To appear in *Proceedings of IJCAI-95*

A Dependency-based Method for Evaluating Broad-Coverage Parsers

Dekang Lin*
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
lindek@cs.umanitoba.ca

Abstract

With the emergence of broad-coverage parsers, quantitative evaluation of parsers becomes increasingly more important. We propose a dependency-based method for evaluating broad-coverage parsers. The method offers several advantages over previous methods that are based on phrase boundaries. The error count score we propose here is not only more intuitively meaningful than other scores, but also more relevant to semantic interpretation. We will also present an algorithm for transforming constituency trees into dependency trees so that the evaluation method is applicable to both dependency and constituency grammars. Finally, we discuss a set of operations for modifying dependency trees that can be used to eliminate inconsequential differences among different parse trees and allow us to selectively evaluate different aspects of a parser.

1. Introduction

With the emergence of broad-coverage parsers, quantitative evaluation of parsers becomes increasingly more important. It is generally accepted that such evaluation should be conducted by comparing the parser-generated parse trees (we call them **answers**) with manually constructed parse trees (we call them **keys**). However, how such comparison should be performed is still subject to debate. Several proposals have been put forward [Black *et al.*, 1991; 1992; Magerman, 1994], all of which are based on the comparison between phrase boundaries in answers and keys. We propose a dependency-based evaluation scheme in which the dependency relations, rather than phrase boundaries, are used in the comparison between answers and keys. We then show that the dependency-based scheme offers several advantages over previous proposals. Note that the use of dependency relations here does not mean that the scheme is only applicable to dependency grammars. It only means that constituency trees have to be transformed into dependency trees before answers and keys are compared. A transformation procedure will be presented in section 4.3.

*The author wishes to thank IJCAI reviewers for pointing out several errors in the draft and Mr. Wei Xiao for implementing the algorithms presented in the paper. The author is a member of the Institute for Robotics and Intelligent Systems (IRIS) and wishes to acknowledge the support of the Networks of Centres of Excellence Program of the Government of Canada, the Natural Sciences and Engineering Research Council (NSERC), and the participation of PRECARN Associates Inc. This research has also been partially supported by NSERC Research Grant OGP121338.

2. Previous Approaches

Given a node in a parse tree, the sequence of words dominated by the node form a phrase and the boundary of the phrase can be represented by an integer interval $[i, j]$, where i is the index of the first word in the phrase and j is the index of the last word in the phrase. For example, the parse tree in (1) contains three phrases boundaries: [0,2], [1,1], and [1,2].

(1) [They [[came] yesterday]]

Previous evaluation schemes can be classified as **phrase-level** or **sentence-level**. In phrase-level evaluation, the following goodness scores are computed:

Precision and recall: The phrase boundaries in the answer and the key are treated as two sets (A and K respectively). The recall is defined as the percentage of phrase boundaries in the key that are also found in the answer ($\frac{|A\cap K|}{|K|}$). The precision is defined as the percentage of phrase boundaries in the answer that are also found in the key ($\frac{|A\cap K|}{|A|}$).

Number of crossing-brackets: A phrase boundary $[i, j]$ in the answer and another phrase boundary in the key $[i', j']$ are a pair of crossing brackets if $i < i' \leq j < j'$. Parsers can be evaluated by the average number of crossing brackets per sentence.

For example, suppose, (1) is the key and (2) is the answer.

(2) [[They [came]] [yesterday]]

The phrase boundaries in (2) are: [0,2], [1,1], [0,1], and [2,2]. Thus, the scores of (2) are: precision=$\frac{2}{4}$=50%, recall=$\frac{2}{3}$=66.7% and there is one pair of crossing brackets: [0,1] in the key and [1,2] in the answer. These scores have to be considered together to be meaningful. For example, treating the sentence "they came yesterday" as a flat list of words [they came yesterday] would achieve 0-crossing-brackets and 100% precision. However, the recall is quite low ($\frac{1}{3}$=33.3%).

In sentence-level evaluation, a sentence is considered to be correctly parsed if certain criteria have been met.

In [Black *et al.*, 1992], the correctness criterion is that the number of crossing-brackets is 0. In [Magerman, 1994], a sentence is correctly parsed if both precision and recall are 100%. The scores assigned to parsers are their percentages of correctly parsed sentences in the test corpus.

A problem with the phrase-level scores is that they do not necessarily reflect the quality of parse trees, because a single error may be counted multiple times. For example, suppose (3a) is the key and (3b), (3c) are answers returned by two parsers.

(3)

a. [I [saw [[a man] [with [[a dog] and [a cat]]]] [in [the park]]]]

b. [I [saw [[a man] [with [[a dog] and [[a cat] [in [the park]]]]]]]]

c. [I [saw [a man] with [a dog] and [a cat] [in [the park]]]]

The only difference between (3a) and (3b) is that the prepositional phrase [in [the park]] is the sister of "saw" in (3a), but the sister of "a cat" in (3b). However, because of this single attachment error, there are 3 pairs of crossing brackets:

1. [a dog and a cat] vs. [a cat in the park]
2. [with a dog and a cat] vs. [a dog and a cat in the park]
3. [a man with a dog and a cat] vs. [with a dog and a cat in the park]

The recall of (3b) is $\frac{6}{10}$=60% and the precision is $\frac{7}{11}$=63.6%. In contrast, (3c) is a very shallow parse of the sentence. It has no crossing brackets, perfect precision (100%) and a better recall ($\frac{7}{10}$=70%) than (3b). Intuitively, the structure (3b) has a lot more in common with (3a) than (3c). Yet, (3b) scored much poorly than (3c) according to the precision, recall and the number of crossing-brackets.

In sentence-level evaluations, an error will not be counted more than once. However, the other extreme has to be adopted: no matter how many mistakes a parser makes in a parse tree, they are only counted as one error. Thus an answer with a single error is treated the same as an answer with many serious errors. Since how much a parse tree deviates from the correct one greatly influence the chance of the sentence being interpreted correctly, the evaluation scheme should take the degree of the deviation into account.

3. Desiderata

In this section, we motivate the dependency-based evaluation by pointing out several desirable properties that are missing from previous evaluation schemes.

3.1. Ignoring the inconsequential differences

An objective of a evaluation scheme is to identify the differences between answers and keys. However, certain types of differences are of no consequence to the interpretation of a sentence. An ideal evaluation scheme should make provisions to ignore such differences.

For example, consider sentence structures in (4):

(4)

a. [[Bellows [made [the request]]] [while [[the all-woman jury] [was [out [of [the courtroom]]]]]]]

b. [Bellows [[made [the request]] [while [[the [[all-woman] jury]] [was [out of [the courtroom]]]]]]]

In (4a), the clause [while ...] is the sister of [Bellows made the request]. In (4b), it is the sister of [made the request]. Differences such as this are typically of no consequence to the interpretation of the sentence. However, if (4b) is evaluated against (4a), its scores are as follows:

recall=$\frac{9}{11}$=81.8%, precision=$\frac{9}{12}$=75.0%, crossings=1

If a sentence-level evaluation is used, (4b) would be classified as incorrect, even though, from linguistic point of view, it may well be as reasonable as (4a).

3.2. Selective evaluation

Previous evaluation schemes only assess the overall performance of parsers. A more flexible scheme should be able to selectively evaluate parsers with respect to any given types of syntactic phenomena. It would be interesting to know, for example, how well a parser handles conjunctions or how well a parser would perform if prepositional attachments were ignored. Answers to these questions would help to determine the suitability of a parser for a particular purpose.

3.3. Facilitate the diagnosis of incorrect parses

Besides measuring the performance of parsers, another service that should be provided by a parser evaluation scheme is to help developers to improve their parsers by pin-pointing exactly where the errors are. As we have discussed earlier, in previous evaluations schemes, a single attachment error may cause several crossing brackets and several spurious/missing phrases. Given the list of crossing brackets or the sets of spurious/missing phrases, it is not obvious what caused them to occur.

4. Dependency-based Evaluation

In this section, we propose a dependency-based parser evaluation scheme that offers the desirable properties discussed in the previous section. Instead of phrase boundaries, the scheme is based on the comparison between the dependency relations in answers and keys.

4.1. Dependency trees

In a dependency tree, every word in the sentence is a modifier of exactly one other word (called its head), except the head word of the sentence, which does not have a head [Melcuk, 1987]. We use a list of tuples to specify a dependency tree. A tuple has the following format:

`(modifier cat position head [relationship])`

where, `modifier` is a word in the sentence; `cat` is its lexical category; `head` is the word that `modifier` modifies; `relationship` is an optional specification of the type of the dependency relationship between `head` and `modifier`, such as `subj` (subject), `adjn` (adjunct), `cmpl` (complement), `spec` (specifier), *etc.*; `position` indicates the position of the head relative to the modifier. It can take one of the following values: {`<`, `>`, `<<`, `>>`,

(5)

a. [I [saw [[a man] [with [[a dog] and [a cat]]]] [in [the park]]]]

b. [I [saw [[a man] [with [[a dog] and [[a cat] [in [the park]]]]]]]]

c.

Key: (5a)					Answer: (5b)					error
`I`	`N`	<	`saw`	`subj`	`I`	`N`	<	`saw`	`subj`	
`saw`	`V`	*			`saw`	`V`	*			
`a`	`Det`	<	`man`	`spec`	`a`	`Det`	<	`man`	`spec`	
`man`	`N`	>	`saw`	`cmpl`	`man`	`N`	>	`saw`	`cmpl`	
`with`	`P`	>	`man`	`adjn`	`with`	`P`	>	`man`	`adjn`	
`a`	`Det`	<	`dog`	`spec`	`a`	`Det`	<	`dog`	`spec`	
`dog`	`N`	<	`and`		`dog`	`N`	<	`and`		
`and`	`Conj`	>	`with`	`cmpl`	`and`	`Conj`	>	`with`	`cmpl`	
`a`	`Det`	<	`cat`	`spec`	`a`	`Det`	<	`cat`	`spec`	
`cat`	`N`	>	`and`		`cat`	`N`	>	`and`		
`in`	`P`	>	`saw`	`adjn`	`in`	`P`	>	`cat`	`adjn`	`yes`
`the`	`Det`	<	`park`	`spec`	`the`	`Det`	<	`park`	`spec`	
`park`	`N`	>	`in`	`cmpl`	`park`	`N`	>	`in`	`cmpl`	

(6)

a. [I [saw [a man] with [a dog] and [a cat] [in [the park]]]]

b.

Key: (5a)					Answer: (6a)					error
`I`	`N`	<	`saw`	`subj`	`I`	`N`	<	`saw`	`subj`	
`saw`	`V`	*			`saw`	`V`	*			
`a`	`Det`	<	`man`	`spec`	`a`	`Det`	<	`man`	`spec`	
`man`	`N`	>	`saw`	`cmpl`	`man`	`N`	>	`saw`	`cmpl`	
`with`	`P`	>	`man`	`adjn`	`with`	`P`	?			`yes`
`a`	`Det`	<	`dog`	`spec`	`a`	`Det`	<	`dog`	`spec`	
`dog`	`N`	<	`and`		`dog`	`N`	?			`yes`
`and`	`Conj`	>	`with`	`cmpl`	`and`	`Conj`	?			`yes`
`a`	`Det`	<	`cat`	`spec`	`a`	`Det`	<	`cat`	`spec`	
`cat`	`N`	>	`and`		`cat`	`N`	?			`yes`
`in`	`P`	>	`saw`	`adjn`	`in`	`P`	?			`yes`
`the`	`Det`	<	`park`	`spec`	`the`	`Det`	<	`park`	`spec`	
`park`	`N`	>	`in`	`cmpl`	`park`	`N`	>	`in`	`cmpl`	

<<<, ..., *, ?}, where < (or >) means that the head of `modifier` is the first occurrence of the word `head` to the left (or right) of the modifier; << (or >>) means head is the second occurrence of the word `head` to the left (or right) of the modifier. If `position` is '`*`', then the word is the head of the sentence. If `position` is '`?`', then the word's head is unknown (either to the parser or the human analyst who created the parse tree).

The dependency trees of (3a) and (3b) (re-written as (5a) and (5b)) are shown in the first and second column in (5c) respectively.

4.2. Evaluation

Once the answer and the key are both represented as dependency trees, the answer can be scored on a word-by-word basis. Since both the answer and the key assign a head to every word in the sentence, we define the **error count** of the answer to be the number of words that are assigned different heads in the answer than in the key. For example, there is one error in (5b). In contrast, the dependency tree corresponding to the shallow parse tree (3c) (re-written as (6a)) contains 5 unknown heads, each of which is counted as an error (see (6b)).

Given two dependency trees `key` and `answer`, the function `evaluate` (7) returns the error count of the answer.

The error count is a Hamming distance between the answer and the key, because it is the number of dependency relationships that must be altered in order to make the answer identical to the key. Compared with scores such as precision, recall, and the number of crossing-brackets, the error-count score is intuitively more meaningful. Since the phrase boundaries themselves are not used directly in semantic interpretation, it is hard to predict how missing or spurious phrase boundaries affect semantic interpretation. On the other hand, since the semantic dependencies are embedded in the syntactic dependencies, the semantic interpretation process should be more sensitive to the number of missing or spurious syntactic dependencies than the number of missing, spurious or crossed phrase boundaries.

4.3. Converting constituency trees into dependency trees

Since the procedure `evaluate` takes the dependency trees as inputs, whereas almost all the broad-coverage parsers and treebanks use constituency grammars, a crucial issue that must be resolved is how to apply the method to constituency grammars.

In this section, we present an algorithm to transform the constituency trees into dependency trees. If one or

(7)
```
int evaluate(DepTree key, DepTree answer)
{
  errorCount = 0;
  for each word in the sentence {
    if (the position of the key is not equal to '?' and the position
        or the head of the key is not equal to that of the answer)
      errorCount = errorCount + 1;
  }
  return errorCount;
}
```

both of the key and the answer are represented as constituency trees, we first transform them into dependency trees and then evaluate the parser with the resulting dependency trees.

The transformation algorithm is based on Magerman's method for determining heads (lexical representatives) in CFG parse trees [Magerman, 1994, p.64–66]. Following Magerman, the transformation is driven by a Tree Head Table, which contains an entry for every non-terminal symbol in the grammar. Given a node in a constituency tree, the corresponding entry in the Tree Head Table can be used to determine the **head child** of the node (the head child of a node is either its lexical head or a child that dominates its lexical head).

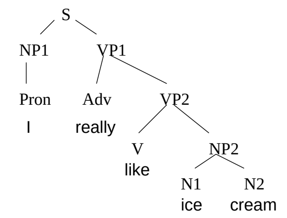
Entries in a tree head table are triples: `(parent direction head-list)`, where `parent` is a grammatical category; `direction` is either `right-to-left` or `left-to-right`; and `head-list` is a list of grammatical categories. Three sample entries are shown in (8).

(8)
```
(S right-to-left (Aux VP NP AP PP))
(VP left-to-right (V VP))
(NP right-to-left (Pron N NP))
```

The first entry means that the head child of an S node is the first Aux node from right to left or, if the S node does not have an Aux child, the first VP node from right to left, ... For example, given the tree head table in (8) and the constituency tree in (9a), the lexical heads and the head children of the nodes in (9a) are listed in (9b).

(9)

a.



b.

node	lexical head	head child
S	V	VP_1
NP_1	Pron	Pron
VP_1	V	VP_2
VP_2	V	V
NP_2	N_2	N_2

The function `findHeadChild` (10) returns the head child of any given node in a constituency tree using the tree head table.

Unlike [Magerman, 1994], where lexical heads of phrases are identified from bottom up, we use a top-down recursive procedure `makeDeps` to construct dependency trees according to parse trees. The procedure returns the lexical head of the tree.

(11)
```
Tree makeDeps(Tree root, DepTree deps)
{
  if (root is a leaf node) return root;
  Tree headChild = findHeadChild(root);
  Tree lexHead = makeDeps(headChild, deps);
  for each non-head child of root {
    lexHeadOfChild = makeDeps(child, deps);
    addDepRel(lexHead, lexHeadOfChild, deps);
  }
  return lexHead;
}
```

The function `addDepRel(head, modifier, depTree)` inserts the dependency between `head` and `modifier` into the dependency tree `depTree`. The main idea of the algorithm is as follows:

- find the head child of the root.
- make a recursive call to construct the dependency tree according to the subtree rooted at the head child and return the lexical head of the head child (which is also the lexical head of the root node).
- for all other children of the root
  - recursively construct a dependency tree according to the subtree rooted at that child and return the lexical head of the child.
  - add the dependency relationship between the lexical head of the root and the lexical head of the child.

5. Modifying dependency trees

In [Black *et al.*, 1991], certain nodes in the answers and keys are erased before they are compared. The erased elements include, for instance, auxiliaries, "not", and pre-infinitival "to". The reason for the removal is that there are many possible ways to analyze structures involving these elements, all of which are correct in their own way. A evaluation scheme should not prefer any one of the theories and penalize the others.

There are many other kinds of allowable differences that may not be eliminated by simply removing elements from parse trees. In this section, we propose a set of operations for modifying dependency trees in a more flexible and principled fashion. We then demonstrate, by means of examples, how these operations can be used to eliminate inconsequential differences, and to allow selective evaluation.

The process of dependency-based parser evaluation is depicted in Figure 1. The `modify` module normalize the

```
(10) Tree findHeadChild(Tree node) /* node is assumed to be interior */
     {
       TreeHeadEntry entry = search_entry(label(node), treeHeadTable);
       for each h in headList(entry) {
         enumerate children of node according to direction(entry) {
           if (label(currentChild)==h) return currentChild;
         }
       }
       if (direction(entry)='left-to-right') return firstChild(node);
       else return lastChild(node);
     }
```

```
(12) void modify(Operations operations, DepTree depTree)
     {
       for each operation (condition, action) in operations
         for each dependency relation dep in depTree
           if (dep satisfies condition) perform action on dep.
     }
```

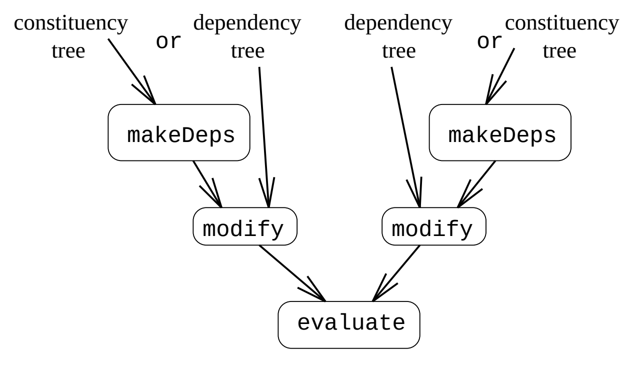


Figure 1: The process of parser evaluation

dependency trees before they are evaluated. The `modify` module consists of a sequence of operations. Each operation specifies a possible alternation to a dependency relationship. It consists of a `condition` part and an `action` part. If a dependency relationship satisfy the condition, the corresponding action will be performed on the dependency. The algorithm for `modify` is shown in (12).

A `condition` is a triple:

`(head modifier [relationship])`

where `head` and `modifier` are restrictions on the head and the modifier of a dependency relationship. The optional `relationship` component is a restriction on the type of the dependency relationship. The first column in Table 1 contains several example conditions. The second column contains the dependency relationships that satisfy the conditions.

The `action` part specifies the modifications to the dependency relationship. We have implemented three types of actions: {deletion, inversion and transfer}

Deletion: `delete(head, modifier, depTree)`
removes the dependency relationship between `head` and `modifier` from the dependency tree `depTree`.

Inversion: `invert(head, modifier, depTree)`
reverses the direction of the dependency relationship between `head` and `modifier`. In the mean time, if `head` also has a head (called `headOfHead`), then the dependency between the `headOfHead` and `head` is replaced with the dependency between `headOfHead` and `modifier`.

Transfer: `transfer(head, modifier, depTree)`
transfers modifiers of `modifier` to `head`. In other words, all the modifiers of `modifier` now become modifiers of `head`.

Figure 2 shows an example of each of these actions.

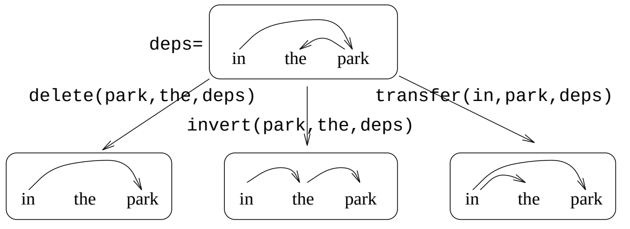


Figure 2: Modifying dependency relationships

In the remainder of this section, we demonstrate how these modifications can be used to eliminate inconsequential differences, and to allow selective evaluation.

5.1. Eliminating inconsequential differences

Different grammars often treat adverbs differently. For example, in "she will leave soon", the adverb "soon" can either be analyzed as the modifier of "will" (Figure 3a) or "leave" (Figure 3b). If the operation:

`(if ((cat Aux) (cat V)) (invert transfer))`

is applied to both trees, they become identical (Figure 3c). In Figure 3a, the dependency link from "will" to "leave" is first inverted, so that "will" becomes a modifier of "leave". Then, the modifiers of "will" ("she" and "soon") are transferred to "leave", resulting in Figure 3c.

Table 1: Example conditions

Condition	Dependency relationship between:
`((cat N) (cat Det))`	a noun and its determiner
`((cat N) (or (string "a") (string "the")))`	a noun and "a" or "the"
`((cat Conj) t)`	a connective (e.g., "and", "or") and any other word
`(t (cat P) (type adjn))`	any word and its prepositional adjunct

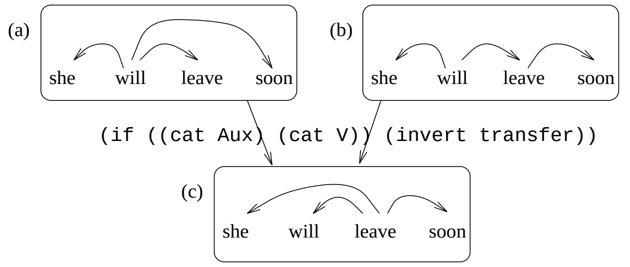


Figure 3: Normalization of auxiliaries

Conjunction is another syntactic phenomenon that tends to be treated differently in different theories. Figure 4 shows three alternative analyses of the dependency tree of "saw A and B." They can be transformed into an identical form by the operations shown in the figure. Note that such variations in the analyses of conjunctions cannot be normalized by simply removing elements from parse trees.

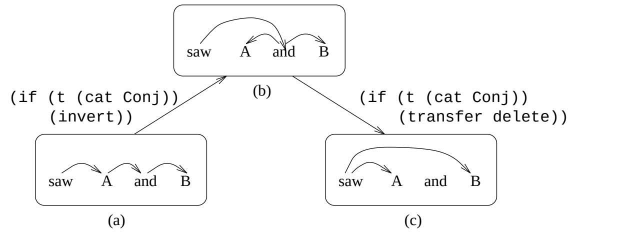


Figure 4: Normalization of conjunctions

5.2. Selective evaluation

The modification to the dependency tree also allows us to selectively evaluate the performance of parsers with regard to various syntactic phenomena. For example, if we want to find out how successfully a parser deals with prepositional phrase attachments, we can use the following operation to delete all the other dependencies except those in which the modifier is a preposition:

```
(if (t (not (cat P))) (delete))
```

On the other hand, evaluating the result of applying

```
(if (t (cat P)) (delete))
```

to dependency trees would tell us how a parser would fare if attachments of prepositional phrases are ignored.

6. Conclusion

We have presented a dependency-based method for evaluating broad-coverage parsers. The method offers several advantages over previous methods that relied on the comparison of phrase boundaries. The error count score is not only more intuitively meaningful than other scores, but also more relevant to semantic interpretation. We also presented an algorithm that transforms constituency trees into dependency trees so that the evaluation method is applicable to both dependency and constituency grammars. Finally, we proposed a set of operations for modifying dependency trees that can be used to eliminate inconsequential differences among different parse trees and allow us to selectively evaluate different aspects of a parser.

References

[Black *et al.*, 1991] E. Black, S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B Santorini, and T. Strzalkowski. A procedure for quantitatively comparing the syntactic coverage of english grammars. In *Proceedings of Speech and Natural Language Workshop*, pages 306–311. DARPA, February 1991.

[Black *et al.*, 1992] Ezra Black, John Lafferty, and Salim Roukos. Development and evaluation of a broad-coverage probabilistic grammar of English-language computer manuals. In *Proceedings of ACL-92*, pages 185–192, Newark, Delaware, 1992.

[Magerman, 1994] David M. Magerman. *Natural Language Parsing as Statistical Pattern Recognition*. PhD thesis, Stanford University, 1994.

[Melcuk, 1987] Igor A. Melcuk. *Dependency syntax: theory and practice*. State University of New York Press, Albany, 1987.