encodes the gist of the sentence, leaving some attachment decisions unspecified.

3.3 Logical Form Generation from Conexor FDG

In contrast with the top-down approach used with Link Grammar, a bottom-up approach is used to explore the output of Conexor FDG. Logical form predicates are constructed incrementally, combining information from one or more words in the sentence. We associate each predicate with the word in the input sentence which introduces the predicate. This enables us to map syntactic head–dependent relations on to logical predicate–argument relations. That is, if the dependency structure shows a certain logical argument then we can easily deduce which predicates need to be modified.

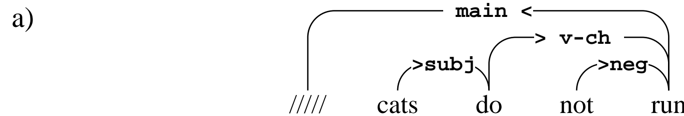
The algorithm for the semantic interpretation has three stages. The first stage, called **Introspection**, associates a (possibly underspecified) predicate with each word. The second stage, **Extrospection**, uses dependency relations to fully specify each predicate by filling in its arguments. Lastly, a **Re-interpretation** stage is needed to reanalyse logical constructs such as coordination and negation, since their correct argument structure is not deducible solely from the dependency structure.

Introspection: The first stage consists of analysing each word and partially constructing the associated predicate. We call this stage Introspection, since each word is examined in isolation. The algorithm iterates through the words in the sentence, and for each word the word's base form and part of speech are used to add information to the predicate associated with that word. For example, Figure 4 shows the results of Introspection on the words *cats*, *not* and *run*. As exemplified by the case of *cats*, object predicates introduce their own logical arguments, whereas other predicates introduce empty arguments that need to be filled (indicated by '`?`').

Extrospection: The logical arguments of non-object predicates are added during the second stage, which again iterates through the words in the sentence. We call this stage Extrospection, since information is deduced by examining the relation between each word and its head (if it has one).

If a word is a *modifier* of its head then the

a)



b)

Processing stage	*Predicates associated with 'cats', 'not' and 'run', and the* `holds` *predicate, respectively*
Introspect(cats)	`object(cat,o2,[x2])`
Introspect(not)	`object(cat,o2,[x2]),log_op(not,l4,[?])`
Introspect(run)	`object(cat,o2,[x2]),log_op(not,l4,[?]) ,evt(run,e5,[?])`
Extrospect(cats)	`object(cat,o2,[x2]),log_op(not,l4,[?]) ,evt(run,e5,[x2])`
Extrospect(not)	`object(cat,o2,[x2]),log_op(not,l4,[e5]),evt(run,e5,[x2])`
Extrospect(run)	`object(cat,o2,[x2]),log_op(not,l4,[e5]),evt(run,e5,[x2]),holds(e5)`
Re-interpretation	`object(cat,o2,[x2]),log_op(not,l4,[e5]),evt(run,e5,[x2]),holds(l4)`

Figure 4: (a) Conexor FDG's analysis of *cats do not run*, and (b) the incremental construction of predicates.

empty argument is instantiated with a value taken from the predicate introduced by the head. Depending on the nature of the relation between the head and the modifier the new value is the reification (1) or the logical argument (2) of the head's predicate:

(1) *don't run*
`log_op(not,l1,[` `e1` `]),`
`evt(run,e1,[x1])`

(2) *red car*
`prop(red,p1,[` `x1` `]), object(car,o1,[x1])`

If a word is a syntactic *argument* of its head it is the head's predicate which instantiates a logical argument:

(3) *cats run*
`object(cat,o1,[x1]),`
`evt(run,e1,[` `x1` `])`

In only a few cases, notably PP agents in passives, a chain of dependencies is inspected in order to assign the proper logical arguments. Figure 4 gives an example of incremental MLF construction through Extrospection. An example of how arguments and modifiers contribute to the MLFs can be seen in the Extrospection of *cats* and *not*, respectively. Arguments in multivalent predicates are managed so that they are ordered in a canonical form, ensuring, for example, that the subject of a passive verb is second in the argument list.

Extrospection is bottom-up in nature, in that for each word the algorithm looks 'up' at its unique head, rather than looking 'down' at its dependents, of which there may be several. This bottom-up nature makes the algorithm robust when faced with disconnected dependency structures, which result when a complete parse cannot be found. In such cases, the Extrospection stage may assign a dummy argument if no logical argument has yet been assigned. For example, if *ate cakes* is parsed as a disconnected chunk, Extrospection of *cake* yields `evt(eat,e1,[X])`, with dummy argument `X`, and when *cakes* too has been processed the resulting predicates are `evt(eat,e1,[X,x1])`,[4] `object(cake,o1,[x1])`. If a verb has no arguments in the dependency structure, the MLF generator arbitrarily assumes the verbs to be intransitive, since it lacks access to a lexicon to check transitivity. The same process of introducing dummy arguments ensures, for example, that `eat` receives a dummy subject in *Eating cakes is fun*.

As with the case of disconnected dependency structures, the algorithm gracefully deals with complex dependency structures that are not fully covered by the semantic interpreter by producing predicates that may not be connected. Without having to introduce a recovery stage like in the top-down algorithm, the bottom-up algorithm can produce the same flat logical form as the top-down algorithm.

Extrospection only examines *local* relationships between words, that is, the logical argu-

[4]The object of a transitive verb introduces a new argument, which in this case takes the value `x1`.

ments of a predicate are deduced just from the associated word, its syntactic head, and the arguments of that head's predicate. While this locality constraint allows a fast and robust generation of MLFs, it also means the present algorithm cannot produce the correct argument structure for long distance dependencies, since these dependencies are not directly expressed in the structure returned by the parser. For example, the object of `eat` cannot be correctly assigned in *the cakes that I like to eat*, since `eat` can only inspect the arguments of `like` in its search for an argument. The effects that this constraint creates on the accuracy of ExtrAns still needs to be evaluated.

Re-interpretation: The last stage of logical form generation re-interprets logical operators expressing conjunctions, implication and negation. In each of these cases there is a mismatch between semantic dependence and syntactic dependence. For example, in Figure 4, although *not* is dependent on *run*, semantically the negation governs the verb's predicate. Re-interpretation involves replacing the argument of the logical operator with the reified logical operator in all predicates, for example the `holds` predicate in Figure 4.

In order to generate the specific logical forms required by ExtrAns a few exceptions to the general algorithm described above are necessary. Firstly, since MLFs ignore tense and aspect, subjects are treated as arguments of the head of the verb chain, ignoring any auxiliary verbs. This is achieved by associating the same predicate with all verbs in the verb chain. For example, from *cats do not run*, we get the desired eventuality predicate `evt(run,e5,[x1])` rather than two predicates such as `evt(do,e2,[x1])`, `evt(run,e5,[x1])`. In addition, predicates expressing copula constructions are not desired either. That is, a sentence such as *cats are sleepy* should give `holds(p1), object(cat,o1,[x1]), prop(sleepy,p1,[x1])`. Associating copulas with the same predicate as their complement has the desired effect. Lastly, if a word is the head of a sentence (as detected by the dependency type 'main', see Figure 3), the special `holds` predicate is created.

4 Summary and Conclusion

Even though Link Grammar and Conexor FDG are dependency-based parsing systems, the dependency structures returned differ substantially and therefore two independent logical form generators are needed. This gave us the possibility to experiment on the methodologies for traversing the dependency structures and constructing the flat logical forms used by our answer extraction system ExtrAns. Whereas a top-down approach was used for Link Grammar, a bottom-up approach was used for Conexor FDG. We found the bottom-up approach easier to implement than the top-down and more robust in handling complex dependency structures. In contrast to the top-down approach, the bottom-up approach would always process all words and produce partial logical forms. The use of dependency structures plus the flat nature of the MLFs makes possible this robust feature of the bottom-up approach.

The choice of a top-down algorithm for Link Grammar and a bottom-down algorithm for Conexor FDG was arbitrary, and it is predicted that a bottom-up approach would be the preferred choice for Link Grammar as well. We will pursue to test this in the near future.

References

Ann Copestake, Dan Flickinger, and Ivan A. Sag. 1997. Minimal recursion semantics: an introduction. Technical report, CSLI, Stanford University, Stanford, CA.

Donald Davidson. 1967. The logical form of action sentences. In Nicholas Rescher, editor, *The Logic of Decision and Action*, pages 81–120. Univ. of Pittsburgh Press.

Christiane Fellbaum. 1998. Wordnet: Introduction. In Christiane Fellbaum, editor, *WordNet: an electronic lexical database*, Language, Speech, and Communication, pages 1–19. MIT Press, Cambrige, MA.

Sanda Harabagiu, Dan Moldovan, Marius Paşca, Rada Mihalcea, Mihai Surdeanu, Răzvan Bunescu, Roxana Gîrju, Vasile Rus, and Paul Morărescu. 2000. Falcon: Boosting knowledge for answer engines. In Voorhees and Harman (Voorhees and Harman, 2000).

Jerry R. Hobbs. 1985. Ontological promiscuity. In *Proc. ACL'85*, pages 61–69. University of Chicago, Association for Computational Linguistics.

Timo Järvinen and Pasi Tapanainen. 1997. A dependency parser for english. Technical Report TR-1, Department of Linguistics, University of Helsinki, Helsinki.

Sylvain Kahane and Alain Polguère, editors. 1998. *Processing of Dependency-Based Grammars: Proceedings of the Workshop COLING-ACL98*. Montreal.

Sylvain Kahane, editor. 2000. *Traitment Automatique des Langues: Special Issue on Dependency Grammars*. Hermes, Paris.

Diego Mollá and Ben Hutchinson. 2002. In vitro and in vivo evaluations of parsing systems within the context of answer extraction. In preparation.

Diego Mollá, Gerold Schneider, Rolf Schwitter, and Michael Hess. 2000a. Answer extraction using a dependency grammar in ExtrAns. *T.A.L.*, 41(1):127–156.

Diego Mollá, Rolf Schwitter, Michael Hess, and Rachel Fournier. 2000b. Extrans, an answer extraction system. *T.A.L.*, 41(2):495–522.

Diego Mollá. 2001. Ontologically promiscuous flat logical forms for NLP. In Harry Bunt, Ielka van der Sluis, and Elias Thijsse, editors, *Proceedings of IWCS-4*, pages 249–265. Tilburg University.

John O'Connor. 1975. Retrieval of answer sentences and answer-figures from papers by text searching. *Information Processing & Management*, 11(5/7):155–164.

Aleksandr Peškovskij. 1934. *Russkij Sintaksis v Nauňom Osveščnii (Russian Syntax: a Scientific Approach*. Učpedgiz, Moscow.

Daniel D. Sleator and Davy Temperley. 1993. Parsing English with a link grammar. In *Proc. Third International Workshop on Parsing Technologies*, pages 277–292.

Richard F. E. Sutcliffe, Heinz-Detlev Koch, and Annette McElligott, editors. 1996. *Industrial Parsing of Software Manuals*. Rodopi, Amsterdam.

Pasi Tapanainen and Timo Järvinen. 1997. A non-projective dependency parser. In *Procs. ANLP-97*. ACL.

Lucien Tesnière. 1934. Comment construire une syntaxe. *Bulletin de la Faculté des Lettres de Strasbourg*, 12(7):219–229.

Ellen M. Voorhees and Donna K. Harman, editors. 2000. *The Ninth Text REtrieval Conference (TREC-9)*, number 500-249 in NIST Special Publication. NIST.

Ellen M. Voorhees. 2000. Overview of the TREC-9 question answering track. In Voorhees and Harman (Voorhees and Harman, 2000).

Ellen M. Voorhees. 2001a. Overview of the TREC 2001 question answering track. In Ellen M. Voorhees and Donna K. Harman, editors, *Proc. TREC-10*, number 500-250 in NIST Special Publication. NIST.

Ellen M. Voorhees. 2001b. The TREC question answering track. *Natural Language Engineering*, 7(4):361–378.