# Towards a Multi-Representational Treebank

Fei Xia
University of Washington
Seattle, WA 98195, USA
fxia@u.washington.edu

Owen Rambow
Columbia University
New York, NY 10115, USA
rawbow@ccls.columbia.edu

Rajesh Bhatt
University of Massachusetts
Amherst, MA 01003, USA
bhatt@linguist.umass.edu

Martha Palmer
University of Colorado
Boulder, Colorado 80309, USA
Martha.Palmer@colorado.edu

Dipti Misra Sharma
International Institute of Information Technology
Gachibowli, Hyderabad 500019, India
dipti@iiit.ac.in

**Abstract**

Computational, descriptive, and theoretical linguistics use both phrase (PS) structure and dependency structure (DS) to represent syntax. We believe that the next-generation treebank should be multi-representational, designed for both representations with an automatic conversion. In this paper, we highlight the assumptions made by existing PS-to-DS and DS-to-PS conversion algorithms and show the limitations of these algorithms. We then propose a new DS-to-PS conversion algorithm that outperforms existing algorithms and allows more flexibility. Our experiments and error analysis show that high-quality DS-to-PS conversion is possible if the conversion process is performed at the designing stage of treebank construction to ensure that all information we wish to represent in PS is provided in DS.

## 1 Introduction

Treebanks have profoundly changed and advanced the field of NLP, and parsing in particular. While most studies on statistical parsing in the 1990s used phrase-structure (PS) treebanks, recently there has been an explosion in interest in dependency structure (DS) parsing (e.g., [1, 5, 6, 2]) which require dependency treebanks. Therefore, we claim that the next-generation treebank should be *multi-*

*representational*; that is, the treebank should document clearly *what* information represented in the the treebank, and *how* it is represented in PS and in DS. If PS and DS represent the same information, then automatic conversion between them should be possible.

We are currently building a multi-representational treebank for Hindi/Urdu with the following strategy: at the guideline designing stage, we develop coordinated DS and PS annotation guidelines. A DS-to-PS conversion algorithm is developed and tested on a test suite and the examples used in the guidelines to ensure that PS and DS guidelines are consistent and that sufficient information (e.g., a rich set of dependency types) is included in the input DS to allow high-quality conversion. At the annotation stage, we will manually create DSs for the sentences, and the corresponding PSs will be generated automatically from the DSs by applying the same DS-to-PS conversion algorithm. We choose the DS-to-PS direction, rather than the other way around, because we believe that annotating DS is faster than annotating PS, and because of we can build on existing work [7]. The success of the strategy depends on the answers to three crucial questions:

**(Q1)** What does consistency between DS and PS mean? Are DS and PS always consistent?

**(Q2)** Is it possible to achieve high accuracy for DS-to-PS conversion?

**(Q3)** What kinds of information should be included in the DS to help with the conversions?

In this paper, we address these questions by first discussing the relation between PS and DS and defining consistency between the two, and then proposing a new DS-to-PS algorithm that outperforms existing algorithms and allows more flexibility. Finally, through an error analysis on the experiments, we show that high-quality DS-to-PS conversion is possible if the conversion process is performed at the designing stage of treebank construction to ensure sufficient information is provided in the input DS.

## 2 Relation between DS and PS

A *syntactic theory* (in the large sense) is an account of how to represent all sentences of a language in the chosen representational framework (PS or DS); there are many possible syntactic theories for each language, for both PS or DS. Some may be better than others, but there is no single PS or DS for any sentence, in any language. Figure 1 shows possible phrase and dependency structures for an example sentence: the former uses as its theory the English Penn Treebank guidelines and the latter is based on an extended version of various deep-syntactic representations.

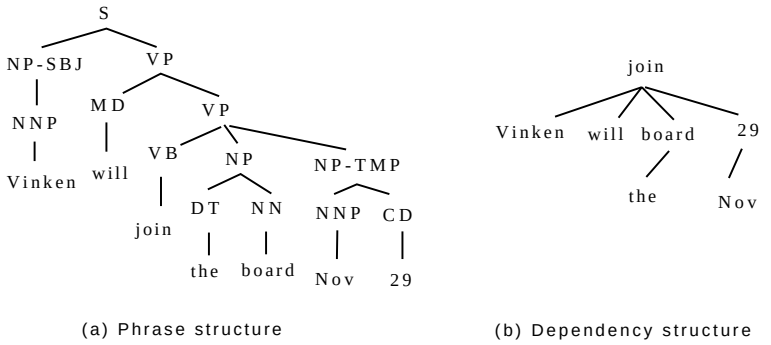


Figure 1: PS and DS for the sentence *Vinken will join the board Nov 29*

The two representations are clearly different. Formally, both PS and DS are types of trees: a DS is a tree whose nodes are all labeled with the symbols appearing in the sentence;[1] a PS is a tree whose leaf nodes are labeled with the symbols in the sentence and whose internal nodes are labeled with syntactic categories (e.g., *NP* four noun phrase). Linguistically, a PS groups consecutive words hierarchically into phrases (or constituents). In a DS, dependency between a head and its dependents is the primary relation represented, and the notion of constituent is only derived. Therefore, PS can represent projections (e.g., a noun phrase as a projection of nouns) directly as edges in the tree, but DS cannot. Similarly, DS can represent dependency relation directly as edges in the tree, but PS cannot.

Other differences that one often finds between PS and DS are *preferential*, not *definitional*. Both representations can have certain properties often believed to be characteristic of one type or the other. For instance, both can use or not use empty categories; both can add labels to edges; both can allow or disallow crossing edges; both (as trees) can be ordered or unordered.

A common PS-to-DS conversion algorithm, used by many parsing algorithms (e.g., [3]) to find head in a PS, works as follows: first, for each internal node in the PS, one of its children is chosen as the head child; next, starting from the leaf nodes of the PS, the head word is propagated from the head child to its parent; finally, a DS is created by making the head word of each non-head-child depend on the head word of the head-child. Given the PS in Figure 1(a), Figure 2(a) shows the tree after the second step, and the resulting DS is in Figure 1(b). This algorithm assumes that the input PS, once *flattened* and with syntactic labels removed, is identical to the desired DS. By *flatten*, we mean that all the internal nodes in the PS are merged

[1]For the sake of this definition, we assume that the surface sentence can (but need not) contain empty categories (i.e., symbols with null phonological value such as trace).

with their head child. In this example, after flattening the PS in Figure 2(a), the new PS, as shown in Figure 2(b), is identical to the DS in Figure 1(b) except for the syntactic labels.

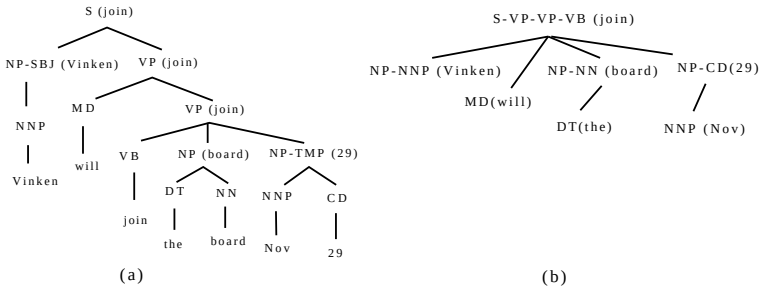


Figure 2: A PS after head word is propagated from bottom up and the flattened version of the PS

A PS can correspond to several flattened trees depending on what node is chosen as head child. We call a PS and a DS *consistent* if there exists a flattened tree for the PS that is identical to the DS once syntactic labels are removed. Given a sentence, if the desired PS and the desired DS are consistent, we say that the *consistency assumption* holds for the sentence. The question is whether consistency assumption always holds. The answer would depends on linguistic choices made in designing the PS and DS. Taking long-distance wh-movement as an example, Figure 3(a) is the PS according to the English Penn Treebank [4]. It uses coindexation with an empty category to represent the long-distance dependency. In contrast, (b) is a DS that uses non-projectivity to represent the long-distance dependency. It is easy to show that the DS and the PS in (a) are inconsistent.[2]

Some of the inconsistency between PS and DS can be handled by creating a new DS that is consistent with the PS. For instance, we can make the DS in 3(b) consistent with the PS in (a) by using the same device to represent the long-distance depend as the DS, namely coindexation with an empty category, as shown in 3(c). (We could also have used non-projectivity in the PS.) In our Hindi/Urdu treebank project, we plan to test whether *all* kinds of inconsistency between PS and DS can be handled by creating a new DS′ which is consistent with the desired PS and whether the process can be totally automated. We expect this to be the case if all information that we want to represent in PS is also represented in DS or can be derived from the syntactic theory underlying the PS.

[2]In order for *who* to depend on *come* as in Figure 3(b), *come* has to be head of the whole sentence, which contradicts with the fact that *think* is the root node in the DS.

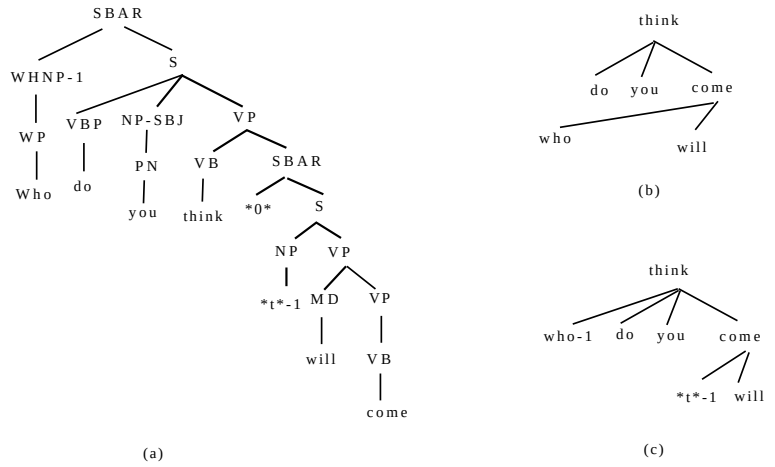


Figure 3: The PS and two DSs for *Who do you think will come*

# 3 A new DS-to-PS algorithm

Xia and Palmer [8] compared three DS-to-PS algorithms, which all could be seen as the reverse of the *flattening* process in the PS-to-DS algorithm: the algorithms expanded each node in the input DS into a projection chain, and labeled the newly inserted nodes with syntactic categories. They differ in the heuristics adopted to build projection chains and attach the PS subtree for a dependent to the subtree for the head.

These algorithms have two major limitations. First, they all make additional assumptions besides the consistency assumption; for instance, they all assume that each POS tag has exactly one possible projection chain. The assumptions turn out to be too strong and the exceptions to the assumptions are the main sources of errors [8]. Second, the input DS may use a rich set of dependency types, but the algorithms cannot take advantage of the information because they distinguish only arguments and modifiers. We propose a new algorithm that overcomes these limitations. (Like the previous algorithm, we assume the DS is consistent with the target PS.) Compared to existing algorithms, the new algorithm only assumes that the consistency assumption holds and any exception to the assumption is handled by a preprocessing step that generates a new DS that is consistent to the desired PS (see Section 2). The consistency assumption states that the input DS is identical to a flattened version of the desired PS; therefore, there is a mapping between the segments in the DS and the segments in the PS. Based on this assumption, the new algorithm works by decomposing the input DS into multiple DS segments,

replacing each DS segment with the PS counterparts, and *glue* these PS segments together to form a PS.

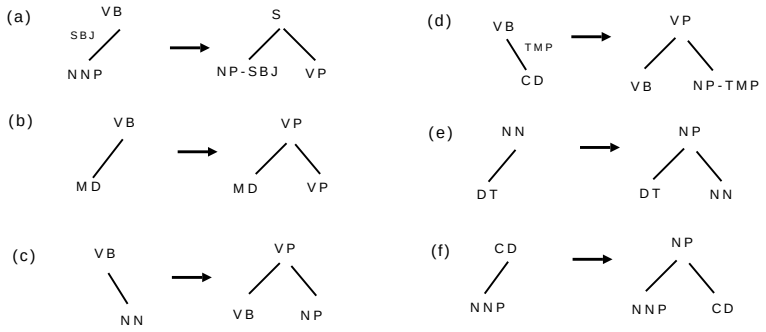


Figure 4: Conversion rules derived from the PS in Figure 2(a)

## 3.1 Conversion rules

A conversion rule is a (DS pattern, PS pattern) pair. In our current experiments, the DS pattern is simply a dependency link with associated information such as dependency type and the POS tags of the head and the dependent, and a PS pattern is one-level tree with a root node and two children. In Figure 4(a), the DS pattern means that an NNP depends on a VB and the dependency type is SBJ. The corresponding PS pattern includes a parent node *S* and two of its children *NP* and *VP*; the function tag of the *NP* comes from the dependency type in the DS pattern.

Conversion rules can be created by hand, or extracted from existing PS. The rule extraction algorithm is the same as the PS-to-DS algorithm in Section 2 except that the last step is replaced with the following, as illustrated in Figure 5: for each internal node ZP in the PS with more than one child, let XP be ZP's head child and X be the POS tag of XP's head word. For every non-head child YP of ZP, let Y be the POS tag of YP's head word, the conversion rule in Figure 5(b) is created. Figure 4 shows the rules extracted from the PS in Figure 2(a).

## 3.2 Applying conversion rules to form PS

Given a DS, the new conversion algorithm selects a conversion rule for each dependency link, and combines the PS patterns of the rules to form the output PS. The algorithm is as follows:

Input: An input DS and a set of conversion rules
Output: An output PS

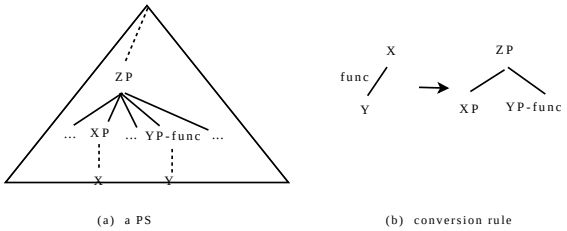


Figure 5: Extracting conversion rules from a PS

Algorithm:
(1) if X, the root of the DS, is a leaf node
(2) the output PS contains only the node X; return
(3) for each child Y of X
(4) build a PS $T_Y$ for the DS subtree rooted at Y
(5) initialize a projection chain, proj_chain, that contains only X
(6) for each left child Y of X in the DS, starting from right to left
(7) (a) choose a conversion rule r for (Y,X) based on the projection chain
(8) (b) apply the rule by updating the projection chain and attaching $T_Y$
(9) Same as (6)-(8), but it is for each right child Y of X, going from left to right
(10) Consolidate the dominant links in the projection chain

Given the DS in Figure 6(a) and the conversion rules in Figure 4, Line (3)-(4) builds the PS trees for the dependents of *join*, as shown in Figure 6(b1)-(b4). Line (5) builds the initial projection chain as in Figure 7(a), and each iteration of Line (6)-(8) (and (9) for the right children) will attach one subtree in Figure 6 to the chain, and update the chain accordingly, as shown in 7 (b) through (e). The circle marks the lowest position on the projection chain that allows future attachment of subtrees for dependents.

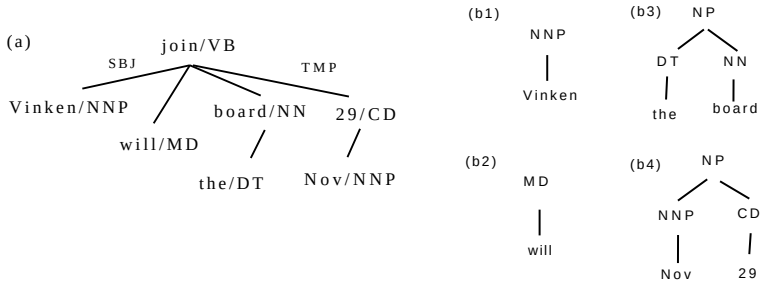


Figure 6: The input DS for the new algorithm and the subtrees built by Line (3)-(4) of the new algorithm

# 4 Experimental results

Ideally, to test a DS-to-PS algorithm, we shall use a treebank with both DS and PS representations. However, since such a treebank does not exist, we followed the same practice as in [8]: we first convert the PS in the English Penn Treebank (PTB) to DS with the PS-to-DS algorithm in Section 2 where the head child is chosen according to a head percolation table [3], then build a new PS from the DS with the new DS-to-PS algorithm, and finally compare the new PS with the original PS.

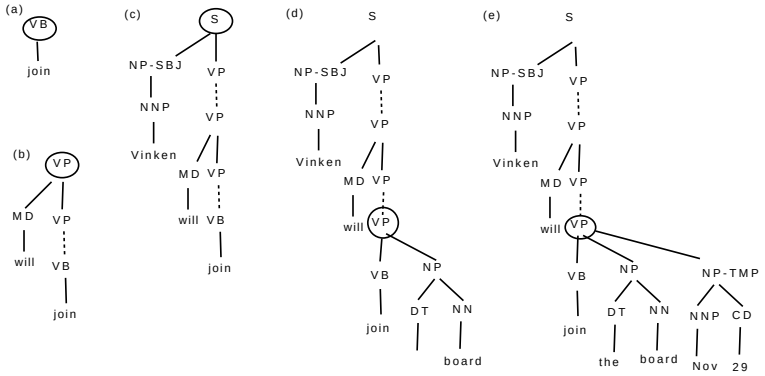


Figure 7: Results after attaching the dependents of *join*

## 4.1 Creating and selecting conversion rules

If multiple rules in a conversion rule set share the same DS pattern, we call the DS pattern *ambiguous*. The ambiguity can be reduced by including more information in the DS pattern. For instance, a DS pattern can include

**(P1)** the POS tags of the dependent and the head, and the position (left or right) of the dependent with respect to the head.

**(P2)** all the information in (P1) plus the dependency type (e.g., -SBJ).

**(P3)** all the information in (P2) plus a flag indicating whether the dependent is a leaf node in the DS.

**(P4)** all the information in (P3) plus a flag indicating whether there are other dependents between this dependent and the head in the DS.

For dependency links in the input DS that match multiple rules, one could build a sophisticated model for rule selection; however, because we believe that most, if

Table 1: Conversion results on Section 22 (*Labeled* precision/recall/fscore)

| | # of rules | # of patts | Non-cheating | | Cheating | |
|---|---|---|---|---|---|---|
| | | | (S1) | (S2) | (C1) | (C2) |
| (P1) | 1841 | 965 | 72.9/90.5/80.7 | 81.9/86.4/84.1 | 85.5/87.6/86.5 | 100/98.2/99.1 |
| (P2) | 2507 | 1645 | 75.0/92.6/82.9 | 83.6/88.9/86.2 | 88.1/90.7/89.4 | 100/98.2/99.1 |
| (P3) | 2826 | 1996 | 79.7/92.4/85.6 | 86.9/90.5/88.7 | 91.2/92.5/91.8 | 100/98.2/99.1 |
| (P4) | 3513 | 2626 | 80.8/92.8/86.4 | 88.1/90.7/89.4 | 92.9/92.9/92.9 | 100/98.2/99.1 |

not all, of the ambiguity could be eliminated for our treebank project (c.f. Section 4.2), we adopted two simple strategies as follows:[3]

**(S1)** Always choose the most frequent rule for the DS pattern assuming the frequency of rules is available from the training data.

**(S2)** Choose the rule based on the current projection chain that is under construction. The algorithm prefers rules that add fewer number of nodes and attach the subtree lower.

## 4.2 Results and Error Analysis

We extracted conversion rules automatically from the phrase structures in one section, Section 19, of the PTB, and ran the new conversion algorithm on another section, Section 22, of the PTB.[4] The results are in Table 1. The experiment shows that when more information is included in the DS pattern, the number of rules increase, as does the performance of the algorithm. Also, (S2) outperforms (S1) for all the rule sets.

As [8] reported the results of Algorithms 1-3 on Section 00, we ran our algorithm on the same section with the (P4)+(S2) setting (the rules were extracted from Section 19). The results are shown in Table 2. Here, we use *unlabeled* precision/recall/fscore following [8]. The experiments show that the new algorithm outperforms all the previous algorithms.

The errors made by our system come from three sources:

**(R1) Missing rules:** Some of the conversion rules needed to produce the gold standard PS for the test data are not in the rule set.

**(R2) Wrong rules selected:** The correct conversion rules are in the rule set, but are not selected by the system.

[3]For dependency links in the input DS that do not match any rules, the algorithm forms a conversion rule on the fly to ensure that a PS is produced.

[4]We did not use a larger data set for rule extraction because using more data or different data does not change the results substantially.

Table 2: Conversion results on Section 0 (*Unlabeled* precision/recall/fscore)

| | Precision | Recall | Fscore |
|---|---|---|---|
| Alg1 | 32.81 | 81.34 | 46.76 |
| Alg2 | 91.50 | 54.24 | 68.11 |
| Alg3 | 88.72 | 86.24 | 87.46 |
| New alg | 89.19 | 91.76 | 90.46 |

**(R3) Exception to the preferences:** For ambiguous DS patterns, the algorithm prefers rules that add fewer number of nodes and attach subtrees lower. Gold-standard PS might have different preferences.

To tease apart the effect of each type of errors, we ran two cheating experiments: in (C1), the *missing* rules are added to the rule set; in (C2), for each dependency link in the input DS, the correct conversion rule for the link is provided as part of the input. The results with (S2) for rule selection are shown in the last two columns of Table 1. In this table, the gap between (C1) and (S2) is caused by (R1), the gap between (C1) and (C2) is caused by (R2), and the gap between (C2) and 100% is caused by (R3). The experiments show that (R2) is the main source of errors, and the f-score in (C2) is very close to 100%, implying that the heuristics adopted by the algorithm to glue PS segments together work very well.

Because most errors are caused by (R2), it is important to understand what causes ambiguity in DS patterns and whether the ambiguity can be eliminated. To answer the question, we compare the rules selected by the gold standard with the rules selected by the best non-cheating system (i.e., (P4)+(S2)).Out of the 42,675 dependency links in Section 22, the system chooses wrong rules for 3,407 links, resulting in an error rate of 7.98% in selecting rules. These correspond to 1,043 link types. We manually checked the most frequent 100 link types (which account for 53.07% of the link tokens) and classified them into four categories.

**Missing content** (13.2% of error tokens): If the PS makes a syntactic distinction which the DS does not make, then the conversion from DS to PS cannot be correct. For example, if the PS distinguishes an SBAR relative clause from an SBAR complement clause by attaching them at different projections of a noun, and the DS does not make this distinction (for example, using arc labels), then there can be no automatic conversion from that DS to that PS.

**Coordination** (14.7% of the error tokens): It is a well known fact that it is not easy to distinguish scope in coordination structurally in a DS (for example, the distinction between "*young (men and women)*" and "*(young men) and women*)". This is another case of missing information: if syntactic structure is not annotated at DS, it cannot be automatically created at PS.

**Inconsistency in the PTB** (7.9% of the error tokens): For instance, when an

adverb appears between the subject NP and the verb V, the adverb may or may not project to an *AdvP*, and the *AdvP* may attach to a *VP* or a *S*. As a result, a DS pattern may correspond to several PS patterns. This type of ambiguity is a by-product of the current experiment setup: conversion rules are extracted from a PS treebank that contain this kind of inconsistency, and automatic creation of the PS from DS will in fact eliminate such inconsistencies.

**Punctuation** (17.4% of error tokens): When the dependent is a punctuation mark, the DS pattern is often very ambiguous because punctuation marks can appear in many places and it is hard to provide an informative dependency type between them and their heads, a problem that we plan to study in the future.

## 5 Conclusion and future work

Because of the usefulness of PS and DS, we believe that the next-generation treebank should be multi-representational. We propose to build such a treebank by creating PS and DS guidelines simultaneously (along with rules that convert DS to PS), building DS manually and then converting DS to PS automatically. In this paper, we address several questions that are crucial for the success of this strategy. First, we define the consistency between PS and DS, and show that the consistency assumption, adopted by all existing DS-to-PS and PS-to-DS conversion algorithms, does not always hold. Second, we propose a new DS-to-PS conversion algorithm that is more flexible and outperforms previous algorithms; furthermore, conversion rules used by the algorithm can be extracted automatically from PS. Third, the error analysis shows that most, if not all, of ambiguity in DS patterns could be eliminated if DS and PS guidelines are designed at the same time (to ensure that we represent the same information at DS and PS). We can use the DS-to-PS conversion process during the guideline design time to detect potential missing information in the input DS and inconsistency between PS and DS.

For the future work, we plan to implement a preprocessor to transform the DS into a PS-consistent $DS'$; and to test and improve the new algorithm while working on the Hindi/Urdu treebank project. For instance, the tree examples in the PS and DS guidelines would serve as the initial test set for our conversion algorithm. We will run the rule extraction algorithm to detect any ambiguous DS patterns and determine whether the ambiguity could be totally eliminated. We will also study whether the DS pattern needs to be extended beyond a single dependency link.

## References

[1] J. Hajič and E. Hajicova and M. Holub and P. Pajas and P. Sgall and B. Vidova-

Hladka and V. Reznickova. The Current Status of the Prague Dependency Treebank. In *Lecture Notes in Artificial Intelligence (LNAI)*, volume 2166, pages 11–20. Berlin, Heidelberg, New York, 2001.

[2] Terry Koo, Xavier Carreras, and Michael Collins. Simple semi-supervised dependency parsing. In *Proceedings of ACL-08: HLT*, pages 595–603, Columbus, Ohio, June 2008.

[3] David M. Magerman. Statistical Decision-Tree Models for Parsing. In *Proc. of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL-1995)*, Cambridge, Massachusetts, USA, 1995.

[4] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a Large Annotated Corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[5] Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. Non-projective dependency parsing using spanning tree algorithms. In *Proc. of HLT/EMNLP*, 2005.

[6] J. Nivre, J. Hall, and J. Nilsson. Maltparser: A data-driven parser-generator for dependency parsing. In *In Proceedings of LREC*, pages 2216–2219, 2006.

[7] Dipti Misra Sharma, Rajeev Sangal, Lakshmi Bai, and Rafiya Begam. Anncorra: Treebanks for indian languages (version - 1.9). Technical report, Language Technologies Research Center IIIT, Hyderabad, India, 2006.

[8] Fei Xia and Martha Palmer. Converting Dependency Structures to Phrase Structures. In *In the Proc. of the Human Language Technology Conference (HLT-2001)*, San Diego, CA, 2001.