Transformation and Combination in Data-Driven Dependency Parsing
Transformation and Combination in Data-Driven Dependency Parsing

Jens Nilsson

Växjö University Press
Abstract


This thesis deals with automatic syntactic analysis of natural language text, also known as parsing. The parsing approach is data-driven, which means that parsers are constructed by means of machine learning, looking at training data in the form of annotated natural language sentences. The syntactic framework used in the thesis is dependency-based. Robustness is one of the characteristics of the data-driven approaches investigated here. The overall aim of this thesis is to maintain robustness while increasing accuracy.

The content of the thesis falls naturally into two tracks, a transformation track and a combination track. The first type of transformation investigated is called pseudo-projective, because it enables strictly projective dependency parsers to recover non-projective dependency relations. Informally, a non-projective dependency tree contains crossing binary directed relations, when drawn above the sentence. Experimental results show that pseudo-projective transformations can improve accuracy significantly for a range of languages. The second type of transformation aims to facilitate the processing of specific linguistic constructions such as coordination and verb groups. Experimental results again show a positive effect on parsing accuracy for several languages, often greater than for the pseudo-projective transformations. However, the improvement of the transformations depends on the internal structure of the base parser, which is not the case for the pseudo-projective transformations.

The combination track compares various approaches for combining data-driven dependency parsers, again as a means of improving accuracy. As different parsers have different strengths and weaknesses, making parsers collaborate in order to find one single syntactic analysis may result in higher accuracy than any of the syntactic analyzers can produce by itself. The experimental results show that accuracy improves across languages, given that appropriate parsers are combined. The thesis ends with an attempt to combine the two tracks, showing that combining parsers with different tree transformations also increases accuracy. Moreover, this experiment indicates that high diversity among a small set of parsers is much more important than a large number of parsers with low diversity.

**Keywords:** Natural Language Parsing, Dependency Parsing, Tree Transformation, Parser Combination
Sammanfattning

Denna avhandling handlar om syntaktisk analys av naturligt språk, också kallat för parsning. Den övergripande strategin är datadriven, vilket innebär att syntaktiska analysatorer (eller parsrar) skapas med hjälp av maskininlärning som utnyttjar träningsdata i form av uppmärkta meningar på naturligt språk. Det syntaktiska ramverket som används i avhandlingen är dependensbaserat. En av de utmärkande egenskaperna hos datadrivna parsrar är att de är robusta. Det övergripande målet för avhandlingen är att bibehålla robusthet och samtidigt förbättra kvaliteten i den syntaktiska analysen.


Till Hanna och Alma
Acknowledgments

I first of all want to thank Joakim Nivre for being my supervisor, and for the help along the way to the completion of this thesis. His elaborate reading of various versions of the thesis has been very important for improving the quality. His guidance has helped me into promising and fruitful directions from the beginning of my time as a Ph.D. student. I am also grateful to my assistant supervisor, Welf Löwe, for providing me with insightful comments on the thesis, and for making me realize that dependency parsing can be useful for more things than just natural languages.

Many thanks are directed to the other members of our small MALT group, Johan Hall and Susanne Ekeklint. The majority of the results presented in this thesis are based on MaltParser, mainly implemented by Johan. Thanks for a very productive collaboration for the last five to six years. I also want to thank my other colleagues in computer science at MSI, Växjö University, especially Jonas Lundberg for reading and commenting on my thesis and accepting the invitation to be my opponent at my final seminar. Niklas Brandt, Morgan Ericsson and Marcus Edvinsson deserve my appreciation for administering the Unix and Linux systems used for most of the experiments presented in this thesis.

I am grateful to MSI, Växjö University for their financial support. This thesis would not have been completed without it. I would also like to express my appreciation to the entire administrative staff, especially our research secretary Gunn Jensen, for making everyday life for Ph.D. students at MSI run smoother.

Mort importantly, thanks to my parents, and my little family – Hanna and Alma – for making every day special.
Contents

1 Introduction 1
1.1 The Transformation Track ................................ 2
1.2 The Combination Track ................................... 4
1.3 Error Propagation ........................................ 4
1.4 Outline .................................................. 5
1.5 Publications ............................................. 6

2 Background 7
2.1 Parsing Natural Language ............................... 7
2.2 Dependency Structure ................................. 9
  2.2.1 Dependency Theories ................................ 9
  2.2.2 Dependency Trees ................................ 11
  2.2.3 Projectivity ........................................ 13
2.3 Data-Driven Dependency Parsing ..................... 14
2.4 Parsing Discontinuous Constructions ............... 19
  2.4.1 Dependency-Based Parsing ......................... 20
  2.4.2 Constituency-Based Parsing ....................... 21
2.5 Tree Transformations ................................. 22
  2.5.1 Dependency-Based Parsing ......................... 22
  2.5.2 Constituency-Based Parsing ....................... 23
2.6 Combining Dependency Parsers ...................... 25

3 Methodological Preliminaries 27
3.1 Evaluation Metrics .................................. 27
3.2 Statistical Significance .............................. 28
3.3 Data Sets ............................................. 29
  3.3.1 CoNLL 2006 Data ................................ 31
  3.3.2 CoNLL 2007 Data ................................ 32
3.4 Software .............................................. 32
  3.4.1 MaltParser ....................................... 32
  3.4.2 MaltEval ......................................... 34

4 Detecting Error Propagation 35
4.1 Four Dependency Parsers .............................. 35
4.2 Error Propagation Metrics ......................... 37
  4.2.1 Error Propagation Metrics ....................... 37
  4.2.2 Pre and Post ..................................... 38
  4.2.3 Error Clustering ................................ 40
4.3 Conclusion ............................................. 42

5 Pseudo-Projective Transformations 43
5.1 Pseudo-Projective Transformations .................. 43
  5.1.1 Step 1: Projectivization ......................... 44
  5.1.2 Encodings ....................................... 47
  5.1.3 Step 4: Deprojectivization ...................... 49
  5.1.4 Back-off for the Deprojectivization .......... 50
5.2 Experimental Setup .................................. 51
5.3 Treebank Transformations ............................ 53
5.4 Parsing Experiments ........................................ 55
  5.4.1 Applying the Pseudo-Projective Transformations .... 56
  5.4.2 Precision and Recall ................................... 58
  5.4.3 Learning Curves ....................................... 59
  5.4.4 Related Work with Other Parsers ...................... 60
5.5 Conclusion .................................................. 62
6 Coordination and Verb Group Transformations ................. 63
  6.1 Coordination Transformations ............................. 63
    6.1.1 Analysis of Coordination ............................ 64
    6.1.2 Step 1: Transformations ............................. 66
    6.1.3 Step 4: Inverse Transformations .................... 69
  6.2 Verb Group Transformation ................................ 71
    6.2.1 Analysis of Verb Groups ............................ 71
    6.2.2 Transformation and Inverse Transformation ......... 72
  6.3 Experimental Setup ...................................... 74
  6.4 Treebank Transformations ................................ 75
  6.5 Parsing Experiments .................................... 77
    6.5.1 Applying the Transformations ...................... 77
    6.5.2 Combining Transformations ........................... 79
    6.5.3 Precision and Recall ............................... 80
    6.5.4 Why do these Transformations Help? ................. 81
    6.5.5 Error Propagation ................................. 83
    6.5.6 Related Work with Other Parsers .................... 85
  6.6 Conclusion ............................................... 86
7 Combining Classifiers ......................................... 89
  7.1 Combination Methods ...................................... 89
  7.2 Experimental Setup ...................................... 92
  7.3 Classifier Combination Experiments ....................... 92
    7.3.1 Combining Classifiers ................................ 93
    7.3.2 Error Propagation .................................. 94
    7.3.3 Detailed Analysis .................................. 95
  7.4 Conclusion ............................................... 97
8 Combining Transition-Based Parsers ........................... 99
  8.1 Single Systems .......................................... 99
  8.2 Experimental Setup ...................................... 100
  8.3 Algorithm Combination Experiments ....................... 102
    8.3.1 Parsing Direction and Parsing Algorithm ........... 102
    8.3.2 Combining Parsers .................................. 104
    8.3.3 Error Propagation .................................. 105
    8.3.4 Detailed Analysis .................................. 106
  8.4 Conclusions ............................................. 109
**List of Tables**

3.1 Information about the data sets .................................. 30
4.1 **Pre, Post** and **Total** error rate for MaltParser, LTH-parser, MSTParser and ILP-parser ............................ 39
4.2 Error clustering in CoNLL-X for MSTParser, MaltParser, LTH-parser and ILP-parser ........................................ 40
4.3 Proportion of tokens in the bins for table 4.2 and figure 4.1 . . 40
5.1 Projectivity encodings ............................................. 49
5.2 Information about non-projectivity in five of the training sets of CoNLL shared task 2006 ........................................ 52
5.3 Distortion results for the pseudo-projective transformations ... 54
5.4 The number of lifts for non-projective arcs ....................... 54
5.5 The number of distinct pseudo-projective dependency labels ... 55
5.6 Results for the pseudo-projective transformations using MaltParser 56
5.7 Precision and recall for arcs marked as lifted ................... 58
5.8 Learning curve figures for Alpino and Tiger measured as error reduction for AS\(_U\) ........................................... 60
6.1 Information about coordination and verb groups in four of the data sets of CoNLL shared task 2006 ......................... 74
6.2 Distortion results for coordination and verb groups ............... 75
6.3 Parsing results for coordination and verb group transformations using MaltParser ........................................... 78
6.4 Parsing results when combining transformations ................... 79
6.5 Precision and recall for conjuncts, and auxiliary verbs ........... 81
6.6 Normalized AS\(_U\) for **Pre** and **Post** for SDT, PADT, PDT, and corresponding error reduction figures ................ 85
7.1 AS\(_L\) for single parsers for the four test data sets together .... 93
7.2 Results for the various voting schemes for TL and GL ........... 93
7.3 **Pre** and **Post** for SVM(Q) and the best GL combination .... 94
7.4 Error clustering for SVM(Q) and the best GL combination ....... 95
7.5 Precision and recall for root tokens and non-root tokens for SVM(Q) and the Best GL combination for AS\(_L\) .................... 95
8.1 AS\(_L\) for the single systems combined in the algorithm combina
tion systems ..................................................... 102
8.2 The algorithm combination systems results compared to the sin
gle systems ...................................................... 103
8.3 **Pre** and **Post** for the single systems and algorithm combination system ...................................................... 106
8.4 Precision and recall for root tokens and non-root tokens for the single systems, reverse single systems and algorithm combination systems for AS\(_L\) ........................................ 106
9.1 Generic transformations results .................................. 115
List of Figures

1.1 Dependency tree with crossing arcs. ................ 2
2.1 Dependency structure for English sentence. ............ 10
2.2 Two non-projective dependency structures. ............. 14
2.3 Transitions sequence of a dependency tree. .......... 17
2.4 Visualization of potential features of Nivre’s arc-eager parsing algorithm. .................. 18
4.1 Normalized error clustering curve in CoNLL-X for AS_L ................... 41
5.1 Non-projective dependency tree for Czech sentence from PDT. .. 46
5.2 Projectivized dependency tree for the sentence in figure 5.1. ..... 46
5.3 Example illustrating that PROJECTIVIZE is a many-to-one function. .................. 48
5.4 Learning curves for Alpino and Tiger measured as error reduction for AS_U. .................. 61
6.1 Dependency structure for coordination. ................. 65
6.2 Coordination example in PS for a Czech sentence from PDT. .. 67
6.3 Coordination in MS for the sentence in figure 6.2. .......... 67
6.4 Dependency structure for verb groups. ................. 72
6.5 Verb group example in PS for a Czech sentence from PDT. .. 73
6.6 Verb group in MS for the sentence in figure 6.5. .......... 73
6.7 Example of consistency for the coordination transformation ... 82
6.8 Unlabeled precision and recall per arc length for PDT ....... 83
6.9 Proportions of arcs per arc length for verb group and coordina-
tion transformations for PDT. .................... 84
6.10 AS_U clustering curves for Slovene, Arabic, Dutch and Czech ... 86
7.1 Labeling strategies. ................................ 91
7.2 Normalized AS_L error clustering of table 7.4 .......... 96
7.3 Precision for arcs at various depths and dependency lengths for SVM(Q) and Best GL. ......... 97
8.1 Reversed dependency structure for English sentence ........ 101
8.2 Normalized AS_L error clustering for the single systems, the re-
verse single systems and the algorithm combination systems ... 107
8.3 Normalized AS_L error rate for grouped token positions. ... 108
8.4 Precision for arcs at various depths and dependency lengths for SVM(Q) and Best GL. .......... 108
9.1 CHILDSWAP and PARENTSWAP transformations. ......... 113
9.2 LIFTCHILDREN and DESCENDSIBLINGS transformations. .... 114
9.3 Selection strategy, with the number of parsers at the x-axis. .. 118
Chapter 1

Introduction

Natural language processing (NLP) deals with the interaction between computers and natural (human) languages. This Ph.D. thesis is concerned with syntactic analysis of natural languages, which is a subfield of NLP. Many NLP systems can benefit from syntactic information. These include systems for machine translation, question answering and information extraction, where syntactic analysis is regarded as a useful step towards a meaning-bearing, semantic representation.

Automatic syntactic analysis – also known as parsing – in this thesis refers to the process of mapping sentences in unrestricted natural language text (or speech) to their syntactic structure. The parsing of natural language texts can be performed using various syntactic representations. Constituency-based representations have dominated the field of natural language processing for a long time, but the parsers used in this thesis deal with dependency parsing of natural language texts, using a type of syntactic representation which has gained renewed attention during the last decade or so. Figure 1.1 shows an example of a dependency tree, where the arrows, or arcs, represent the syntactic relations that hold between the words.

This thesis deals with supervised data-driven dependency parsing. A supervised data-driven dependency parser requires training data, constituting a collection of sentences annotated with their correct syntactic dependency trees, known as a treebank. The task of the parser is to learn a parsing model by looking at the training data, a model that is used for constructing dependency trees for new unseen sentences. Data-driven parsing is normally implemented as a technique for robust parsing, in the sense that any sentence – syntactically correct or not – will be subjected to some syntactic analysis. However, the price to pay for the robustness is that data-driven parsers can commit errors even if the input sentence is syntactically correct.

The data-driven parsing techniques utilized here are also disambiguating, meaning that at most one syntactic analysis is produced for each sentence. More precisely, robust and disambiguating parsers construct exactly one syntactic structure for each sentence. The overall goal of this thesis is to make use of such dependency parsing techniques while achieving as high accuracy as possible.

Two tracks are discernible in this thesis, one transformation and one combination track. The first track deals with tree transformations, which are
applied to the dependency trees of the training data as well as to the parser output. Constituency-based parsers with state-of-the-art accuracy on unrestricted language texts are trained on syntactically annotated sentences, and a major part of their success can be attributed to extensive manipulations of the training data and the output of the parser, including various tree transformations. The transformation track investigates whether manipulations in the form of tree transformations can improve accuracy for dependency parsers as well.

The combination track is devoted to another prominent trend in data-driven parsing – parser combination. Different data-driven parsers have different strengths and weaknesses. By taking the syntactic analyses of several parsers and merging them into one syntactic analysis, the parsers’ strengths can ideally be exploited while compensating for their weaknesses. The aim is thus to improve accuracy. This second track investigates different strategies for combining dependency parsers.

1.1 The Transformation Track

The general methodology in the transformation track is a four-step procedure. In step 1, tree transformations are applied to the training data. The data-driven dependency parser is then trained in step 2, and new sentences are parsed in step 3. In order to restore the original tree structure, an inverse transformation is applied to the parser output in step 4. The outlined method makes it possible to regard the parser as a replaceable black box. A tree transformation means in this thesis that arcs are moved in the dependency tree.

The first part of the transformation track is devoted to the treatment of non-projective structures in dependency parsing. Informally, a projective dependency tree can be said to be free from crossing dependency arcs. The dependency tree in figure 1.1 is non-projective, since the arc from be to who crosses the arcs from <ROOT> to may and from may to ?.

Somewhat simplified one can say that whereas most existing fast dependency parsers are only able to create projective dependency trees, most linguistic dependency theories are not constrained to projective structures. The amount of non-projectivity is highly dependent on the properties of the
language and on the linguistic theory. As the vast majority of linguistic phenomena do not normally require non-projectivity, using a projective parser is sometimes an adequate approximation. However, it is theoretically unsatisfactory to know that a completely correct analysis is impossible to create for any sentence containing non-projectivity. The first research question pursued in this track focuses on tree transformations and the distinction between projectivity and non-projectivity:

Q1 Is it possible to produce non-projective dependency trees using a data-driven projective dependency parser by means of tree transformations?

The aim of the tree transformations is also tied to the overall goal of the thesis, to achieve as high accuracy as possible. The issue pursued is whether various tree transformations can make it possible to parse non-projective constructions (i.e., increasing the accuracy for non-projective constructions), while maintaining the advantages of projective parsing, such as efficient parsing and high accuracy for projective constructions.

Moreover, a data-driven parser cannot make any assumptions about the structure of individual linguistic phenomena. It is simply forced to use nothing but the information in the treebank, which is the result of the design choices of the treebank annotators. These design choices are usually made on linguistic grounds, often without having automatic syntactic parsing in mind. This may make some linguistic constructions more difficult to parse for certain data-driven parsing algorithms.

Which linguistic dependency constructions are difficult to parse for certain data-driven dependency parsers? In this thesis, the focus is primarily on two linguistic phenomena for which the disagreement of representation in dependency theories is high, namely coordination and verb groups. For instance, in the title of this thesis, the words Transformation and Combination constitute a coordination, and the words may and be in figure 1.1 form a verb group. Essentially, the relationship between the conjunction (and) and the conjuncts (Transformation, Combination), and the relationship between the auxiliary verb (may) and the main verb (be), will be investigated here. Coordination is a linguistic phenomenon that tends to be especially difficult to parse. Choosing an appropriate base representation for parsing is therefore an important task for dependency-based parsing.

As already mentioned, constituency-based parsing studies reveal that preprocessing the data and postprocessing the parser output are important in order to achieve state-of-the-art accuracy. In other words, a linguistically sound constituency-based representation is not necessarily a good representation for data-driven parsing. The research question is whether some corresponding processing of the training data can facilitate the learning task in data-driven dependency parsing:

Q2 Is it possible to improve parsing accuracy for a data-driven parser by means of tree transformations targeting structures that are hard to parse?
1 Introduction

The transformations for Q1 and Q2 become even more useful the more language- and treebank-independent they are. An additional research question is therefore:

Q3 How do the effect of the transformations for Q1 and Q2 vary across languages and treebanks?

The search for tree transformations for improving accuracy is not restricted to just coordination and verb groups in this thesis. A technique for finding other types of beneficial tree transformations is also proposed and evaluated.

1.2 The Combination Track

The parser type used in this thesis is called transition-based. Transition-based dependency parsers often resemble shift-reduce parsers for constituent-based parsing, and parsing is performed from left to right, having a stack for storing partially parsed tokens (words) and a queue containing remaining input tokens. A transition-based parser applies a transition in each parser state, such as manipulating the stack and queue, and produces more and more dependency structure incrementally from left to right. New transitions are applied until a terminal condition is met, e.g., that the queue is empty.

Grammar-based parsers (e.g., ordinary shift-reduce parsers) use a grammar to determine the next transition. A data-driven transition-based parser, on the other hand, uses the training data to learn how to predict the next transition. The parser can therefore make prediction mistakes, which normally leads to a partially incorrect dependency tree.

The data-driven transition-based parsers enable combination in at least two ways, which is the main focus of the combination track. Combination can either take place in each parser state during parsing by choosing one transition decision from a set of parsers, or it can take place after parsing by merging a set of dependency trees from different parsers into one tree. These two types of combination strategies, including several variations within each type of strategy, are compared here. The general research question for the combination track is:

Q4 How can different transition-based dependency parsers be combined in order to increase accuracy as much as possible?

It is worth noting that, although combination in this thesis will only be applied to transition-based parsers, combination after parsing can use all kinds of dependency parsers.

1.3 Error Propagation

The transition-based parsing algorithms used throughout the thesis are also characterized as greedy (or deterministic). That is, only the locally optimal
predicted transition is applied in each parser state, entailing that the parser never explores more than one partial dependency tree simultaneously during parsing. The benefit of the greedy approach is potentially very fast parsing time. However, if a greedy transition-based parser commits an error, the probability that the parser will commit a new error can be assumed to increase. The reason for this is that the transition-based parser is then located in an unknown parser state, and has no knowledge about how to get back on the correct parser state path.

Evaluation metrics directly targeting error propagation have not previously been proposed and used. In this thesis, such evaluation metrics are proposed and defined. The new metrics are first applied to state-of-the-art parsers with various properties, including transition-based parsers, in order to measure and assess the impact of error propagation. The metrics are thereafter applied in the two tracks to investigate the following research question:

Q5 Do the transformations and the various parser combination strategies counteract or aggravate the influence of error propagation for transition-based dependency parsers?

The error propagation metrics will be used as diagnostics in investigating this question, but the final validation of these metrics lie outside the scope of this thesis.

1.4 Outline

The rest of this thesis is structured as follows. Chapter 2 is a background chapter, with an introduction to natural language parsing in general. The chapter also contains a formal definition of dependency structure and dependency parsing, and ends with previous research related to the two tracks. Chapter 3 sorts out a number of methodological issues, such as evaluation metrics, statistical significance tests, data sets and software used throughout the experiments. Chapter 4 presents the error propagation metrics. The metrics are then applied to various dependency parsers for comparison, including transition-based ones.

Chapter 5 is the first chapter in the transformation track and aims at answering research question Q1. The chapter proposes tree transformations dealing with non-projective structure and presents parsing experiments. The transformation track continues in chapter 6 with coordination and verb group transformations, tied to research question Q2. The tree transformations of both chapters are applied to a range of languages and treebanks, in order to investigate research question Q3.

The focus of chapters 7 and 8 is the combination track, and its research question Q4. The experiments presented in chapter 7 investigate how to combine machine learners in various ways in data-driven transition-based
1 Introduction

dependency parsing. Chapter 8 continues by combining various transition-based dependency parsing algorithms, using the best combination approach of chapter 7.

In chapter 9, the two tracks come together in an investigation of parser combination with parsers using different generic tree transformations. The research question concerning error propagation, Q5, permeates most chapters containing parsing experiments. The thesis ends with conclusions and some possible directions for future research in chapter 10.

1.5 Publications

This thesis is primarily based on the following publications:

1. Pseudo-Projective Dependency Parsing
   Joakim Nivre and Jens Nilsson

2. Graph Transformations in Data-Driven Dependency Parsing
   Jens Nilsson, Joakim Nivre and Johan Hall

3. Generalizing Graph Transformations in Data-Driven Dependency Parsing
   Jens Nilsson, Joakim Nivre and Johan Hall

4. Dependency Parsing by Transformation and Combination
   Jens Nilsson and Joakim Nivre

5. Single Malt or Blended? A Study in Multilingual Parser Optimization
   Johan Hall, Jens Nilsson, and Joakim Nivre
Chapter 2

Background

As mentioned in the previous chapter, this thesis explores robust and disambiguating parsing techniques for natural languages using dependency representations. The two tracks—transformation and combination—of the thesis were also introduced. This chapter contains the necessary background for the two tracks handled in the coming chapters. The focus will thus be on data-driven dependency parsing, but other syntactic frameworks are brought in when relevant, especially for related work for the transformation track.

Section 2.1 sets the more general context, providing an overview of natural language parsing. The chapter continues with an introduction to dependency structure in general in section 2.2, which also contains the formal definition of dependency trees and projectivity used throughout the thesis. The following section 2.3 is based on the two preceding sections, discussing existing approaches in data-driven dependency parsing in more detail. Various relevant approaches for the two tracks are presented in the following three sections. For the transformation track, work related to parsing non-projectivity is handled in section 2.4, and tree transformations in data-driven natural language parsing in section 2.5. This chapter ends with section 2.6, presenting various ways to combine data-driven dependency parsers.

2.1 Parsing Natural Language

Parsers for natural languages can roughly be characterized as either grammar-driven or data-driven, even though hybrid approaches exist. Both grammar-driven and data-driven approaches can be further divided into constituency-based and dependency-based ones.

Grammar-driven constituency-based parsers are more common than dependency-based ones. Several different frameworks within constituency-based representation exist, the most basic one being context-free grammar. Grammars for natural languages are normally ambiguous, that is, a sentence can have more than one syntactic analysis according to the grammar. Hence, the basic grammar-driven context-free parsing algorithms, such as CKY (Kasami, 1965; Younger, 1967) and Earley’s algorithm (Earley, 1970), derive all valid parse trees, given an ambiguous context-free grammar.
These algorithms can be easily extended to deal with probabilistic context-free grammar (PCFG), where each non-terminal has an associated weight in order to select the (normally) single correct syntactic analysis (Charniak, 1993). The score of each valid parse tree can be determined by computing the product of the weights of all non-terminals in the parse tree. Disambiguation is then simply a matter of choosing the parse tree with the highest score. More elaborate grammar-driven systems typically use a grammar-driven parser coupled with a statistical model for parse selection, as exemplified by parsers for LFG (Riezler et al., 2002), HPSG (Toutanova et al., 2002), and CCG (Clark and Curran, 2004).

A few grammar-driven dependency-based parsing approaches exist as well, such as Karlsson (1990) for Constraint Grammar and the further development into Functional Dependency Grammar (Tapanainen and Järvinen, 1998). Earlier work on grammar-driven dependency parsing includes, for instance, Hays (1964), whose parser has great similarities to the CKY algorithm for context-free grammar, as both are based on bottom-up approaches using dynamic programming. More recent grammar-driven frameworks are Weighted Constraint Dependency Grammar (WCDG) (Foth et al., 2004) and Discontinuous Grammar (Kromann, 2004), both based on constraint satisfaction.

Constituency-based data-driven parsing approaches have been prominent from the mid 90’s and onwards, mainly due to the new availability of large constituency-based treebanks, such as the Penn Treebank (Marcus et al., 1993). In contrast to grammar-driven parsing, one of the major benefits of data-driven parsing is that robust parsing is easier to achieve. Formal grammatical frameworks, including basic context-free grammar, have difficulties dealing with the fuzziness of natural language, leaving slightly ungrammatical sentences (or grammatically correct sentences not covered by the grammar) without a syntactic analysis. It is natural to assume that some analysis is better than no analysis for many applications such as machine translation, even if the analysis is slightly incorrect.

The previously most accurate constituency-based data-driven parsers trained and evaluated on the Penn Treebank are Collins (1997; 1999) and Charniak (2000). Their parsers are not based on PCFG but rather on so-called Markov grammars, which incorporate billexical dependencies. This is not easily done in a PCFG, even though Markov grammars can be mapped to PCFG. The tree structure constructed by these dependencies is essentially a projective dependency tree, apart from the fact that the arcs are not labeled with dependency labels.

It is worth noting that the parsers of Collins and Charniak output a single parse tree with the highest probability, but are able to produce an $n$-best ranking of the derived syntactic analyses. Current state-of-the-art parsers for the Penn Treebank use discriminative learning methods in order to rerank these syntactic analyses, e.g., Collins and Duffy (2005) and Charniak and Johnson (2005), which results in higher accuracy than simply selecting the parse tree with the highest probability. Another recent trend
in constituency-based data-driven parsing is the use of latent variables to induce more fine-grained PCFGs (Petrov et al., 2006; Petrov and Klein, 2007).

Data-driven dependency parsers have become more popular in recent years, partly due to large-scale multilingual evaluation campaigns such as the CoNLL shared tasks 2006 (Buchholz and Marsi, 2006) and 2007 (Nivre et al., 2007). The two dominant approaches in data-driven dependency parsing are graph-based parsing (Eisner, 1996; McDonald et al., 2005), which is based on global scoring of dependency graphs, and transition-based parsing (Yamada and Matsumoto, 2003; Nivre et al., 2004), which is based on local scoring of parser actions.

Data-driven dependency parsing will be discussed in more detail in section 2.3, preceded by an introduction to dependency structure in the next section. The constituency-based data-driven parsers presented in this section will be further discussed in conjunction with the transformation track, in sections 2.4 and 2.5.

2.2 Dependency Structure

The first modern theory of dependency structure as a means for syntactic representation was created by Tesnière (1959). Constituent structure was formalized by Chomsky (1956) in phrase structure grammar, or context-free grammar. While the notion of constituency cannot be traced further back than the beginning of the twentieth century (Percival, 1976), the notion of dependency has been independently accepted several times in the grammatical history, and as early as two millennia ago (McCawley, 1973).

2.2.1 Dependency Theories

Since Tesnière (1959), a large number of grammatical theories based on the notion of dependency have evolved in various directions, but they all share a core of common concepts and assumptions. The most basic notion is that syntactic structure consists of binary and asymmetrical relations between lexical elements, called dependencies. The asymmetrical property naturally creates a hierarchy, where one of the lexical elements acts as head (or governor) and the other as dependent (or modifier). The most obvious difference compared to constituent structure is the lack of phrase nodes, or non-terminals.

An important part of the creation of a dependency theory is to establish criteria for imposing dependencies between the lexical elements. A first broad distinction is often drawn between syntactic and semantic criteria. Other criteria, such as morphological ones, have also been proposed. Zwicky (1985), for instance, points out that the notion of dependency can easily be extended into the area of morphology by linking lexemes using dependency relations. Fraser (1994) enumerates the following four criteria:
2 Background

Figure 2.1: Dependency structure for English sentence from the Penn Treebank (Marcus et al., 1993), converted to dependency structure by Johansson and Nugues (2007).

1. A head determines whether a dependent is optional or obligatory (as a modifier of the head), and not vice versa.
2. A head subcategorizes for a dependent, and not vice versa.
3. The head determines which inflectional form of a dependent occurs, and not vice versa.
4. The head identifies a meaningful object that the meaning of the dependent contributes to (specifies further), and not vice versa.

Zwicky has a similar partly overlapping list. Figure 2.1 is a dependency tree for an English sentence, where these criteria are applicable. The arc from the noun cucumber to the determiner a indicates that the noun is the head and that the determiner is a dependent. Criterion (2) is not applicable to this relation, but it is supported by (1) and (4), and in a sense also (3) (cf. a banana - an apple, and an apple - many apples).

It is, however, important to note that no single set of criteria provides the necessary or sufficient conditions for establishing dependency relations in all cases. It is also important to keep in mind that different criteria may suggest different solutions. The arc from the verb was to the pronoun she indicates that the verb is the head and that the pronoun is a dependent. The label on the arc further specifies that the noun is a subject of the verb. This is in line with (2), and most syntactic theories say that the verb opens "slots" (subcategorizes), where one slot is filled by a subject. However, it has also been suggested that it is more natural to regard the subject controlling the person inflection of the verb in English, which would instead, according to (3), make the subject she the head and the verb was the dependent.

Mel’čuk (1988), who advocates the multi-stratal Meaning-Text Theory (MTT), emphasizes more than Zwicky and Fraser the importance of distinguishing syntactic criteria from semantic and morphological ones, and argues that dependency syntax first and foremost should be controlled by syntactic criteria. MTT has three layers of dependency annotation: a morphological, a syntactic and a deep syntactic/shallow semantic. The theoretical framework Functional Generative Description (FGD) (Sgall et al., 1986) is also a multi-stratal theory with three layers, where the annotation for several phenomena, such as coordination and verb groups, on the
analytical (syntactic) layer in FGD follows semantic rather than syntactic criteria. This is an issue that will be discussed further in sections 6.1 and 6.2.

In order to find the head in a group of words having a relation to each other, Nikula (1986) suggests that it is important to find out whether the construction is *endocentric* or *exocentric*. Bloomfield (1933) defines an endocentric construction as one whose distribution is identical with one or more of its constituents. That is, one of the words can readily replace the whole construction without unpredictable changes in meaning. The prototypical endocentric case is the noun in a noun phrase, where adjectives, usually located to the left of nouns in languages such as English, can be present or not without affecting the grammaticality or the semantics in an unforeseeable way. For instance, an adjective such as *green* can readily be inserted between *a* and *cucumber*. An exocentric construction is the opposite of an endocentric construction, where no entity can replace the whole group of words. The mutual dependence between *was* (the predicate) and its subject is such a case, as both need each other to compose a grammatical sentence.

The distinction between syntactic and semantic criteria, and endocentricity and exocentricity, is relevant for the linguistic phenomena studied in the transformation track, and will also be further discussed in sections 6.1 and 6.2.

### 2.2.2 Dependency Trees

Dependency trees will here be formally defined using graph theory. Every sentence has been segmented into $n$ words or tokens, which are represented as $w_1, \ldots, w_n$. A dependency tree is *directed* and *labeled* with dependency types $r_0, r_1, \ldots, r_m \in R$, such as $\text{sbj}$, $\text{vmod}$ and $\text{nmod}$ for representing subject, verb modifiers and noun modifiers. The definition of a dependency tree is based on the definition of a *dependency graph*, which is formally a triple $G = (V, E, L)$, defined as in Nivre (2006):

**Definition 1** Given a set $R$ of dependency types, $G = (V, E, L)$ is a labeled dependency graph for a sentence $x = (w_1, \ldots, w_n)$, iff:

1. $V = Z_{n+1} = \{0, 1, 2, \ldots, n\}$
2. $E \subseteq V \times V$
3. $L : E \rightarrow R$

The tokens of a sentence are assigned an integer according to the order in which they appear in the sentence, which are the nodes of $V$ including a special root node $0$. The set $E$ contains the arcs for the graph connecting two nodes in $V$. Each arc in $E$ is assigned one dependency type (arc label) in $R$ using the function $L$. To simplify the notation, a directed arc $(i, j) \in E$ will be denoted $i \rightarrow j$, and the corresponding arc labeled $r$, i.e., $(i, j) \in E$ and $L(E) = r$, will be denoted $i \xrightarrow{r} j$. For instance, the arc from *was* to
She in figure 2.1 is represented by 2 \(\text{SBJ} \rightarrow 1\), or even more conveniently by \(\text{was SBJ she}\) when the words are unambiguous. Moreover, the notation for the reflexive and transitive closure of the arc relation \(E\) is \(i \rightarrow^* j\), that is, this relation holds if and only if there is a path of zero or more arcs from \(i\) to \(j\). Token \(i\) is then said to dominate \(j\). The relation \(i \leftrightarrow^* j\) is then the reflexive and transitive closure of the undirected arc relation.

Despite this fairly unrestricted definition, a number of dependency theories fail to comply with this notion of dependency graph. The dependency description of Tesnière (1959), for example, distinguishes between three different types of syntactic relations (connection, junction and transfer), where only connection corresponds to directed and lexical dependency. The definition also excludes any dependency relations that are not binary, something which can be useful in relations that involve more than two words. This holds in different ways for both coordination and verb groups. However, most such constructions tend to decompose quite naturally into binary relations, as noted by Mel’čuk (1988).

In this thesis, definition 2 of a dependency tree below holds for the data resources and parsers used in the coming experiments.

**Definition 2** A dependency graph \(G\) is a dependency tree if and only if:

1. **There should be one independent element:** There is no \(i \in V\), such that \(i \rightarrow 0\).

2. **The dependency graph must be connected:** For all nodes \(i, j \in V\), \(i \leftrightarrow^* j\).

3. **Every word should have at most one head:** For all nodes \(i, j, k \in V\), if \(i \rightarrow j\) and \(i \neq k\), then not \(k \rightarrow j\).

Constraint (1) states that there should be one node without a head, which is the special node with index 0. This special node is the root of the dependency tree. According to (2), there is an undirected path between any two nodes in any dependency graph, while (3) makes sure that no node can depend on more than one other node.

Definition 2 entails another important property of the type of dependency graphs that is allowed. A dependency graph is acyclic, that is, if \(i \rightarrow j\), then not \(j \rightarrow^* i\). Constraints (2) and (3) are not sufficient to induce acyclic graphs, but the independence of node 0, according to (1), breaks the possibility of forming cycles. In other words, these three constraints entail that the graph fulfills the requirements of a rooted tree in graph theoretic terms, and it will be referred to as a dependency tree. The constraints hold for figure 2.1 which consequently forms a dependency tree.

Some well-known theories based on the notion of dependency do not subscribe to definition 2. The EUROTRA system (Johnson et al., 1985) and Word Grammar (Hudson, 1984, 1990) allow dependents to have multiple heads, mainly for semantic reasons such as subject sharing in *John likes walking*. In addition, Word Grammar allows cycles, which also violates definition 2.
2.2 Dependency Structure

2.2.3 Projectivity

An interesting constraint that can be imposed on dependency structures is *projectivity*. The notion of projectivity was discussed first by Lecerf (1960) and later also by others, e.g., Hays (1964); Solomon (1965); Robinson (1970), and it has been defined in various ways that are not always equivalent. The definition adopted here is a reformulated version of the definition of Kahane et al. (1998), regarding projectivity as a property of individual arcs:

**Definition 3** A dependency arc \( i \to k \in E \) is **projective** if and only if for every token \( w_j \) occurring between \( w_i \) and \( w_k \) in the string (\( i < j < k \) or \( i > j > k \)), \( i \to * j \).

According to this definition, all arcs in the upper dependency tree of figure 2.2 are projective except *believe* → *what*, since there is at least one word located between *what* and *believe* that is not dominated by *believe*. In this case, there are in fact no words in between that are dominated by *believe*. It is the other way around for *does*, i.e., *does* → *believe*. It is noteworthy that this definition of projectivity implies that a non-projective tree contains crossing arcs, when arcs are drawn above the sentence, but does not imply that all arcs involved in some crossing are non-projective. The arcs <ROOT>→*does*, *does*→*So* and *does*→? are projective. For example, the only word located under *does*→*So* is *what*, and *does*→*what* (via *believe*).

In the lower dependency tree, there is also only one non-projective arc, i.e., *calls*→*from* (the word *yesterday* does not transitively depend on *calls*), whereas *received*→*yesterday*, on the other hand, is projective.

Definition 3 can then be used to define projectivity for an entire dependency tree:

**Definition 4** A dependency tree \( G = (V, E, L) \) is **projective** if and only if all arcs in \( E \) are projective.

This definition entails that the dependency tree in figure 2.1 is projective, whereas the dependency trees in figure 2.2 (as well as figure 1.1) are non-projective.

The corresponding notion in a constituency-based representation is a discontinuous construction. One of the often mentioned advantages of dependency structure is that non-projectivity can be explicitly and naturally described, compared to discontinuous constructions in constituency structure. Context-free grammar is unable to handle discontinuous constructions, which therefore require special treatment. However, there is a discrepancy between, dependency-based syntactic theories on the one hand, and practical parsing systems on the other. Any sensible dependency-based syntactic theory recognizes that projectivity (and continuous constructions) can cover the vast majority of dependency structures, but realizes that there is a need for non-projectivity in certain situations. Several practical parsing systems are, however, still projective for at least two reasons: efficiency and accuracy. A projective parser is often more time-efficient, and the errors on the
small number of non-projective constructions can often be neglected since accuracy for projective constructions in projective parsing is often higher than in non-projective parsing. This discrepancy is nevertheless unsatisfactory.

As noted by, for instance, Hall and Novák (2005), word order is another issue that is tied to the notion of projectivity, since languages with freer word order tend to have a higher proportion of non-projectivity than languages with more static word order. English is a language with relatively static word order, at least compared to for instance Slavic languages, for which dependency-based representations have been more influential. In Meaning-Text Theory (MTT) of Mel’čuk (1988), there is no order among the words in the syntactic layer, and any syntactic information encoded using the linear order of the words is instead captured by additional annotation associated with the tokens or dependency labels. On the other hand, such additional annotation is not used for FGD (Sgall et al., 1986), as it preserves the word order. It is worth noting that, at the tectogrammatical (shallow semantic) layer in FGD, all trees are projective, even though both FGD and MTT allow non-projectivity at the syntactic/analytical layer.

2.3 Data-Driven Dependency Parsing

This section brings the two previous sections together by discussing data-driven approaches for dependency parsing. A majority of these approaches are not utilizing a grammar at all, but it is worth mentioning that exceptions exist. One such example is Wang and Harper’s statistical parser (2004) evaluated on English (Penn Treebank), which uses a grammar but trains on empirical data. Another one is the system worked out by Foth et al. (2004), which combines WCDG with various statistical components. All other approaches discussed below in this section do not rely on any grammar
Data-driven dependency parsing, i.e., using syntactically annotated corpora, or treebanks, to construct robust broad-coverage parsers, has gained a great deal of attention in the last few years. As one consequence, the CoNLL shared tasks of 2006 and 2007 were about multilingual dependency parsing (Buchholz and Marsi, 2006; Nivre et al., 2007). The shared tasks have then further increased the interest. A range of different parsers producing dependency trees, according to the definition in section 2.2, have been proposed and implemented lately. McDonald and Nivre (2007) identify two categories of data-driven dependency parsers, graph-based and transition-based ones.

The best known graph-based dependency parsers build on the idea of searching for the highest scoring dependency tree on the basis of some scoring mechanism. As the parser output forms trees, this is equivalent to finding the highest scoring directed spanning tree in a weighted directed graph. Eisner (1996) was one of the first to propose an algorithm for graph-based probabilistic dependency parsing, producing projective and unlabeled dependency trees with the help of machine learning. This approach was further developed by McDonald et al. (2005), who used the Chu-Liu-Edmonds algorithm for non-projective dependency parsing, having $O(n^2)$ time complexity in relation to sentence length. Both the Eisner parsing algorithm and the Chu-Liu-Edmonds algorithm are implemented as (directed) maximum spanning tree algorithms, which require that scores, or weights, are assigned to the permissible arcs of a sentence to parse. Some graph-based dependency parsers with state-of-the-art performance are McDonald et al. (2006) and Nakagawa (2007) participating in the CoNLL shared tasks 2006 and 2007, respectively.

While graph-based dependency parsers try to produce a dependency tree by means of a global optimization strategy, dependency parsers using a transition-based approach, on the other hand, make locally optimal decisions without knowledge about the global dependency structure. The advantage is that more contextual information can be used in the local optimal decisions in comparison with graph-based dependency parsers without sacrificing efficient parsing to the same extent. While graph-based parsers conduct exhaustive (or nearly exhaustive) search, transition-based parsers often perform deterministic parsing, as they apply a classifier trained on a treebank to guide the parsing. The classifier learns to predict the next transition from one parser state to another by examining the richer contextual information. During training, it has access to the correct transition, which is extracted from the correct dependency tree in the training data.

Some examples of transition-based dependency parsers are Yamada and Matsumoto (2003) for English and Kudo and Matsumoto (2000) for Japanese, both having $O(n^2)$ time complexity. Another dependency parser with quadratic time complexity, which can be formalized as a transition-based dependency parser, is Covington’s algorithm (2001).

Another similar algorithm for dependency parsing was proposed by Nivre (2003) for unlabeled dependency structure, but with linear, $O(n)$, time com-
plexity. The approach was extended to labeled dependency structure in Nivre et al. (2004), but the algorithm is only able to produce projective dependency trees. Transition-based approaches also have the ability to perform at state-of-the-art level, as shown by two transition-based systems that participated in the CoNLL shared tasks 2006 and 2007, Nivre et al. (2006) and Hall et al. (2007), using the parser system MaltParser.

There are a number of approaches to avoid the completely greedy and locally optimal parsing decisions by exploring a larger region of the search space. Both Johansson and Nugues (2006) and Titov and Henderson (2007a) propose beam search, keeping several potential parse paths active during parsing, with a mechanism for selecting the best one in the end. Titov and Henderson, for instance, report improved accuracies for Dutch and Slovene from the data sets of the CoNLL shared tasks 2006 in comparison with MaltParser. Another recent beam-search system, combining transition-based parsing with graph-based features and global optimization, is Zhang and Clark (2008), reporting state-of-the-art results for the (English) Penn Treebank and Chinese Penn treebank.

Dependency parsers can also be categorized as either projective or non-projective, depending on whether they produce dependency trees conforming to the definition of projectivity in section 2.2 (or another equivalent definition). While unable to produce non-projective dependency trees, projective dependency parsers can be motivated practically. The parsing problem for a non-projective parser (normally) becomes more complex, which usually affects efficiency negatively. There probably does not exist an algorithm with linear time complexity producing non-projective dependency trees. The time complexities for different graph-based dependency parsers were for instance investigated by McDonald and Satta (2007).

This thesis uses three different transition-based parsing algorithms. The algorithm devised by Nivre (2003), called Nivre’s arc-eager algorithm, is the most frequently used and will therefore be described in more detail here. The parsing algorithm produces rooted, single-headed, and projective dependency graphs. These graphs are not guaranteed to be connected, but they are always acyclic. A rooted, single-headed and acyclic dependency graph can trivially be transformed into a tree by adding arcs from the special root 0 to all other roots. The algorithm is in many ways similar to the shift-reduce parser for context-free grammars (Aho et al., 1986), the most apparent difference being that terminals (not non-terminals) are pushed onto the stack. In a way similar to the shift-reduce parser, the construction of syntactic structure is created by a sequence of transitions.

Parser configurations (or parser states) are represented by triples \( \langle S, I, A \rangle \), where \( S \) is the stack and \( I \) the list of (remaining) input nodes. The (current) set of arcs for the dependency graph is represented by \( A \). The parser can be initialized to the configuration \( \langle [0], [1, \ldots, n], \emptyset \rangle \), where 0 is the special root node, and \( [1, \ldots, n] \) the nodes corresponding to input tokens. The parsing algorithm terminates whenever it reaches a configuration of the form \( \langle S, [], A \rangle \) (for any list \( S \) and set \( A \)), where \( [] \) represents the empty list.
Data-Driven Dependency Parsing

2.3 Transition Config

\(\langle S, I, A \rangle \)

\(\langle [0], [1, 2, 3, 4, 5], A_0 = \emptyset \rangle\)

\(\langle [0, 1], [2, 3, 4, 5], A_0 \rangle\)

\(\langle [0, 2], [3, 4, 5], A_1 \rangle\)

\(\langle [0, 2, 3], [4, 5], A_2 \rangle\)

\(\langle [0, 2], [4, 5], A_3 \rangle\)

\(\langle [0, 2], [5], A_4 \rangle\)

\(\langle [0, 2], [], A_5 \rangle\)

**Left-Arc** adds an arc \(j \rightarrow i\) from the next input node \(j\) (i.e., the first element in \(I\)) to the node \(i\) on top of the stack (i.e., the top-most element of \(S\)), and pops \(i\) from the stack.

**Right-Arc** adds an arc \(i \rightarrow j\) from the node on top of the stack \(i\) to the next input node \(j\), and pushes \(j\) onto the stack.

**Reduce** pops the stack.

**Shift** is applied pushing the next input node \((j)\) onto the stack.

Reduce can only be used when the top node of the stack has a head. Shift can only be used when the input list is not empty, which is always the case during parsing since an empty stack is the termination condition. Left-Arc is not permissible if \(i = 0\), and neither Left-Arc nor Right-Arc is possible to apply when the dependent already has a head. These additional constraints associated with each type of transition guarantee single-headed, acyclic and projective, but not necessarily connected, graphs.

In order to construct the dependency tree of figure 2.3, the sequence of transitions listed below the dependency tree is applied. The algorithm is guaranteed to terminate after at most \(2n\) transitions, given that the length of \(I\) is \(n\), cf. Nivre (2003). That is, the time complexity is linear in the length of the input, assuming that transitions take constant time.

The transition system above is nondeterministic since there is normally more than one potential transition for an arbitrary parser configuration. One exception is that only Shift is possible when the stack is empty. As
discussed above, a transition sequence creates a dependency structure during parsing, but it is also possible to create a transition sequence from a dependency tree. This is known as oracle parsing. That is, an oracle, knowing the well-formed and projective dependency tree, outputs a transition sequence that, in turn, will correctly reconstruct the same dependency tree.

One way of producing a set of transition sequences is to use a dependency treebank. The parser configurations and transition sequences generated from the treebank can be provided as training data to a classifier, which tries to predict the correct transitions – including dependency types from $R$ for LEFT-ARC and RIGHT-ARC – given parser configurations. In other words, the classifier approximates the oracle function:

$$ f: \text{CONFIG} \rightarrow (\{\text{LEFT-ARC, RIGHT-ARC}\} \times R) \cup \{\text{REDUCE, SHIFT}\} $$

where CONFIG is the set of all possible parser configurations. A parser configuration in CONFIG contains a vast amount of information located in $(S, I, A)$, i.e., in the stack, the list of remaining input nodes and the partially built dependency structure (including annotation associated with tokens corresponding to nodes, such as parts-of-speech, etc.). This is too much for any classifier to cope with in practice. It is therefore necessary to abstract it into a set of features. Figure 2.4 contains a visualization of potential features. Top and Next represent the top of the stack and the next input node, respectively. TH, TL, TR and NL are other nodes in the parser configuration: the head of the top, the leftmost and rightmost child of the top node, and the leftmost child of the next input node, respectively. A lookahead of three nodes in the input queue is represented by L1, L2 and L3. The subscripts L, P and D are values associated with the nodes, i.e., the lexeme (L), the part-of-speech (P) and the dependency label to its head (D).

The number and types of features can be adjusted in order to create a better approximation of $f$, which is dependent on the chosen classifier. A number of different classifiers are used in this thesis, described later in subsection 3.4.1. The set of feature values and the correct transition produced by the oracle are referred to as a training instance.

In other words, oracle parsing is a method for generating training data for a classifier in the form of feature vectors, representing the parser configura-
tion. A classifier, trained on these data, can then be used for constructing projective dependency trees of new sentences, by repeatedly consulting the classifier in each new parser configuration. However, the classifier can only be an approximation of the oracle, and can commit errors at any point. It means that the parser will then be located in an unknown parser configuration. This is why transition-based parsers are assumed to suffer from error propagation, which will be discussed in more detail in chapter 4.

Two other transition-based algorithms are used as well, denoted Nivre’s arc-standard algorithm (Nivre, 2008), and Covington’s non-projective algorithm. The arc-standard algorithm has a transition system which is similar to the arc-eager algorithm producing projective dependency trees. It has only three transitions, where REDUCE is removed and RIGHT-ARC has a slightly different definition. Both algorithms remove the next node $j$ from $I$, but while RIGHT-ARC for arc-eager pushes $j$ in $I$ onto $S$, arc-standard instead moves the top-most node $i$ on $S$ back to $I$ as the next node. Node $i$ consequently replaces $j$ in $I$. As noted by Nivre (2008), the arc-standard algorithm is even more similar to shift-reduce parsing for context-free grammar. They are both strict bottom-up parsers, while the arc-eager algorithm can be characterized as a hybrid between bottom-up and top-down. It is worth noting that the feature model is dependent on the parsing algorithm. For instance, the top-most node on the stack for arc-standard can (in contrast to arc-eager) never be a dependent, which entails that is does not make sense to use any feature value associated with $\text{TH}$, e.g., the lexical value of $\text{TH}$ or the dependency type of $\text{Top}$. The next node in the input queue for arc-standard can, on the other hand, have dependents to the right. That is, arc-standard can use features values of $\text{NR}$ (i.e., next right), which is a node that can never exist for the arc-eager algorithm in figure 2.4.

Covington (2001) proposed a number of incremental transition-based parsing algorithms for dependency structure, one of which is able to recover non-projectivity. The version of the algorithm used in this thesis is the one described by Nivre (2007). In contrast to the arc-eager algorithm, Covington’s non-projective algorithm uses one additional stack, known as the context stack. The algorithm essentially builds a dependency tree by considering every pair of nodes $i, j \in V$, where $i < j$. This entails a quadratic time complexity. As described by Nivre (2008), non-projectivity is enabled due to the context stack. Consequently, Covington’s non-projective algorithm can also use other types of features, where the most apparent addition consists of attributes associated with the tokens on the context stack.

## 2.4 Parsing Discontinuous Constructions

This section will bring up a number of studies that in one way or another relate to chapter 5 with the research question Q1 in the transformation track, i.e., dealing with non-projectivity. The discussion is limited to data-driven approaches, but includes both dependency-based parsing and constituency-
based parsing, with discontinuous constructions as the phenomenon corresponding to non-projectivity.

2.4.1 Dependency-Based Parsing

There are essentially two approaches for producing non-projective dependency structure. Parsers can either do it directly, or apply post-processing on the output of projective parsers.

A number of language-specific robust broad-coverage parsers producing non-projective dependency structure exist, such as Tapanainen and Järvinen’s (1997) the grammar-driven, or rule-based, parsers for English, Foth et al. (2004) for German, and Holan and Žabokrtský’s (2006) two parsers for Czech. One of the better known data-driven dependency parsers producing non-projective dependency trees directly, also mentioned in section 2.3, is McDonald et al. (2005). As it uses the Chu-Liu/Edmonds algorithm, only guaranteeing that the output is a tree, non-projective structure can be produced even if all dependency trees in the training data are projective. Covington’s transition-based parsing algorithm (2001), also mentioned in section 2.3 and used in Nivre (2007), can also produce non-projective dependency structure directly. Attardi (2006) also experimented with a transition-based parsing algorithm, extending the approach of Yamada and Matsumoto (2003) to non-projective structure.

A study focusing on non-projectivity is Hall and Novák’s corrective modeling (2005). The motivation is that the parsers devised by Collins et al. (1999) and Charniak (2000), adapted to Czech, are not able to create the non-projective arcs present in the treebank, which is unsatisfactory. They therefore aim to recover the non-projective arcs that are absent in the parser’s output. By training a maximum entropy classifier, which identifies erroneous arcs (specifically the arcs that should be non-projective), they can pinpoint wrongly placed arcs and move them in order to recover their original (non-projective) positions. They report an improved accuracy for non-projective arcs over 50%, presumably compared to close to 0% without it. Another study that also applies post-processing on the output of projective parsers is Nivre and Nilsson (2005), a process which is known as pseudo-projective parsing. This technique is treated in depth in chapter 5.

The approach of McDonald and Pereira’s (2006) approximative dependency parser is similar to the corrective modeling of Hall and Novák in the sense that it conducts postprocessing on the output of a projective parser (the Eisner algorithm). They investigate all arcs and determine, using a model of the non-projective training data, whether the overall score of the whole parse tree can be improved by replacing arcs of the projective output with new arcs. In contrast to Hall and Novák, the new arcs have to be non-projective, since the Eisner algorithm finds the highest scoring projective dependency tree. The tree that increases the overall score the most is then selected.
2.4.2 Constituency-Based Parsing

The parsers constructed by Collins (1997; 1999) and Charniak (2000), mentioned in section 2.1, are most famous for parsing the Wall Street Journal part of the Penn Treebank. However, Collins and Charniak’s parsers are only able to produce trees that can be generated by a context-free grammar, even though the syntactic annotation goes beyond ordinary context-free structure in a number of ways. The Penn Treebank contains so-called empty categories and co-indexation dealing with non-local dependencies, closely related to the notion of non-projectivity for dependency structure, which these parsers normally disregard. Much research has been carried out to handle empty categories, discontinuous constructions and non-local dependencies in constituency-based parsing. Various successfully applied tree transformations dealing with discontinuous constructions are discussed below.

Using the same annotation strategy as the Penn Treebank, figure 2.2 (page 14) would contain an empty NP node located to the right of believe. The empty NP node would be located inside a VP node together with believe, where the empty NP node has nothing but a pointer to the real object what. This indicates that the normal location for an object is to the right of the main verb, which in terms of dependencies means that the main verb is the head of the empty NP node. In turn, the pointer from the empty NP node to what entails the non-projective arc from the main verb believe to what, shown in the picture.

The above example of so-called wh-movement is just one type of usage of empty categories in the Penn Treebank. The construction for dealing with passive verbs is another. Plain unparsed text obviously does not contain empty categories, and even though Collins’ Model 3 (1999) tried to resolve non-local relative pronoun dependencies using trace threading as in GPSG (Gazdar et al., 1985), mainstream statistical parsers, such as that of Charniak (2000), are in general not able to deal with these constructions.

Several studies have been performed to recover empty categories. Besides the in-processing of some non-local relations in Collins’ Model 3, two other approaches to the problem can be identified. Johnson (2002) implemented an approach based on post-processing using a pattern-matching algorithm. It can be characterized as a memory-based learning procedure on constituent-based subtrees on the parser output, where the patterns are extracted from the training data. He concluded that a large proportion of the empty categories can be recovered. Dienes and Dubey (2003a; 2003b) take on the issue from the other angle by preprocessing unparsed sentences. They train a Hidden Markov Model (HMM) in order to identify empty categories before parsing. Postprocessing was performed on the parser output to attach each empty node to an antecedent. They report better results than Collins and Johnson.

The experiments by Levy and Manning (2004) on deep dependencies extend the approach to German (the NEGRA treebank). They present
results that compare favorably with the other studies for English on the Penn Treebank. Other contemporary similar studies, such as Cahill et al. (2004), Jijkoun and de Rijke (2004) and Campbell (2004), have developed the approaches further. Jijkoun and de Rijke have an approach facilitating preprocessing and postprocessing. For instance, they use dependency relations in order to recover non-local dependencies in the Penn Treebank, while treating the parser as a black box.

The Penn Treebank also contains grammatical functions that for instance are used for marking subjects and predicates. Among others, Gabbard et al. (2006) parse the Penn Treebank with all the available information, including grammatical functions and discontinuity.

Another case in point is Schmid (2006), who performs real slash style parsing on the Penn Treebank, annotated with GPSG-style features such a link traces and fillers. He employs an unlexicalized PCFG parser, and reports generally improved parsing accuracy, as well as the best published results for the empty category prediction task and the trace-filler coindexation.

2.5 Tree Transformations

Like the previous section, this section discusses work related to the transformation track, but for chapter 6 with focus on research question Q2. This section is also, like the previous section, limited to data-driven approaches and brings up both dependency-based and constituency-based parsing approaches.

2.5.1 Dependency-Based Parsing

The literature contains far fewer studies in dependency-based parsing compared to constituency-based parsing utilizing tree transformations. There are at least two possible explanations. The first reason is that dependency-based parsing has been studied less compared to constituency-based parsing. The second is that dependency structure usually results in less complex tree structures, mainly due to the absence of non-terminals, which leaves less room for applying transformations. For instance, the very important parent annotation in PCFG, which distinguishes NPs in subject position (with S as parent) from NPs in object position (with VP as parent), has no correspondence in dependency structure. First, dependency structures normally have direct relations from verbs to subjects and objects. This also makes the search for these types of relations trivial, and less complicated than in constituent structure. Secondly, these relations are usually explicitly marked in the dependency structure through labels such as Sbj and Obj.

Several preprocessing and postprocessing transformations in data-driven dependency parsing are actually caused by the fact that constituency-based parsers have been modified in order to output dependency structure. Collins’
parser was adapted to parse Czech, which required that the dependency
trees in the training data were converted to phrase structure (Collins et al.,
1999). The internally used bilexical relations were then considered the pri-
mary output of the parser, not the phrase structure trees. Besides the
important preprocessing step of converting from dependency structure to
phrase structure, which is their base representation, they apply a number of
linguistically motivated phrase tree transformations, discussed in the next
subsection.

Even though tree transformations are not applied, a relevant study for
the coordination and verb group transformations in chapter 6 is Riedel and
Clarke’s Integer Linear Programming parser (ILP-parser) (2006). The max-
imum spanning tree algorithm, discussed in section 2.3, can be reformulated
as an integer linear programming problem. The benefit is that additional
constraints – not just those that guarantee that the parser produces a rooted
tree – can be imposed. As an example, a rule like “In a symmetric coordi-
nation there is exactly one argument to the right of the conjunction and at
least one argument to the left” can be formulated as constraints that the
parser must obey. Even though they discuss a number of problems, such
as slower parsing time, they report improved results with such additional
constraints. It is also worth noting the approach adopted by Zeman (2004)
in order to improve accuracy for coordination for Czech. He uses a kind of
pattern matching, based on frequencies of the parts-of-speech of conjuncts
and conjunctions, which seems to improve the overall performance of the
parser.

2.5.2 Constituency-Based Parsing

It has been shown for constituency-based data-driven approaches that it
is important that the base representation is well-chosen. Applying various
transformations before and after parsing, using constituency-based repre-
sentations, has been an important research topic for several years. This
does not only hold for the recovery of constructions that are impossible
for existing practical parsing systems, as described above, but also for im-
proving the accuracy for constructions that the parsers are able to produce,
although less successfully. One such example is PCFG, which was discussed
in section 2.1.

Although the simplicity of PCFG is attractive, parsing based on (unlexi-
calized) PCFG tends to perform relatively poorly when the probabilities are
induced directly from a corpus. Such data-driven PCFG parsing is heavily
dependent on the base representation of the grammar (e.g. Johnson 1998,
Klein and Manning 2003), but several of its shortcomings can be compen-
sated by applying tree-transformations on the parse trees as preprocessing
and postprocessing.

One of the weaknesses of PCFG is the lack of non-local relationships be-
tween the non-terminal nodes, but Johnson (1998) reports a significantly
increased accuracy by simply augmenting the node names with the par-
ent’s node name as preprocessing for the Penn Treebank. The implicit independence assumption of context-free grammar is therefore weakened by incorporating this kind of contextual information, which is presumably important to accurately resolve a number of linguistic phenomena. For instance, knowing that the parent of an NP is S (subject position) and not VP (object position) considerably increases the probability that the NP will expand to a pronoun.

Johnson also elaborates with different constructions to represent PPs in a VP by applying tree transformations, such as flattening or deepening the structure of the training data, with a subsequent inverse transformation on the parser output. These tree transformations have an impact on the performance, although less significant than the parent annotation. He also points out that “the choice of tree representation can have a noticeable effect on the performance of a PCFG language model”, and notes that a tree representation that is well-chosen on linguistic grounds can be quite different from what results in good performance for the PCFG model. Later studies have extended the methods for preprocessing and postprocessing, including various tree transformations, by means of PCFG models. Dienes and Dubey (2003c), for example, perform related transformation experiments on German. Klein and Manning (2003), who close the gap to lexicalized parsing models, iteratively combine linguistically motivated transformations.

Later studies have refined the method even further, for example Petrov and Klein (2007). One of the reasons why PCFG parsing performs relatively badly for Penn Treebank is the too coarse-grained set of non-terminal names. They have, among other things, automatized the process of making the set of non-terminals more fine-grained by iteratively splitting them more and more while recording how it affects parsing accuracy, with a back-off mechanism to undo bad splits. They even report that these automatically induced splits can be linguistically motivated.

Establishing the right base representation, which can be created through various corpus transformations, was also an important part of the development of the constituency-based parsers constructed by Collins (1997; 1999) and Charniak (2000), mentioned in section 2.1. They have for a long time performed considerably better than approaches based on PCFG. The numerous transformations taking place for the Markov grammar in Collins’ parsers for the Penn Treebank are however not described in detail in his thesis (Collins, 1999), but Bikel (2004) enumerates eleven applied transformations in the parser, including the special treatment of the bilexical relations in coordination. Other transformations target e.g., relative clauses and punctuation, which leads to improved parsing accuracy.

Without exaggerating, one can conclude that various tree transformations are of great importance in most constituency-based state-of-the-art parsers. This section has also shown that coordination is problematic for both dependency-based and constituency-based data-driven parsing. By applying transformations in constituency-based parsing, and by other means in dependency-based data-driven parsing, the accuracy can be improved. Ap-
plying transformations for dependency-based data-driven parsing is therefore one of the issues addressed in the coming transformation track.

### 2.6 Combining Dependency Parsers

While the two previous sections dealt with related work for the transformation track, this final section in the background chapter brings up related work relevant for the combination track in chapters 7–9. It discusses various ways of combining dependency parsers, but combination has also been performed for constituency-based parsers (e.g., Henderson and Brill 1999).

One approach, which is not restricted to just transition-based parsing, is to combine the output of several dependency parsers. It is a natural next step for increasing accuracy further for existing dependency parsers. Zeman (2004) proposed a language-independent ensemble approach, which for each token greedily chooses a head token based on the head tokens of all single parsers. They present improved results in comparison to the best single parser for Czech, but – unless explicitly forbidden – their approach can cause cycles in the combined dependency graph even though all single parsers are acyclic.

By contrast, the ensemble approach proposed by Sagae and Lavie (2006), who also report significantly increased accuracies, is based on finding the maximum directed spanning tree given a dense weighted graph. The output is guaranteed to be acyclic even if some single parsers produce cycles. The arc weights of the dense graph are determined on the basis of the m output dependency graphs according to the following description.

Given the output dependency graphs $G_i$ (1 ≤ i ≤ m) of m different parsers for an input sentence $x$, a new graph is constructed containing all the labeled dependency arcs proposed by some parser, with each arc $a$ being weighted by a score $s(a)$ reflecting its popularity among the $m$ parsers. The output of the ensemble system for $x$ is the maximum spanning tree of this graph (rooted at the node 0), which can be extracted using the Chu-Liu-Edmonds algorithm, as shown by McDonald et al. (2005). For the best performing combination strategy, Sagae and Lavie (2006) make $s(a) = \sum_{i=1}^{m} w_{ia}^{c}$, where $w_{ia}^{c}$ is the average labeled attachment score of parser $i$ for the word class $c$ of the dependent of $a$, and $a_i$ is 1 if $a \in G_i$ and 0 otherwise.

Parser combination is also possible in a completely different way by making the input of one parser into features of another, which can be characterized as parser stacking. This was done by Nivre and McDonald (2008), where the two best systems of the CoNLL shared task 2006 (Buchholz and Marsi, 2006), MSTParser and MaltParser, were combined in this fashion. They reported a higher improvement on the CoNLL shared task 2006 data when the output of MaltParser is used as input to MSTParser compared to vice versa. Parser stacking was further investigated by Martins et al. (2008), who also report improved performance over existing state-of-the-art
dependency parsers on the CoNLL shared task 2006 data. Zhang and Clark (2008) is another recent study, which was also mentioned in section 2.3, proposing a beam-search-based parser that combines both graph-based and transition-based parsing into a single system.

In other words, parser combination has in different ways been applied successfully, as shown in these studies. Parser combination will therefore be studied in the combination track, especially the approach presented by Sagae and Lavie (2006).
Chapter 3

Methodological Preliminaries

This chapter settles a few methodological preliminaries that are of importance throughout the rest of this thesis. It starts by describing the evaluation metrics and statistical significance test in sections 3.1 and 3.2. The data sets are then presented in section 3.3, followed by a description in section 3.4 of the primary software systems used.

3.1 Evaluation Metrics

The task of a dependency parser is to assign a syntactic head and a dependency label to every token in the input sentence. The main evaluation metric used for dependency parsers is therefore the proportion of correctly parsed tokens in the test data. The correctness of a parsed token is based on whether the token has been assigned the correct head, with or without the correct dependency type, compared to the correct tree in the gold-standard test data.

- Labeled attachment score ($AS_L$):
  \[
  \frac{\text{# of tokens with correct heads and dependency types}}{\text{total # of tokens}}
  \]

- Unlabeled attachment score ($AS_U$):
  \[
  \frac{\text{# of tokens with correct heads}}{\text{total # of tokens}}
  \]

$AS_L$ is normally considered to have the highest priority, for instance by Buchholz and Marsi (2006) and Nivre et al. (2007), but $AS_U$ will also be used in case it is of interest for the experiment.

When investigating the accuracy for certain classes of tokens in the parser output, accuracy must be divided into precision and recall.

- Precision (P) for tokens classified as X by the parser is:
  \[
  \frac{\text{# correctly parsed tokens classified as X by the parser}}{\text{total # of tokens classified as X by the parser}}
  \]
• Recall (R) for tokens classified as X in the gold-standard is:

\[
\frac{\text{# correctly parsed tokens of classified as X by the gold-standard}}{\text{total # of tokens classified as X by the gold-standard}}
\]

Correctly parsed can again mean that the token has the correct head, correct label, or both. This division is for instance necessary when investigating the accuracy on non-projective tokens (or arcs), where the set of non-projective arcs in the gold-standard is not the same as the set of non-projective arcs in the parser output.\(^1\)

When applying, e.g., a modification that improves accuracy, it can be of interest to measure how much the errors have decreased in relation to the total number of errors. Error reduction is defined as:

\[
\frac{p' - p}{1.0 - p}
\]

where \(p\) is any accuracy metric value – attachment score, precision or recall – for a parser, and \(p'\) the accuracy metric value for the modified (improved) parser, parsing the same data. For instance, if \(p = 0.8\) and \(p' = 0.85\), then the error reduction is 0.25. When convenient, we will express the error reduction and the other metrics as percentage instead, for example, 25% in this case. Error reduction can be used to facilitate comparison when evaluating a modification using for instance different treebanks.

### 3.2 Statistical Significance

In order to know how much importance we should attach to an observed difference in accuracy, it is useful to test whether the difference is statistically significant. In this thesis, McNemar’s test (Bland, 2000) will be used to assess whether a difference in accuracy between the outputs of two different parsers on the same data set is statistically significant. It is applicable when the observations are matched pairs with exactly one of two outcomes. The evaluation measurements labeled and unlabeled attachment score, precision and recall are all well-suited for McNemar’s test. Each observation, i.e., either a token or a sentence, has exactly one of two outcomes, since each observation can be either correct (+) or incorrect (−) for each parser. A 2x2 classification table can thus be created

<table>
<thead>
<tr>
<th></th>
<th>Parser A</th>
<th></th>
<th>Parser B</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td>+</td>
<td>a</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

\(^1\)Note that the terms precision and recall will be used even though they are not completely correct. A more correct use of the terms precision and recall for instance for projectivity is whether the arcs are classified as either projective or non-projective, not whether the arcs have the correct head and/or label.
where \( a + b + c + d \) equals the number of observations. McNemar’s test ignores the pairs with identical labels. Hence, only \( b \) and \( c \) count. The null hypothesis is that the \( b \) pairs are as likely as the \( c \) pairs. That is, if \( p_1 = P(\text{correct(}\text{ParserA}\mid E) \) and \( p_2 = P(\text{correct(}\text{ParserB}\mid E) \) where \( E \) is the situation that Parser A and Parser B disagree, then \( p_1 = p_2 = 0.5 \). The binomial distribution can then be used for computing the two-tailed \( p \)-value in order to determine whether there is a statistically significant difference between Parser A and Parser B.

Moreover, if \( b + c > 25 \) (which they normally are given the large test sets used in this thesis), then the binomial distribution approximates a standard normal distribution, where

\[
Z = \frac{|b - c| - 1}{\sqrt{b + c}}
\]

This way of computing McNemar’s test is used throughout the thesis. The significance levels \( p < 0.05 \) and \( p < 0.01 \) are applied when appropriate. For the 0.05 significance level, the two tails are located at \( |Z| > 1.9600 \), and for 0.01 at \( |Z| > 2.5758 \).

### 3.3 Data Sets

The data sets used in this thesis are five data sets from the CoNLL shared tasks 2006 (Buchholz and Marsi, 2006) and all ten data sets from the CoNLL shared task 2007 (Nivre et al., 2007). These data sets are introduced below, making a total of 15 data sets representing 13 languages.

Each data set (or treebank) has one dedicated set of dependency trees for training set and another set for testing. The training set is used for model selection in order to optimize accuracy, and the test set for final evaluation (or model assessment). Table 3.1 gives an overview of the treebank properties, which, among other things, shows that the largest training set is the Czech data set from the CoNLL shared tasks 2006, containing approximately 73 thousand sentences comprising 1.3 million words. For the data sets of the shared tasks, it is worth noting that the number of tokens in each test data set is about 5 000 tokens, but the sizes of the training sets vary greatly.

Over-fitting the parsers to the test data should be avoided, in order to get reliable results. This can happen when a parser is iteratively evaluated on the test data during the development phase, for example, when performing feature optimization. The approach in this thesis is thus to evaluate the parsers on data sets distinct from the test data during the development phases. However, for the data sets of the two shared tasks, no development test sets were provided. To facilitate testing during the development phase, the training set of each treebank has been divided into ten equally large sets in a pseudo-randomized way. Every sentence \( i \) with the same value for \( i \) mod 10 will belong to the same set. For instance, sentences 1, 11, 21, . . .
### Training data

<table>
<thead>
<tr>
<th></th>
<th>#T</th>
<th>#S</th>
<th>T/S</th>
<th>LF</th>
<th>#D</th>
<th>%N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arabic</td>
<td>54</td>
<td>1.5</td>
<td>37.2</td>
<td>Y</td>
<td>27</td>
<td>11.2</td>
</tr>
<tr>
<td>Czech</td>
<td>1249</td>
<td>72.7</td>
<td>17.2</td>
<td>Y</td>
<td>78</td>
<td>23.2</td>
</tr>
<tr>
<td>Dutch</td>
<td>195</td>
<td>13.3</td>
<td>14.6</td>
<td>Y</td>
<td>26</td>
<td>36.4</td>
</tr>
<tr>
<td>German</td>
<td>700</td>
<td>39.2</td>
<td>17.8</td>
<td>N</td>
<td>46</td>
<td>27.8</td>
</tr>
<tr>
<td>Slovene</td>
<td>29</td>
<td>1.5</td>
<td>18.7</td>
<td>Y</td>
<td>25</td>
<td>22.2</td>
</tr>
</tbody>
</table>

### Test Data

<table>
<thead>
<tr>
<th></th>
<th>#T</th>
<th>#S</th>
<th>T/S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arabic</td>
<td>5373</td>
<td>146</td>
<td>36.8</td>
</tr>
<tr>
<td>Czech</td>
<td>5853</td>
<td>365</td>
<td>16.0</td>
</tr>
<tr>
<td>Dutch</td>
<td>5585</td>
<td>386</td>
<td>14.5</td>
</tr>
<tr>
<td>German</td>
<td>5694</td>
<td>357</td>
<td>15.9</td>
</tr>
<tr>
<td>Slovene</td>
<td>6390</td>
<td>402</td>
<td>15.9</td>
</tr>
</tbody>
</table>

**Table 3.1:** Information about the data sets used in this thesis. #T * 1000 = Number of tokens; #S * 1000 = number of sentences; T/S = mean sentence length; LF = lemma and syntactic and/or morphological features is available (Y/N); #D = number of dependency types; %N = percentage of sentences with non-projective dependency tree.

constitute the set 1, and 10, 20, 30, \ldots set 0. One of these sets can constitute the development test set, when using the other nine sets as training data, but exactly how these sets have been used for training and testing, respectively, will be described in connection with each experiment.

The CoNLL format, the data format of both shared tasks, is a column-based one with six input columns, where each row represents a token (normally the same as a word) and sentences are separated by a blank line. Each row has token counter (ID), word form (FORM), lemma or stem of word form (LEMMA), coarse-grained part-of-speech tag (CPOSTAG), fine-grained part-of-speech tag (POSTAG) and unordered set of syntactic and/or morphological features (FEATS), and four output columns: head of the current token (HEAD), dependency relation to the head (DEPREL), projective head (PHEAD) and projective dependency relation (PDEPREL). The last two columns were never used in the shared tasks and will not be used here either. Furthermore, some treebanks have dummy values for LEMMA and FEATS, if the original treebank did not have this information. These treebanks have an N in the LF column.

The transformation track utilizes five of the 2006 treebanks and the combination track all ten treebanks of the 2007 data in table 3.1. The main reason why the two tracks use different data sets is that the experiments of the transformation track were performed during or right after the development of the MaltParser system for the shared task 2006 (Nivre et al., 2006). Most of the work and experiments reported in the combination track were, on the other hand, performed as part of the development of MaltParser for...
the shared task 2007 (Hall et al., 2007).

The presentation of the 2006 treebanks below is more detailed than for the CoNLL 2007 treebanks. The reason for this is to facilitate analysis, since the CoNLL 2006 treebanks are studied and parsed individually in the transformation track. The experiments in the combination track are on the other hand mostly presented as averaged over all treebanks.

### 3.3.1 CoNLL 2006 Data

The CoNLL shared task in 2006 was devoted to multilingual dependency parsing. Five treebanks representing five different languages (Arabic, Czech, Dutch, German and Slovene), taken from the shared task, are in focus in the transformation track in chapters 5 and 6. The treebanks have been selected since they have annotation properties suitable for the transformations presented in these chapters. They are all, or have been converted to, dependency treebanks.

As other Slavic languages, Czech is characterized by rich inflection and relatively free word order. The Prague Dependency Treebank (PDT) contains newspaper text and is the largest manually annotated treebank (Hajič 1998, Böhmová et al. 2003). The CoNLL shared task 2006 version of PDT also contains almost 1.3 million words of training data. The annotation is based on the linguistic theories of the Prague school (Functional Generative Description). Only the analytical layer (Sgall et al., 1986) has been used in the CoNLL version.

Slovene is like Czech a Slavic language, with similar properties in terms of rich morphology and free word order. The creation of the Slovene Dependency Treebank (SDT) (Džeroski et al., 2006) has been highly influenced by PDT. Therefore, SDT uses an annotation that is very similar to PDT, including the set of syntactic tags labeling the dependency relations. SDT contains 2,000 sentences from the first part of the Slovene translation of Orwell’s 1984, of which 1,500 are extracted as training data in the conversion to the CoNLL format. Compared to PDT, it is a very small treebank, and the authors characterize the treebank as a proto-released treebank, meaning that the treebank is still under development.

As the name implies, the Prague Arabic Dependency Treebank (PADT) is also an offspring of PDT (Hajič et al., 2004), and is also influenced by the Prague school tradition. PADT contains newswire text of Modern Standard Arabic with both morphological and analytical annotation. It has the same number of sentences as SDT although with almost twice as many tokens, but is still small compared to PDT. The sentence length of PADT is noteworthy, being more than two times longer than any other of the four treebanks of the shared task 2006, as shown in table 3.1.

The CoNLL version of the Dutch Alpino Treebank (van der Beek et al., 2002) contains over 13,000 sentences. The original Alpino Treebank uses a mixture of the annotation schemes and guidelines applied to the CGN Corpus of spoken Dutch and the TIGER Treebank (Kakkonen, 2005).
The German TIGER Treebank (Brants et al., 2002) has emerged from the NEGRA Corpus project (Skut et al., 1998) and its annotation combines both phrase structure and dependency structure. It is a collection of German newspaper text on various topics. It is a relatively large dependency treebank, the second largest in the CoNLL shared task 2006 with almost 40,000 sentences.

3.3.2 CoNLL 2007 Data
The CoNLL shared task in 2007 (Nivre et al., 2007), which was also devoted to dependency parsing, was divided into two tracks: a multilingual track and a domain adaptation track. In this thesis, only the data for the ten languages in the multilingual track have been used. The characteristics of the data sets for the 10 languages are shown in table 3.1 as well.

As mentioned, the data sets were also formatted according to the CoNLL dependency data format. The ten languages are Arabic (Hajič et al., 2004), Basque (Aduriz et al., 2003), Catalan (Martí et al., 2007), Chinese (Chen et al., 2003), Czech (Böhmová et al., 2003), English (Marcus et al., 1993; Johansson and Nugues, 2007), Greek (Prokopidis et al., 2005), Hungarian (Csendes et al., 2005), Italian (Montemagni et al., 2003), and Turkish (Oflazer et al., 2003). The organizers motivated the selection of languages by the wish to have a typologically diverse set of languages.

PADT and PDT were used in both shared tasks. But whereas PADT has become about twice as large (since more sentences have been annotated since the first shared task 2006), PDT has been reduced to about 1/3. The table shows that the average sentence length varies a great deal, with Arabic again having the longest sentences, and Chinese the shortest. The amount of non-projectivity also varies greatly.

3.4 Software
This section gives an overview of the primary software systems used in this thesis, one dependency parser system, MaltParser, and one evaluation system for dependency structure, MaltEval.

3.4.1 MaltParser
The MaltParser system is a language-independent system with several transition-based parsing algorithms that derive dependency trees. It contains implementations of all transition-based algorithms used in this thesis, Nivre’s arc-eager and arc-standard algorithms and Covington’s non-

---

2 The organizers of the shared task 2007 decided to reduce the amount of training data for Czech, because several participants of the shared task 2006 had problems training their systems on so much data.

3 MaltParser is an open-source software, which can be downloaded from http://www.maltparser.org.
projective algorithm, which were described in section 2.3. The dependency structures produced comply with definition 2 in chapter 2.

All algorithms can be executed in two modes: learning mode and parsing mode. The dependency trees are reconstructed by an oracle function with access to the syntactically annotated sentences during learning. The oracle function derives the correct sequences of transitions by reconstructing the dependency trees. These transitions, together with the feature vectors representing parser configurations, are used as training data for the classifier to approximate the oracle function by means of inductive learning. During parsing, the dependency trees are created by predicting sequences of transitions by means of the induced classifier.

MaltParser has support for varying the feature model with the help of specification files, where individual features (e.g., word forms, part-of-speech and dependency labels) can easily be added or removed. As discussed in section 2.3, the feature model is dependent on the parsing algorithm. For instance, features can be extracted from the context stack for Covington's non-projective algorithm, but such features do not make sense for the other two.

MaltParser supports several supervised machine learning algorithms for inducing classifiers. Three different types of classifiers occur in the experiments of this thesis:

- Support Vector Machines (SVM) (Vapnik, 1979)
- Maximum Entropy Models (MaxEnt) (Berger et al., 1996)
- Memory-Based Learning (MBL) (Mitchell, 1997)

The classifiers are used to map each parser configuration to one single transition, even though all three could generate n-best lists. The parsing approach of MaltParser in this thesis is, however, deterministic in the sense that the parser never deals with more than one parser configuration at a time, and we are consequently only interested in the single locally optimal transition.

The idea behind SVM is that two classes, the positive and negative instances in the training data, can be linearly separated by a hyper-plane. The learning problem is to find the hyper-plane with the largest margin. This is implemented in the LIBSVM package (Chang and Lin, 2001), which, among other things, also contains functionality for separating the classes by means of a quadratic kernel. LIBSVM also contains functionality for the one-versus-one strategy for multi-class classification, which is necessary since the number of transitions (including the label) is always more than two. Symbolic features are converted to numerical ones using the standard technique of binarization in MaltParser (Hall and Nivre, 2008).

A maximum entropy classifier has similarities to SVM without a kernel. It is a conditional model that tries to maximize the conditional likelihood of the training data. One of the most apparent differences is that maximum entropy classifiers assign a probability to each possible transition. In this
thesis, only the transition with the highest probability is used. The software package MALLET (McCallum, 2002), which implements a maximum entropy model, has been integrated into MaltParser.

The third classifier, MBL, is in contrast to the two others known as a lazy learning method. Learning is essentially equivalent to storing all training instances in memory. Classifying new instances is done by finding the class of the most similar instance(s) in memory, by applying a similarity metric. While MaxEnt, known as an eager learning method, generalizes from the training data when inducing a model, MBL never abstracts away from (or eliminates) low frequency events. TiMBL (Daelemans and van den Bosch, 2005), a software package implementing MBL, has also been integrated into MaltParser.

For most of the larger treebanks, the training time is too long to be acceptable for SVM (e.g., more than 12 hours on an ordinary computer) during the development phases of the experiments. The time can be reduced considerably by splitting the training data into smaller parts and training separate classifiers on each part. The approach taken here is to determine which part each training instance will belong to based on the value of either the fine-grained (POSTAG) or coarse-grained (CPOSTAG) part-of-speech of the first input token (Hall et al., 2006). During parsing the appropriate classifier is then consulted by examining the value of the first input token, often with only a small decrease in accuracy. Splitting has only been done for SVM in this thesis, but splitting the training data is in principle applicable to any type of classifier in MaltParser.

3.4.2 MaltEval

MaltEval\(^4\) (Nilsson and Nivre, 2008b) is a software tool written in Java that provides quantitative and qualitative evaluation. It is to a large extent adapted to the evaluation script eval.pl, the evaluation script provided by the organizers of the CoNLL shared task 2006 (Buchholz and Marsi, 2006). The functionality of MaltEval is essentially a superset of this script (when used in the most verbose mode). It is also more flexible and contains many additional features. In contrast to eval.pl, MaltEval is able to visualize dependency structures and provides visual support for qualitative evaluation by highlighting errors. All evaluation tests mentioned in this chapter are implemented in MaltEval, which is the primary evaluation tool in this thesis. Also, all dependency trees shown in this thesis have been created using the visualization functionality in MaltEval.

\(^4\)Can be downloaded from: http://w3.msi.vxu.se/users/jni/malteval/.
Chapter 4

Detecting Error Propagation

This chapter is about error propagation. As noted earlier, multilingual data-driven dependency parsing has gained a lot of interest in the past years, and the CoNLL shared tasks of 2006 (Buchholz and Marsi, 2006) and 2007 (Nivre et al., 2007) have increased the interest even further. One observation from 2006 is that the two top-scoring systems – with virtually the same overall accuracy – were based on completely different parsing strategies. For these two parsing systems, MSTParser (version 0.2) and MaltParser (version 0.4), a detailed comparison was conducted by McDonald and Nivre (2007). One of their assumptions is that the deterministic transition-based parsing strategy of MaltParser leads to error propagation, while the graph-based parsing strategy of MSTParser does not. However, none of their evaluation metrics do in fact directly measure error propagation.

In this chapter, we will propose two new metrics designed to detect and measure the effect of error propagation and apply them to four different systems from the CoNLL shared task 2006. Although we will not be able to prove that the metrics actually capture error propagation as such, we will argue that they can be useful as diagnostics and show that they give the expected results for the four systems examined. We begin by describing the four parser systems in section 4.1, and go on to define and apply the new metrics in section 4.2. We conclude in section 4.3.

4.1 Four Dependency Parsers

As mentioned in chapter 2, MaltParser uses a transition-based parsing algorithm similar to shift-reduce parsing, where a parse is reduced to a sequence of transitions. The task is to learn this sequence, e.g., with a discriminative classifier using features based on the history of previous transitions. During parsing, the learned parsing model makes \textit{local} and \textit{greedy} transition decisions based on an informative history, and can commit errors at any point of the transition sequence. The parser is then located in a parser configuration with an incorrect history that was not seen during learning, which is presumed to increase the error probability for subsequent transition predictions.

One method that has been proposed to alleviate the presumed error propagation in transition-based dependency parsing is to relax its strictly greedy
Detecting Error Propagation

search strategy. Johansson and Nugues (2006), another participating team in the CoNLL shared task 2006, use a transition-based parser with beam search. That is, there is more than one partial parse tree under consideration during parsing. The partial parse trees are ranked according to some scoring function, making it possible to prune away lower scoring partial parse trees, in order to maintain tractable parsing time. While this beam search parser was not as accurate as, for instance, MaltParser and MSTParser, Johansson and Nugues report improved figures compared to their baseline parser with no beam. It is commonly assumed that beam search can counteract error propagation in transition-based parsing. But again, no evaluation metrics have been used that directly support this claim. Johansson and Nugues’ parser will be denoted LTH-parser from now on.

MSTParser (McDonald et al., 2006) is on the other hand characterized by its global and exhaustive parsing strategy. It is global since the accuracy of the entire graph is considered during learning for assessing the weights of the model parameters. It is essentially exhaustive since the optimal (projective) dependency tree of each sentence is found.\footnote{This is the approximate second-order spanning tree parser described in section 2.3.} To make the search tractable, less contextual information must be used compared to transition-based parsing algorithms. In this non-projective version of MSTParser, the context was restricted to one single sibling. This contextual information makes exhaustive search NP-hard, and this version of MSTParser is therefore strictly speaking exhaustive only with respect to the derivation of the optimal projective approximation. Nevertheless, in this graph-based parsing strategy, error propagation can be assumed to be less severe compared to MaltParser.

Another team that also participated in the CoNLL shared task 2006 was Riedel et al. (2006). Their approach is in a number of ways very similar to the parsing approach represented by McDonald et al. (2005). Their parser, which we denote ILP-parser, is more closely related to the graph-based dependency parsers than the transition-based ones. Even though the actual parse is not performed using the maximum spanning tree algorithm, their approach essentially performs an exhaustive search. The parsing problem is reformulated as an equivalent Integer Linear Programming problem, making it possible to impose linguistically motivated constraints. This results in a weakening of the assumption that each dependency decision is independent of all the others. However, it is worth noting that they applied additional constraints only for four languages, and that these constraints for some languages had very little influence (e.g., few constraints were violated in the first place), making it very similar to MSTParser with no sibling information at all. Error propagation therefore ought to be even less problematic than for MSTParser.
4.2 Error Propagation Metrics

This section begins with a presentation of two new attempts at measuring the effect of error propagation. This is followed by applying them to the parsers discussed above, MSTParser, MaltParser, LTH-parser and ILP-parser. The parsed test data sets from the twelve official languages in the CoNLL shared task 2006, which are freely available at the CoNLL shared task 2006 web page, have been merged into one file by concatenation for each parser, in order to hide the influence of specific languages. Since each test set has approximately the same size (~5000 tokens, as mentioned in chapter 3), the total number of tokens is slightly over 50k.

4.2.1 Error Propagation Metrics

For transition-based parsers that start parsing at the first token of a sentence, one can conceptually distinguish between classification errors and errors caused by previous errors, but in most situations it is hard to tell them apart. As discussed in section 2.3 for the arc-eager version of Nivre’s algorithm in MaltParser, a classification error occurs when the parser model wrongly predicts the next transition. Whereas all other errors can be a combination of both, the first erroneous decision of a sentence must be a result of pure misclassification. The term Pre will be used for pure misclassification errors and Post for all the other errors. It is worth noting that a subsequent error can also be a result of misclassification alone, especially if the previous error happened so much earlier that the feature model is not affected, but only the first error is guaranteed to be a pure misclassification.

In order to compare the accuracy for pure classification errors with the other types of errors, each parsed sentence is divided into two parts. The first part consists of all tokens from the first token up to and including the first erroneous token, which could be either an ASL or ASU error. The second part contains all other subsequent tokens. For instance, let us say that a sentence with ten tokens has been parsed and evaluated as follows

+ + + - - + + - + +

where + means correctly parsed and - incorrectly parsed. The first part then consists of tokens 1 . . . 4 and the second part tokens 5 . . . 10. Pre is then defined as: the proportion of - in the first part of all sentences in the parsed data, and Post is defined as: the proportion of - in the second part of all sentences in the parsed data. For this single sentence, Pre=25% and Post=33%. For multiple sentences, they are computed as a micro-average over tokens. That is, Pre is defined as:

\[
\frac{\sum_{i=1}^{s} \text{# of erroneous tokens in Pre for sentence } i}{\sum_{i=1}^{s} \text{# of tokens in Pre for sentence } i}
\]

and Post as:

\[
\frac{\sum_{i=1}^{s} \text{# of erroneous tokens in Post for sentence } i}{\sum_{i=1}^{s} \text{# of tokens in Post for sentence } i}
\]
where $s$ is the number of sentences. Note that the number of erroneous tokens in Pre is zero for all sentences without errors, and that these sentences do not have any impact on Post.

In the experiments, Pre and Post will be used unnormalized, as just described. But in order to compare Pre and Post for different parsers, they will also be normalized in relation to overall error rate for each parser, where we define error rate as 100% minus any labeled or unlabeled measurement. This is done by dividing Pre and Post, respectively, with the overall error rate, since all parsed tokens are part of either Pre or Post, but not of both.

Another way of capturing error propagation is to measure the error rate of tokens subsequent to erroneous tokens. If a parser suffers from error propagation, the probability that a new error will occur right after an error ought to be higher than if the first error did not occur. The definition of this metric can be illustrated by the same example as for Pre and Post. All tokens are classified by an integer value representing when the latest error occurred:

```
+ + + - - + + - + +
1 2 3 4 1 1 2 3 1 2
```

For instance, the first error occurs at position 4, so the fifth token is classified as 1, since it is located 1 step to the left of the latest error. Token 6 also belongs to class 1 since token 5 is incorrect, whereas token 7 is in class 2, located 2 steps to the right of the latest error. The initial correctly parsed tokens (1...3) are here classified as if there is an error just before the sentence, at position 0. Since the error rate is of interest here, the proportion of - for each class (1, 2, 3, ...) is measured. In this tiny and unrepresentative example, the error rate for class 1 = $1/4 = 25\%$, class 2 = $0/3 = 0\%$, class 3 = $1/2 = 50\%$, and class 4 = $1/1 = 100\%$.

Measuring the effect of error propagation in this way will be referred to as error clustering, since it does not make sense to talk about error propagation for all types of data-driven dependency parsers. This type of metric could have been defined in a different way, e.g. not including every initial +. However, just as for Pre and Post, a fairer comparison between parsers with different overall accuracy is achieved by normalizing according to the overall error rate. This is done by dividing the error rate for each class by the overall error rate, which would not be possible unless each and every token is included exactly once in some class.

### 4.2.2 Pre and Post

In table 4.1, Pre is the parsing accuracy for all tokens from the start of a sentence up to and including the first error, whereas Post is the parsing accuracy for all subsequent tokens. The accuracies reported in the table are the error rates for the labeled attachment score.

The first column (Total) shows the total AS$_L$ error rate for the parsers, also reported in Buchholz and Marsi (2006). MSTParser had the best result, but there was no statistically significant difference in comparison with
4.2 Error Propagation Metrics

<table>
<thead>
<tr>
<th></th>
<th>ASₘ Error Rate</th>
<th>Norm. ASₘ Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Pre</td>
</tr>
<tr>
<td>MaltParser</td>
<td>19.8</td>
<td>12.0 (42.2%)</td>
</tr>
<tr>
<td>LTH-parser</td>
<td>25.1</td>
<td>20.2 (32.3%)</td>
</tr>
<tr>
<td>MSTParser</td>
<td>19.7</td>
<td>12.7 (41.3%)</td>
</tr>
<tr>
<td>ILP-parser</td>
<td>22.1</td>
<td>14.2 (37.9%)</td>
</tr>
</tbody>
</table>

Table 4.1: Pre, Post and Total error rate for MaltParser, LTH-parser, MSTParser and ILP-parser.

MaltParser. Looking at the error rate for Pre for MaltParser at 12.0%, one can see that it is considerably lower than the total error rate. It is also lower than Pre for MSTParser, which is 12.7%. The percentages in parentheses show the proportion of tokens in each set of Pre-tokens, that is, 42.2% of all tokens belong to Pre and consequently 57.8% to Post. Post for MaltParser is also much higher then the total error rate, and higher than the error rate for Post for MSTParser. In other words, the figures are consistent with the assumption that error propagation does affect the results for MaltParser, due to its local and greedy parsing approach.

LTH-parser and ILP-parser, on the other hand, have higher error rates in comparison with the two top-scoring parsers, and have consequently higher values for Pre and Post. This, in turn, entails a lower proportion of tokens in Pre, e.g., 32.3% for LTH-parser. An adequate comparison requires that Pre and Post are normalized according to the total error rate. The shown normalized figures are also based on the error rates for ASₘ in the Total-column (i.e., ASₘ error rate = 100% – ASₘ = either a head or label error, or both). For instance, Pre for MSTParser is computed according to: 24.7/19.7 = 125.2%, where 19.7% is the error rate for MSTParser. In other words, there is a 25.2% increase in probability that a token in Post for MSTParser is misclassified in relation to its average error rate of 19.7%.

MaltParser has – not surprisingly – a lower error rate than MSTParser for Pre (−3.4%) and a higher error rate for Post (+3.5%). That is, in relation to MSTParser, MaltParser has a higher proportion of errors when an error has already occurred. What is more noteworthy is that LTH-parser in fact has the highest normalized error rate for Pre, and the lowest normalized error rate for Post. This may be an indication that an incremental parsing strategy from left to right is prone to error propagation, but that this problem can be alleviated to a large extent if the strictly local and greedy decision strategy is relaxed using beam search.

In terms of normalized error rate, ILP-parser is located between MSTParser and LTH-parser for both Pre and Post. In other words, ILP-parser exhibits less clustering of errors than MSTParser, which accords well with the fact that ILP-parser uses less contextual information during parsing than MSTParser.

The possibility to counteract error propagation is probably one of the reasons why Titov and Henderson’s ISBN Dependency Parser (2007b) per-
formed so well in the CoNLL shared task 2007. Their parser was one of the highest scoring single parsers, and as mentioned in chapter 2, it is an incremental left-to-right parser with beam search.

### 4.2.3 Error Clustering

The other way of trying to capture error propagation, proposed in section 4.2, is to measure the error rate of tokens subsequent to erroneous tokens. Table 4.2 contains the error rate for ASₖ using the same parsers and data sets as in the subsection above. The figures show the average error rate for the seven subsequent positions after an incorrect token. The proportion of tokens that are located further away from the latest head or label error are placed in the rightmost bin “$>7$”.

For example, MaltParser has an error rate of 29.3% for tokens immediately following an error. This can be compared to MaltParser’s average error rate of 19.8% (100%− the value in Total in table 4.1). All parsers exhibit a similar behavior with decreasing error rates as the distance to the latest error increases. This holds even for MSTParser and ILP-parser despite their very limited knowledge of the local context. Even if the term error propagation is not applicable to MSTParser and ILP-parser, both tables 4.1 and 4.2 show that they also clearly cluster errors together.

It is worth noting that the error clustering figures drop below the total error rates after 2 for MSTParser, MaltParser and ILP-parser, and already after 1 for LTH-parser (with its total error rate of 25.1%). In relation to MaltParser, this is another indication that LTH-parser recovers much faster from errors.

Table 4.3 contains the proportion of tokens in the bins of table 4.2 (as well as in figure 4.1, presented below). For convenience, the bins that contribute with more errors than average for MSTParser, MaltParser and ILP-parser,
4.2 Error Propagation Metrics

Figure 4.1: Normalized error propagation curve in CoNLL-X for AS, where • = MaltParser, △ = MSTParser, ◦ = LTH-parser, ■ = ILP-parser.

i.e., bins 1–2, have been merged into one. Tokens in bins 3–7 are also merged into one. Concentrating on for instance MaltParser, one can see that the tokens with the highest error rate, located one or two steps after an error, in fact constitute almost half of all tokens (45.1%). The figures are very similar for the other parsers as well, with an even higher proportion for 1–2 for LTH-parser and ILP-parser due to their overall higher error rate.

In order to really compare the amount of error clustering of the parsers, the figures need to be normalized with respect to each parser’s overall error rate. The normalized error clustering metrics are shown in figure 4.1. For instance, the data point for MaltParser in bin 1 (148%) means that there is a 48% increase in probability for a token located right after a misclassified token, compared to the average error rate. MSTParser, on the other hand, exhibits just below +40% probability for bin 1, and LTH-parser only +34% probability. ILP-parser, with the least amount of contextual information during parsing, has the lowest increased probability of an additional error for tokens in bin 1.

Looking to the right in the diagram of figure 4.1, the relationship between the parsers is now exactly the opposite. ILP-parser has the highest normalized error rate and MaltParser the lowest. The curves cross one another around bins 3 and 4. The general tendency is that the curve for MaltParser has the steepest slope, followed by MSTParser, which in turn is followed by LTH-parser. ILP-parser has the flattest curve. The curve for MaltParser after bin 3 is constantly located below the curves for the other parsers. In other words, the errors are the most evenly distributed in the parsed sentences for ILP-parser, and the least evenly distributed for MaltParser, indicating that MaltParser uses the parsing strategy that is the most sensitive to errors in the local context. The curves also indicate that
the contextual information is even more constrained for ILP-parser than for MSTParser with its context of one sibling.

One final observation is that LTH-parser does not seem to suffer the least from error propagation, which is the case for the normalized figures for \textit{Pre} and \textit{Post}. It could in fact be that beam search is beneficial because it postpones the first error until later in the sentence, but does not actually help error propagation when the first error has occurred. However, this requires more analysis.

4.3 Conclusion

This chapter has proposed two new metrics for detecting and measuring the effect of error propagation. Although we have not been able to evaluate the metrics as such, the findings in subsections 4.2.2 and 4.2.3 are consistent with the assumption that error propagation is problematic for transition-based parsers like MaltParser. The results also indicate that error clustering occurs even for parsers without the assumed error propagation, such as the evaluated graph-based parsers. It is therefore the relative increase for transition-based parsers compared to others parsers that signals the effect of error propagation. The results also indicate that one way to counteract, or at least to postpone, the first error this is to use beam search.

Although not completely evaluated, the error propagation metrics will be applied as diagnostic instruments throughout the rest of the thesis to investigate whether any positive effect on accuracy for different types of transformation and combination can be related to error propagation. It is worth pointing out that the arc-eager version of Nivre’s algorithm in MaltParser, used in this chapter and throughout the rest of the thesis, actually constructs some arcs from right to left. The metrics for error propagation presented here are not able to capture such situations. When producing multiple left dependents of the head, where the head is the first node in the list of remaining input nodes, the dependents are attached from right to left. This is, however, the only exception, the majority of arcs being constructed from left to right.
Chapter 5

Pseudo-Projective Transformations

This is the first chapter in the transformation track. The idea presented in
this chapter comes from Nivre and Nilsson (2005), while the experimental
part is essentially based on Nilsson et al. (2007). As discussed in chapter 2,
the full potential of dependency-based parsing cannot be realized unless non-
projective dependency trees can be produced, which is impossible for the
large number of projective dependency parsing algorithms. We will in this
chapter explore a transformational approach to this problem. The output
of a projective parsing algorithm will, by means of tree transformations, be
turned into non-projective trees.

In section 5.1, tree transformations that transform any non-projective de-
pendency tree to a projective one are described, including the corresponding
inverse transformation. This is followed by section 5.2, presenting the ex-
perimental setup of the experiments of subsequent sections. Section 5.3
uses the tree transformation without involving any parser, in order to study
how much distortion they cause. The following section aims at investigat-
ing if the tree transformations can improve parsing accuracy, and to what
extent they then can improve accuracy across languages and treebanks (sec-
tion 5.4). The chapter ends with a conclusion in section 5.5.

5.1 Pseudo-Projective Transformations

The idea investigated here is whether a projective dependency parser can
be trained so that non-projectivity can be recovered during postprocessing.
This can be facilitated by adding information during the transformation of
the training data, informally called pseudo-projective dependency parsing.

One may ask whether it is worth the effort to pay so much attention to
such a rare linguistic phenomenon as non-projectivity. After all, even in
languages with a free word order, usually only around 2% or less of the arcs
are non-projective in existing dependency treebanks (see section 5.2 later).
The penalty for neglecting these is therefore usually quite low. However,
the situation looks different if non-projectivity is instead measured by the
proportion of non-projective dependency trees. A treebank with approx-
imately 2% non-projective arcs can have 25% non-projective dependency
trees. In other words, it is unsatisfactory not to be able even in theory to
derive a completely correct dependency tree for that many sentences.
This problem can be solved by trying to recover non-projective arcs from the output of a projective dependency parser. The solution mentioned in section 2.4, the approach proposed by Hall and Novák (2005), relies only on postprocessing. The idea of pseudo-projective dependency parsing (Nivre and Nilsson, 2005) combines postprocessing with a preceding preprocessing step. It is worth noting that preprocessing only takes place before the training begins. Furthermore, the postprocessing step is only applied when the complete projective dependency tree has been constructed by the parser. The pseudo-projective transformations, which constitute the preprocessing and postprocessing, are presented in this section.

The overall methodology for all transformations in the transformation track was briefly mentioned in chapter 1 but is worth repeating here. It is schematically the same for all transformations and is divided into these four steps:

1. Apply the tree transformation to the training data.
2. Train a parser model using the transformed training data.
3. Parse a text using the parser model.
4. Apply the corresponding inverse transformation to the output of the parser.

In detail, the rest of this section will discuss steps 1 and 4 for pseudo-projective parsing.

5.1.1 Step 1: Projectivization

Any non-projective dependency tree can be projectivized, i.e. transformed into a projective dependency tree, by replacing all non-projective arcs with projective ones. As noted by Kahane et al. (1998), this can be automatized by using a lifting operation. It informally moves all non-projective arcs “upwards” in a dependency tree until the lifted arcs become projective, according to definition 3. However, Kahane et al. (1998) do not propose an inverse transformation. The idea of building a projective tree by means of lifting also appears in Kunze (1968) and is used by Hudson (1990). Kahane et al. (1998) define a lifting operation, but the one presented here is slightly different. For a dependency tree \( G = (V, E, L) \) obeying definition 2, it is defined as:

\[
\text{LIFT}(j \rightarrow k) = \begin{cases} 
  i \rightarrow k, & \text{if } i \rightarrow j \in E \\
  \text{undefined, otherwise}
\end{cases}
\]

The lifting operation takes an arc \( j \rightarrow k \) as argument. In case \( j \) has a head token \( i \), which according to definition 2 is unique, it returns the lifted arc \( i \rightarrow k \). If no arc such as \( i \rightarrow j \) exists, it means that \( i \) is attached to the root token, and the arc \( j \rightarrow k \) can consequently not be lifted.

\text{LIFT} is applied to each non-projective arc in \( E \), which is replaced by the output of \text{LIFT}. However, the result of \text{LIFT}(e) for a non-projective arc \( e \)
5.1 Pseudo-Projective Transformations

is not necessarily projective. It may therefore be necessary to apply $\text{LIFT}$ more than once and preferably in a deterministic manner. The following algorithm is deterministic and the returned tree is projective:

$$\text{PROJECTIVIZE}(G = (V, E, L))$$

1. $E' \leftarrow E$
2. \textbf{while} some arcs in $E'$ are non-projective
3. \hspace{1em} $e \leftarrow \text{SMALLEST-NONP-ARC}(E')$
4. \hspace{1em} $E' \leftarrow (E' - \{e\}) \cup \{\text{LIFT}(e)\}$
5. \textbf{return} $(V, E', L)$

Since more than one arc can be non-projective in a tree, $\text{SMALLEST-NONP-ARC}$ at line 3 determines the lifting order. With no particular reason other than making the algorithm deterministic, it selects the arc with the smallest span (measured as the distance between the head and the dependent), breaking ties from left to right. The lifting order will only matter if two or more non-projective arcs in the same sentence interact in some way, but usually the result is independent of it. An alternative approach would be to lift the arcs in the order of increasing height. The selected approach has nevertheless a very limited influence in practice, because interactive non-projective arcs are relatively rare in existing dependency treebanks.\(^1\)

Line 4 performs the arc replacement, which together with line 3 is repeated until the (unlabeled) tree $(V, E')$ becomes projective. It is worth pointing out that an arc with the root as head cannot be non-projective (see the discussion after theorem 1), which entails that $\text{LIFT}$ at line 4 can never return an undefined result.

Figure 5.1 shows a non-projective Czech sentence. The arc $\text{jedna} \rightarrow Z$ violates the projectivity constraint, as $\text{jedna}$ does not dominate $\text{je}$. This arc will consequently be removed by the algorithm and replaced by the returned arc of the lift operation, which is the arc $\text{je} \rightarrow Z$. This arc is projective because the only token it spans over ($\text{nich}$) is dominated by $\text{je}$. Since no more non-projective arcs remain, the algorithm returns the projectivized dependency tree shown in figure 5.2. Following the terminology of Kahane et al. (1998), the original head ($\text{jedna}$) of $Z$ is denoted the \textit{syntactic head} and the new head ($\text{je}$) of $Z$ the \textit{linear head}.

As illustrated by the lower dependency tree of figure 2.2 (page 14), where $\text{calls} \rightarrow \text{from}$ is non-projective, one single application of $\text{LIFT}$ is not always sufficient for each arc. $\text{LIFT}$ replaces it with another non-projective arc, i.e. $\text{phone} \rightarrow \text{from}$, which must be replaced by the projective arc $\text{received} \rightarrow \text{from}$ with yet another application of $\text{LIFT}$.

The dependency labels of the lifted arc and the arc it was lifted over in the figure have been replaced by two variables, $\text{LL}$ (lifted label) and $\text{PL}$ (path label). It is linguistically questionable whether the original labels can be used in the projectivized dependency structure without distorting the

\(^{1}\text{Both approaches are implemented in the open-source software MaltParser, from version 1.1 (and can be used stand-alone without involving any of the parsing algorithms in MaltParser), but only the first strategy is applied in the coming parsing experiments.}\)
meaning. In the former case, a label is tied to the properties of the head token that an arc governs. This holds in the latter case as well, since the number of tokens and the token properties in a subtree may affect the meaning of the tokens dominating them. For instance, *jedna* in figure 5.1, with an *AuxP*-dependent, may not have the same meaning as in figure 5.2, without an *AuxP*-dependent. Moreover, all these labels are important because they will be used to encode the lifts in different ways in order to facilitate the inverse transformation. This will be discussed in subsection 5.1.2. We end this subsection by proving that the algorithm *PROJECTIVIZE* is correct.

**Theorem 1** For any dependency tree $G = (V, E, L)$ satisfying definition 2, the algorithm $\text{PROJECTIVIZE}(G)$ will always (1) terminate and (2) return a projective dependency tree $(V, E', L)$ satisfying definition 4.

**Proof 1** Definition 3 states that an arc $h \to d$ is projective iff all the tokens it spans over are dominated by the head of $h \to d$. The key consequence of definition 2 is that the dependency graphs form rooted trees, where token 0 is the only token that dominates all other tokens ($0 \to^* k, k \in V$). Therefore, for all arcs $h \to d$ such that $h = 0$, it follows that all tokens $k$ (where $0 < k < d$) are also dominated by $h$. In other words, arcs with the root as head ($0 \to d$) are projective. This conclusion also justifies why it is unnecessary to deal with the undefined case of the *LIFT*-operation.

What is left to show is that any (non-projective) dependency tree after a finite number of lifts in *PROJECTIVIZE* forms a dependency tree with all arcs attaching to the root. As a dependency tree has a finite number of tokens
V = \{0,\ldots,n\}, it also has a finite number of arcs E, where |E| = |V| − 1. The definition of a dependency tree implies that 0 →* i → k, where i → k is to be lifted. The path distance from the root to k can in the worst case be |E|. If 0 →* j → i holds, the lift operation replaces i → k with j → k, where 0 →* j → k also holds. The lift operation can be applied at most |E| − 1 times until 0 → k, which according to the previous paragraph is enough to conclude that the arc is projective.

The loop spanning from line 2 to 4 will not stop iterating until all arcs are projective, which according to definition 4 implies that the complete tree is projective. It will terminate since the maximum number of iterations is finite. When it has stopped iterating, the new dependency tree is no longer manipulated by the algorithm and it will therefore return a projective dependency tree.

In the worst case, an arc can be lifted at most l = |E| − 1 times, which implies that the height of the dependency tree is reduced to l − 1. This, in turn, means that the next arc can be lifted at most l − 1 times, and so forth, until the height of the dependency tree is 1. In total, at most

\[ l + (l - 1) + \ldots + 1 = \sum_{i=0}^{l} (l - i) = \frac{l^2 - l}{2} \]

lifts can be applied. Since determining whether a dependency tree is projective or not can be done in linear time in relation to the number of tokens (Havelka, 2005), and lines 3–4 take constant time, PROJECTIVIZE has in the worst case quadratic running time.

### 5.1.2 Encodings

Whereas the projectivization algorithm is deterministic, the same does not hold for the inverse transformation. That is, PROJECTIVIZE, mapping from dependency trees to dependency trees, is a many-to-one function. This can be illustrated by the two different non-projective dependency trees in the upper part of figure 5.3. They are mapped to the same projective dependency tree by PROJECTIVIZE, shown at the bottom of the figure. The difference is that the upper left dependency tree requires two lifts, and the upper right only one.

The inverse transformation performs a search for the syntactic head “downwards” among the tokens dominated by the linear head. The linear head of the lifted arc in figure 5.3 is B, and the inverse transformation needs guidance to determine the correct syntactic head, that is, either C or D. The linear head in figure 5.2 dominates four potential syntactic heads, the tokens 4, 5, 6, and 7. To facilitate the search, four encoding schemes, including one baseline version, have been implemented for the labels of lifted arcs and the arc(s) which have had arcs lifted over them (known as the path), illustrated by the labels LL and PL in figure 5.2.
The encoding schemes incorporate different amounts of information concerning the lifts. Needless to say, the more such information there is in the projectivized data, the easier it will be to recover the original non-projective structure. In principle, the exact syntactic head could in some way be encoded in the labels, yielding an error-free inverse transformation. However, this would imply a potentially infinite number of labels. The labels should be learned by the parser and assigned to arcs of new unseen sentences, and the more information that is incorporated into the labels, the harder the learning task is for the parser. Finding the right balance of information in the new labels is therefore important, which may depend on properties such as the size of the training data, the number of distinct labels and the amount of non-projectivity.

The four encoding schemes are shown in table 5.1, with the labels for $LL$ and $PL$ of figure 5.2. In the BASELINE encoding, the original labels are kept unchanged. This means that the lifted arc label of $Z$ becomes $d$ ($\text{AuxP}$) and the path label of $\text{jedna}$ becomes $h$ ($\text{Sb}$). This is the simplest possible encoding, but it does not include any information that helps finding the syntactic head during the inverse transformation.

In the second encoding, HEAD, the lifted arc concatenates the label of its syntactic head ($h$) to its original label ($d$), separated by the symbol $\uparrow$.

The path labels will, however, contain the same information as BASELINE. In other words, the lifted arc signals what the label of its syntactic head is, which will guide the inverse transformation later on. Since $\text{Sb}$ is the label of the syntactic head in figure 5.1, $LL=\text{AuxP}\uparrow\text{Sb}$ in figure 5.2.

The encoding PATH, on the other hand, does not include the label of the syntactic head in the lifted arc. It only signals that it has been lifted ($LL=\text{AuxP}\uparrow$), which compared to HEAD leaves the label of the syntactic head unspecified. However, the path labels are extended in such a way that

\[
\begin{align*}
<\text{ROOT}> & \quad A \quad B \quad C \quad D \\
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4
\end{align*}
\]
5.1 Pseudo-Projective Transformations

Table 5.1: Projectivity encodings. LL = lifted label (value of LL in figure 5.2), PL = path label (value of PL in figure 5.2).

<table>
<thead>
<tr>
<th>Encoding</th>
<th>LL</th>
<th>PL</th>
<th># labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline ($p_0$)</td>
<td>$d$</td>
<td>$h$</td>
<td>$</td>
</tr>
<tr>
<td>Head ($p_H$)</td>
<td>$d'h$</td>
<td>$h$</td>
<td>$(Sb)\cdot(</td>
</tr>
<tr>
<td>Path ($p_P$)</td>
<td>$d’h$</td>
<td>$h$</td>
<td>$(Sb)\cdot(</td>
</tr>
<tr>
<td>Head+Path ($p_{HP}$)</td>
<td>$d'h$</td>
<td>$h$</td>
<td>$(Sb)\cdot(</td>
</tr>
</tbody>
</table>

they indicate that (at least) one arc has been lifted over them, attaching the symbol $\downarrow$ to all labels along the path ($PL=Sb\downarrow$). This creates a continuous trace from the linear to the syntactic head.

The last encoding, which combines Head and Path, uses the same label $LL$ as Head and the same path labels as Path. This is the most informative encoding and provides the inverse transformation with a lot of information to find the syntactic head.

The last column of table 5.1 reveals the theoretical upper bound on the number of labels that the projectivized data can contain. Path has only a linear increase (possible labels: $r$, $r\uparrow$, $r\downarrow$, $r\uparrow\downarrow$). Due to the previously discussed trade-off, it is therefore the least informative encoding and should presumably at the same time be the least complicated labels for the parser to learn. The two others have a quadratic increase, because any label can be combined with any label in $R$, probably making the assignment of labels during parsing harder. Other factors of course influence the complexity of the learning task, too, such as the number of relabeled arcs. This will be discussed in conjunction with the results in section 5.4.

By and large, including the label of the syntactic head in the label of the lifted arc really limits the number of possible syntactic head candidates, but if there are few distinct labels the number of candidates increases on average. This is not a problem if path information can be used, because a complete trace back to the syntactic head is provided. However, path information may be misleading whenever non-projective arcs interact during projectivization, that is, they are lifted along the same path but have different syntactic heads. Furthermore, the situation becomes even more delicate during parsing since the parser may assign incomplete or inconsistent pseudo-projective information, such as broken traces back to the syntactic head and syntactic head labels in the lifted arc labels that do not exist further down in the tree. Resolving these issues is the topic of the subsection below.

5.1.3 Step 4: Deprojectivization

The deprojectivization (i.e. the inverse transformation) can rely on several different methods, including various machine-learning methods. Here a completely algorithmic strategy has been implemented, making use of the pseudo-projective information provided by the applied encoding scheme.
The search for possible syntactic heads is performed among the tokens dominated by the linear head, excluding the tokens dominated by the lifted arc. The first token conforming to the provided pseudo-projective information is selected as the new syntactic head. The new (probably non-projective) arc to the new syntactic head replaces the lifted arc.

The results presented later in this chapter are based on the breadth-first search approach, which in preliminary experiments has been the most accurate. (Other investigated approaches were for example depth-first, and closest-first that considers the tokens closest to the linear head first.) The search starts by identifying each lifted arc $i \rightarrow k$ in the dependency tree, which is easily done by looking for the symbol $↑$ in the labels of all tokens, basically $i \overset{d↑}{\rightarrow} k$ for PATH and $i \overset{d↑h}{\rightarrow} k$ for the two others, from left to right in the sentence. For each such arc, the search is done breadth-first and left-to-right, starting at $i$ (excluding the subtree of $k$). For the lifted arc in figure 5.2 with the linear head je, it means that the tokens $\{5, 6, 4, 7\}$ are considered left-to-right as possible syntactic heads. Then, depending on the encoding, the first candidate token $m$ satisfying the following conditions is selected (cf. table 5.1 for the labeling):

- **For HEAD:** If $m$ has the label $h$ to its head, i.e., $l \overset{h}{\rightarrow} m$, then replace $i \overset{d↑h}{\rightarrow} k$ with $m \overset{d}{\rightarrow} k$.
- **For PATH:** If all arc labels between $m$ and $i$ have the form $\overset{p↓}{\rightarrow}$ and $m$ does not have any outgoing arcs of the form $m \overset{p↓}{\rightarrow} o$, then replace $i \overset{d↑}{\rightarrow} k$ with $m \overset{d}{\rightarrow} k$.
- **For HEAD+PATH:** The same as PATH with the additional condition that the arc from $m$ to its head must have the label $h↓$, i.e. $l \overset{h↓}{\rightarrow} m$.

For instance, when the inverse transformation for HEAD finds jedna (token 5, the first to be checked), it sees that the arc label to jedna (Sb) matches the label to the right of $↑$ in the lifted arc label (AuxP↑Sb). It correctly recognizes jedna as the syntactic head, and accordingly replaces je→Z labeled AuxP↓Sb with jedna→Z labeled AuxP.

The syntactic head for PATH is not allowed to have a dependent with a label containing ↓, which happens for repeated lifts, as the dependent would then be a more likely syntactic head. In the example, where the arc from je to jedna is labeled Sb↓ without any outgoing arcs labeled $p↓$, jedna again becomes the correct syntactic head.

Finally, when PATH and PATH+HEAD have deprojectivized a dependency tree, each arc label of the form $\overset{p↓}{\rightarrow}$ is replaced by $\overset{p}{\rightarrow}$.

### 5.1.4 Back-off for the Deprojectivization

All three encodings adopt a back-off in case none of the possible candidates can be selected given the above conditions of the chosen encoding. There

\[^3\text{Depth-first would be }\{5, 4, 6, 7\} \text{ and closest-first }\{4, 5, 6, 7\}.\]
are two possible explanations why the search may fail: the information is inconsistent or incomplete (which it often is when parsing), or the correct syntactic head is not among the candidates due to the interaction of several lifts (i.e., broken paths, which can happen when an arc along a path of a lifted arc is lifted itself). Therefore, if a search fails and any subsequent lifted arcs in the sentence succeed in finding their syntactic heads, the failed search is repeated, hoping that a broken path has been repaired. This will partially solve the latter problem. The process iterates until no subsequent searches succeeds any longer (which in practice happens after very few iterations, normally 1-3).

It is also important to mention that the algorithm for HEAD-PATH relaxes the search condition if the search fails, namely that it performs a search equivalent to HEAD. This means that all paths are admitted, not just those of the form $p_\downarrow$.

For any remaining lifted arc with a failed search, irrespective of the encoding, the algorithm’s last resort is to leave the lifted arc in place. That is, it makes the linear head into the syntactic head, and the label is simply changed to $d$ (i.e. $i \xrightarrow{d} k$).

## 5.2 Experimental Setup

As mentioned in the beginning of this chapter, the pseudo-projective experiments are divided into two sections, treebank transformations in section 5.3 and parsing experiments in section 5.4. The experiment for the treebank transformations uses the projectivization and deprojectivization transformations alone to study the number of errors they cause. The other experiment aims at investigating whether the tree transformations can improve parsing accuracy. This section describes data sets of these experiments.\(^4\)

The experiments use the five treebanks presented in subsection 3.3.1, i.e., data from the CoNLL shared task 2006. The treebank transformations experiment uses all and only the training data shown in the table. During the development phase of the parsing experiments, the training data have been divided into ten equally large sets in a pseudo-randomized way as described in section 3.3. Since having less test data implies less reliability in the comparison, ten-fold cross-validation was conducted during the development phase for the smaller treebanks, i.e., Slovene (SDT), Arabic (PADT) and Dutch (Alpino), in order to obtain a more reliable evaluation. The parser is trained ten times, where each set is excluded from the training data once while the excluded set is the test set. The various metrics are presented by computing the mean and standard derivation (denoted $\sigma$ in the experiments) of the ten test sets. Ordinary training is used for the two

\(^4\)Moreover, the software used for the pseudo-projective transformations in this chapter can be downloaded from: http://w3.msi.vxu.se/users/jni/pproj/. However, consider downloading the MaltParser 1.1 (or later) instead, which contains a stand-alone reimplementation.
larger treebanks, German (Tiger) and Czech (PDT) during the development phase, where 20% (sets 0 and 9) was used as development test data and the other 80% (sets 1–8) as training data.

Table 5.2 gives information about the training sets which is of interest for the pseudo-projective transformations, ordered by ascending treebank size. As the last column reveals, PDT contains quite a high proportion of non-projectivity, since almost every fourth dependency tree contains at least one non-projective arc. The proportion of non-projectivity in SDT is quite similar to the proportion in PDT, but, as noted in chapter 3, it has much less training data.

The table also reveals that the number of non-projective arcs in PADT (Arabic) is less than 25% in relation to PDT and SDT, although the number of non-projective sentences is still as much as half compared to PDT and SDT. As noted in chapter 3, PADT has very long sentences. According to the organizers of the shared task, a sentence in the CoNLL format is for PADT in many cases instead an entire paragraph.

One of the characteristics of the Dutch treebank Alpino is the high amount of non-projectivity. More than one third of all sentences are non-projective, making it especially interesting to study. In the Alpino treebank, neglecting non-projective arcs would result in at least 5.4 percentage points lower accuracy as the asymptotic goal (compared to 100% correct accuracy) in an otherwise maximally correct dependency structure of a projective parser.

The size and high proportion of non-projectivity compared to the treebanks based on the Prague school style make the German Tiger treebank interesting for the experiments with the pseudo-projective transformations below. As table 5.2 shows, the treebank contains 2.3% non-projective arcs, and 27.8% of all sentences contain at least one non-projective arc.

The evaluation in the parsing experiments excludes all tokens having the property “Punctuation” according to the Unicode directives. This is the default behavior of the evaluation script of the CoNLL shared task 2006. The treebank transformation experiment includes all tokens, to enable comparison with other published studies.
5.3 Treebank Transformations

As the implemented transformations and inverse transformations are not perfect, they always distort the data to some degree. The question pursued in this section is to what extent the various versions of the pseudo-projective transformation introduce errors, which is of interest for the parsing experiments. The experimental procedure is as follows:

1. Apply a transformation to a complete training data set of a treebank.
2. Apply the corresponding inverse transformation to the output of step 1.
3. Compute the attachment score by comparing the original untransformed treebank with the output of step 2.

The figures presented below are constrained to AS\textsubscript{U}, mainly because the corresponding AS\textsubscript{L} figures do not contribute any new interesting knowledge (besides being slightly lower than the AS\textsubscript{U} figures). Table 5.3 contains the AS\textsubscript{U} figures for the four pseudo-projective transformations Baseline (\textit{p}\textsubscript{0}), Head (\textit{p}\textsubscript{H}), Path (\textit{p}\textsubscript{P}), Head+Path (\textit{p}\textsubscript{HP}), with the AS\textsubscript{U} (which in this case is identical to unlabeled recall) restricted to the non-projective arcs in parenthesis.

The first thing to note is that 100\% minus \textit{p}\textsubscript{0} equals \%-NA in table 5.2 for the five treebanks. Since \textit{p}\textsubscript{0} simply projectivizes the data without any attempt to perform an inverse transformation, each non-projective arc counts as one AS\textsubscript{U}-error. For the other three transformations, it can be concluded that the most informative transformation, \textit{p}\textsubscript{HP}, is also the most accurate one. The distortion introduced is for all treebanks virtually negligible, with no result lower than 99.98\% overall AS\textsubscript{U} (for Alpino), or 99.3\% AS\textsubscript{U} for non-projective arcs alone (for PDT). It is worth noting that despite the fact that there is only a linear increase in the number of pseudo-projective dependency labels for \textit{p}\textsubscript{P}, it has a comparable (SDT and PDT) or higher (PADT, Alpino and Tiger) AS\textsubscript{U} than \textit{p}\textsubscript{H}.

A quite expected general observation is that the more non-projectivity, the lower overall AS\textsubscript{U}. Alpino has the lowest overall AS\textsubscript{U} for all pseudo-projective encodings, probably due to the highest proportion of non-projectivity, whereas PADT with the lowest proportion of non-projectivity has the highest AS\textsubscript{U}. However, as regards the AS\textsubscript{U} figures for the non-projective arcs alone, the situation is in fact reversed. The inverse transformations for \textit{p}\textsubscript{H}, \textit{p}\textsubscript{P} and \textit{p}\textsubscript{HP} manage to correctly recover most non-projective arcs for Alpino.

PADT, on the other hand, has a very low AS\textsubscript{U} of 73.3\% for \textit{p}\textsubscript{H}, but has AS\textsubscript{U}-figures for \textit{p}\textsubscript{P} and \textit{p}\textsubscript{HP} that are comparable to the other treebanks. A possible reason for this is that PADT has the most skewed distribution of dependency labels. As much as 35\% of all arcs have the label \textit{Atr}, which is more than for any other treebank.\footnote{Also, there are actually 36\% non-projective arcs with a (syntactic) head labeled \textit{Atr}.} This is more problematic for \textit{p}\textsubscript{H}, since the number of possible syntactic heads increases for the numerous
non-projective arcs with a syntactic head labeled $Atr$. The two other transformations are less sensitive to this, as the path back to the correct syntactic head is clearly marked.

Two other variables that may affect the amount of distortion are presented in tables 5.4 and 5.5. The former table contains the distribution of the number of lifts that non-projective arcs have to undergo in order to find their linear heads. One can expect that the more deeply nested a non-projective arc is, the harder it will be to correctly find its syntactic head. The figures reveal some interesting differences between the treebanks. For example, 93.8% of all arcs in PDT require only one lift before they become projective, whereas the corresponding figure for PADT is as low as 66.5%. PADT also has a very high proportion of very deeply nested non-projective arcs (>3) in comparison to the other treebanks. These facts are problematic for the breadth-first approach of the inverse transformation for PADT, since the search for the syntactic head investigates depth 1 first, before moving on to depth 2, followed by depth 3, and so on. It is worth noting that the Danish Dependency Treebank (Kromann, 2003), for which an improvement in accuracy was not observed in Nivre and Nilsson (2005), is both larger and less deeply nested than PADT. One can therefore anticipate that a positive effect of the pseudo-projective parsing will be hard to achieve.

The other variable is the number of distinct dependency labels, which of course is related to the distribution of dependency labels discussed above for PADT. In this particular experiment, a large set of dependency labels, evenly distributed over all arcs, should provide the best conditions to achieve as little distortion as possible. For instance, a more fine-grained set of dependency labels for $Atr$ in PADT is likely to decrease distortion. However, this is usually not a beneficial property during parsing. The task of as-

<table>
<thead>
<tr>
<th></th>
<th>$p_0$</th>
<th>$p_H$</th>
<th>$p_P$</th>
<th>$p_{HP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDT</td>
<td>98.24</td>
<td>99.82 (90.1)</td>
<td>99.82 (90.0)</td>
<td>99.99 (99.4)</td>
</tr>
<tr>
<td>PADT</td>
<td>99.61</td>
<td>99.89 (73.3)</td>
<td>99.96 (90.3)</td>
<td>100.00 (99.5)</td>
</tr>
<tr>
<td>Alpino</td>
<td>94.68</td>
<td>99.47 (90.2)</td>
<td>99.74 (95.2)</td>
<td>99.98 (99.7)</td>
</tr>
<tr>
<td>Tiger</td>
<td>97.74</td>
<td>99.76 (89.5)</td>
<td>99.83 (92.7)</td>
<td>99.99 (99.5)</td>
</tr>
<tr>
<td>PDT</td>
<td>98.11</td>
<td>99.86 (92.5)</td>
<td>99.86 (92.7)</td>
<td>99.99 (99.3)</td>
</tr>
</tbody>
</table>

**Table 5.3:** Distortion results for the pseudo-projective transformations with figures for $A_{SU}$; in parenthesis: $A_{SU}$ for non-projective arcs.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>&gt;3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDT</td>
<td>88.4</td>
<td>9.1</td>
<td>1.7</td>
<td>0.8</td>
</tr>
<tr>
<td>PADT</td>
<td>66.5</td>
<td>14.4</td>
<td>5.2</td>
<td>13.9</td>
</tr>
<tr>
<td>Alpino</td>
<td>84.6</td>
<td>13.8</td>
<td>1.5</td>
<td>0.1</td>
</tr>
<tr>
<td>Tiger</td>
<td>83.6</td>
<td>14.8</td>
<td>1.4</td>
<td>0.2</td>
</tr>
<tr>
<td>PDT</td>
<td>93.8</td>
<td>5.6</td>
<td>0.5</td>
<td>0.1</td>
</tr>
</tbody>
</table>

**Table 5.4:** The number of lifts for non-projective arcs.
assigning the correct dependency label to arcs becomes harder as the number of dependency labels increases, and as the number of training instances for each dependency label decreases. This holds irrespectively of whether parsing is pseudo-projective or not.

The first column of table 5.5 contains the number of distinct dependency labels for $p_0$. These figures are always the same as before the transformation, since $p_0$ does not extend the set of dependency labels. However, the number of distinct pseudo-projective dependency labels for the other encodings for each treebank is noteworthy. The theoretical upper bound on the number of dependency labels for the two transformations with a quadratic increase ($p_H$ and $p_{HP}$) is much higher than the figures observed in practice. For instance, 643 pseudo-projective dependency labels for PDT with $p_{HP}$ constitute a very large set, but it is far less than 13944, the theoretical upper bound given 83 original dependency labels (cf. table 5.1). Fortunately for the parser, there are a huge number of combinations that for linguistic reasons – or data sparseness – never occur.

### Table 5.5: The number of distinct pseudo-projective dependency labels.

<table>
<thead>
<tr>
<th>Treebank</th>
<th>$p_0$</th>
<th>$p_H$</th>
<th>$p_P$</th>
<th>$p_{HP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDT</td>
<td>27</td>
<td>129</td>
<td>66</td>
<td>149</td>
</tr>
<tr>
<td>PADT</td>
<td>28</td>
<td>82</td>
<td>61</td>
<td>99</td>
</tr>
<tr>
<td>Alpino</td>
<td>26</td>
<td>112</td>
<td>70</td>
<td>148</td>
</tr>
<tr>
<td>Tiger</td>
<td>46</td>
<td>268</td>
<td>117</td>
<td>329</td>
</tr>
<tr>
<td>PDT</td>
<td>83</td>
<td>527</td>
<td>206</td>
<td>643</td>
</tr>
</tbody>
</table>

5.4 Parsing Experiments

The course of action in the experiments was presented at the end of section 5.1. As mentioned, the parser used in steps 2–3 in the list will be MaltParser, and, furthermore, the experiments will be constrained to one set of features per treebank. The experiments will also be constrained to SVM, as it has delivered the best results in previous comparisons of machine learners for dependency parsing, e.g., Hall et al. (2006). Moreover, one single set of features and SVM parameter values have been used for each treebank, optimized for the MaltParser system in the CoNLL shared task 2006 (Nivre et al., 2006). It should be noted that the modifications in the training data, caused by the transformations, are likely to change the optimal combination of feature set and SVM parameter values. This has, however, been an issue of lower priority in the experiments below.

Subsection 5.4.1 presents the overall parsing results, while subsection 5.4.2 investigates precision and recall for non-projectivity in more detail. More parsing experiments investigating how the amount of training data influ-

---

6. The feature sets and SVM parameter values are reported at: http://w3.msi.vxu.se/users/jha/conllx/index.htm.
Table 5.6: Results on development sets, or as a result of cross-validation, for the pseudo-projective transformations using MaltParser.

The question in focus here is first of all whether the pseudo-projective transformations can improve parsing accuracy, whether any difference in accuracy between the various transformation versions can be detected, and whether the effect of the pseudo-projective transformations varies between treebanks and languages.

Table 5.6 presents the overall attachment score results for all five treebanks. The column marked **None** contains the result when the training data sets have not been projectivized and when there is no inverse transformation of the output of the parser. The bold figures mark the best accuracy for each row.

The first thing to note is that the pseudo-projective baseline (p0) – which projectivizes the training data without trying to conduct an inverse transformation – compared to **None** results in the highest improvement for Alpino. There are corresponding, although less prominent, improvements for PADT and Tiger, which can be a result of the higher proportion of non-projectivity in Alpino. However, this is not always the case, as the impact of the baseline...
transformation for SDT and PDT is in fact negative. One can identify two reasons for the results. On the one hand, training a projective parser on non-projective data ought to be less profitable, since the training module may behave unexpectedly. On the other hand, projectivizing the data corrupts the data, i.e., noise is essentially added, which also ought to degrade accuracy.

A comparison between $p_0$ on the one hand, and $p_H$, $p_P$ and $p_{HP}$ on the other, reveals that both $AS_U$ and $AS_L$ for all treebanks increase, except for PADT. The highest improvement is obtained for Alpino (the error reduction for $AS_U$ is 22.7% for $p_{HP}$ compared to $p_0$), which again can be explained by its highest proportion of non-projectivity. Even $AS_L$ for SDT increases compared to None, although $AS_U$ is still lower. The absence of a positive result for PADT is understandable given the characteristics of PADT presented in the treebank transformation experiment (section 5.3), for instance the deeply nested non-projectivity. Nevertheless, the overall results clearly show that the parser can successfully learn to correctly identify some lifted dependency relations and/or the lifted path, and that the inverse transformation can correctly recover non-projectivity to a large extent, yielding an overall improved accuracy.

When comparing $p_H$, $p_P$ and $p_{HP}$ with each other, no clear winner can be proclaimed. If a difference should have emerged, a conceivable hypothesis is that the more training data, the better $p_{HP}$ works, since it is the most informative encoding. However, this cannot be verified, as the largest treebank (PDT) has the highest accuracy for $p_P$, whereas $p_{HP}$ shows the best results for at least Alpino and Tiger. Of course, the differences for all languages are too small to be statistically significant. It is worth noting that $p_H$ had the best accuracy in Nivre and Nilsson (2005), but the difference was again too small to be statistically significant. Another possibility can be that a high proportion of non-projectivity is more beneficial for the most informative encoding. This will remain a hypothesis until further experiments have been conducted.

The pseudo-projective parsing approach also improves accuracy more or less for other languages, such as Danish, Portuguese and Turkish, as reported by Hall and Nilsson (2006). The cross-linguistic validity is further strengthened by Hall et al. (2009), reporting that 6 of 9 treebanks containing non-projectivity benefit from applying the pseudo-projective transformations. The results can also be compared to Nivre (2008), who, among other things, presents figures showing that Nivre’s arc-eager algorithm with $p_H$ performs better than $p_0$ averaged over all 13 treebanks of the CoNLL shared task 2006.

Moreover, it is interesting to compare the figures for standard deviation between the three treebanks for which cross-validation was performed. They confirm that the more available data, the more reliable results. SDT, the smallest treebank, exhibits a large variation in accuracy between the ten parts of the cross-validation, which is reflected in the standard deviation for both $AS_U$ and $AS_L$. As the amount of data grows, the standard deviation
5 Pseudo-Projective Transformations

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$p_H$</td>
<td>$p_P$</td>
</tr>
<tr>
<td>SDT</td>
<td>71.6</td>
<td>71.2</td>
</tr>
<tr>
<td>PADT</td>
<td>17.3</td>
<td>14.2</td>
</tr>
<tr>
<td>Alpino</td>
<td>71.6</td>
<td>74.8</td>
</tr>
<tr>
<td>Tiger</td>
<td>67.4</td>
<td>66.8</td>
</tr>
<tr>
<td>PDT</td>
<td>69.7</td>
<td>61.6</td>
</tr>
</tbody>
</table>

Table 5.7: Unlabeled precision (P) and recall (G) for arcs marked as lifted during pseudo-projective parsing.

decreases, which happens for PADT and Alpino. One can therefore expect that corresponding cross-validation experiments for Tiger and PDT have an even lower standard deviation than Alpino.

Before ending this part, a discussion about the running time for pseudo-projective parsing can be of interest. Both the projectivization and de-projectivization have worse than linear time complexity, entailing that the complete parsing procedure also has worse than linear time complexity. However, while the parsing time for a test set using SVM can take hours for the larger treebanks and minutes for smaller ones, deprojectivizing the same test set takes at most a few seconds. There is also ample room to improve the running time of the deprojectivization implementation. Another property that can make the parsing time slower is that the projectivization increases the number of dependency labels, and consequently the number of distinct transitions. This, in turn, may increase the prediction time of the classifier, but the time complexity of the parser is still linear.

5.4.2 Precision and Recall

In the first detailed analysis the question is: how is accuracy affected for the arcs directly involved in the transformation, the non-projective arcs? Precision and recall for lifted arcs for the parsing experiment of table 5.6 are shown in table 5.7.

Any arc that is identified as lifted (i.e. if the arc label contains ↑) in the parser output is part of the set of arcs used for computing precision. The encodings identify different arcs as lifted. However, because None and $p_0$ do not identify lifted arcs in the parser output, their precision results are undefined and are therefore not shown in the table. This is not a problem for recall, as the set of lifted arcs of the gold-standard trees is identical for all encodings.

The figures for recall reveal that the parser trained on non-projective data (None) actually manages to assign the correct head to quite a high proportion of non-projective tokens, for instance as much as 17% for Tiger. Whether this is a good or bad result for Tiger is actually hard to determine, since it must inevitably mean that these non-projective tokens have (several) erroneous neighboring arcs.
The differences in recall between None and \( p_0 \) are for all treebanks except PADT in favor of the former. This may seem contradictory for SDT, Alpino and Tiger, having higher overall \( AS_U \) and \( AS_L \) for \( p_0 \) than None, according to table 5.6. This yields decreased recall for all non-projective arcs in the test data for \( p_0 \) compared to None. Learning a projective model using non-projective data can be beneficial for non-projective arcs but harmful in general, leading to an overall lower accuracy. By exchanging all non-projective arcs, there are simply no correct non-projective arcs for the parser to learn from. Training on completely projective data, using a projective training module, is on the other hand beneficial for all projective arcs, since the training module is only designed to cope with the projective arcs that constitute the majority of all arcs.

The recall results for the three other encodings exhibit higher values compared to None and \( p_0 \) for all treebanks, again with PADT as the expected exception. For the others, Alpino has the highest increase and SDT the lowest, which correlates quite well with the corresponding figures in table 5.3. This again indicates that the amount of training data, both the total amount of data and the amount of non-projectivity, has an impact on the parser’s ability to conduct pseudo-projective parsing (as well as how deeply nested the non-projective arcs are). Moreover, the differences in recall for the three encodings \( p_0 \), \( p_H \) and \( p_P \) are statistically significant compared to both None and \( p_0 \) for each language, but not when comparing \( p_0 \), \( p_H \), and \( p_P \) with each other (McNemar’s test, \( p < 0.01 \)).

Whereas the recall for the treebanks ranges from less than 35% to above 65%, it is interesting to note that the precision figures are much closer to each other (again excluding PADT). Alpino still has the highest figures, but it seems that precision is less sensitive to the total amount of data and the proportion of non-projectivity. Moreover, all precision figures are slightly higher than the corresponding figures for recall, indicating that each pseudo-projective parser marks fewer arcs as lifted than there are non-projective arcs present in the treebank. Together with the low precision for PADT, this indicates that it is mostly the simpler (shallowly nested) non-projective arcs that are correctly marked as lifted by the parsers. This is then followed by a subsequent simple and successful search for the syntactic heads.

### 5.4.3 Learning Curves

To test the hypothesis that more data gives better parsing results for pseudo-projective transformations, a learning curve experiment has been performed, constrained to Alpino and Tiger. The training set for Alpino was divided into the same ten parts as during the cross-validation, with sets 0 and 9 for testing and sets 1–8 for training. To normalize the values, compensating for the increasingly higher accuracy with more data, the \( AS_U \) results are presented as error reduction in relation to using no transformations (None). Table 5.8 contains all the values for the learning curve experiments, as illustrated by the diagram in figure 5.4.
The dotted curves represent Alpino and the unbroken curves Tiger, where the error reduction for the four encodings $p_0$, $p_H$, $p_P$ and $p_{HP}$ is computed. For both Alpino and Tiger, the error reduction for $p_0$ is unaffected by the amount of data. While the error reduction varies slightly, it turns out that the error reduction is virtually the same between 10% and 80%, which is about 4% and 2.5% for Alpino and Tiger, respectively.

However, for $p_H$, $p_P$ and $p_{HP}$, which actively try to recover non-projectivity, the learning curves clearly indicate that the amount of data matters. Tiger, starting at an error reduction of approximately 5.5%, has an increasing performance, with a more than doubled error reduction with 80% data. Alpino, with 36% non-projective arcs, starts at an error reduction of about 17% and has a climbing curve up to an error reduction of almost 25%.

Although this experiment shows that there is a correlation between the amount of data and the accuracy for pseudo-projective parsing, it probably does not tell the whole story. If it did, one could expect that the error reduction for $p_H$, $p_P$ and $p_{HP}$ would be much closer to $p_0$ when the amount of data is low (to the left in the figure) than they apparently are. Of course, the difference is likely to diminish with even less data, but it should be noted that 10% of Alpino has about half the size of PADT, for which the positive impact of pseudo-projective parsing is absent.

### 5.4.4 Related Work with Other Parsers

The previous experiment shows that an increased accuracy for MaltParser is often the result of the various transformations that have been applied. This section brings up previous studies using other data-driven dependency parsers together with the pseudo-projective transformations.

In the study by Nilsson (2007), the pseudo-projective transformations are applied to MSTParser (McDonald et al., 2005) as well, investigating whether the improvement extends to parsers that are not transition-based. This version of MSTParser is of the first order, that is, it does not condition its decision on any surrounding arcs. MSTparser contains two parsing algorithms, the non-projective Chu-Liu/Edmonds algorithm and the projective Eisner algorithm. The study investigated whether the pseudo-projective transformations can improve accuracy for the projective Eisner algorithm, and how
5.4 Parsing Experiments

Figure 5.4: Learning curves for Alpino and Tiger measured as error reduction for ASU.

high this accuracy is in relation to the non-projective Chu-Liu/Edmonds algorithm (obviously without using the pseudo-projective transformations). In the study, a reduced version of the Dutch data from the CoNLL shared task was used, containing only the word forms and coarse-grained parts-of-speech.

No transformation (None), p0 and pH were applied to the Eisner algorithm. The result is very similar to the result for MaltParser, having a boost in accuracy for p0 compared to None, and with a significantly higher accuracy for pH over p0. pH with the Eisner algorithm has an error reduction of 23.8% compared to p0 for ASU, whereas the corresponding error reduction for MaltParser for Dutch in table 5.6 is 22.7%. For ASL, MaltParser has instead the highest error reduction of 16.3% for pH in relation to p0, when the Eisner algorithm has 10.9% error reduction.

When comparing the Eisner algorithm together with pH to the Chu-Liu/Edmonds algorithm, the study showed that they have comparable accuracies. The Chu-Liu/Edmonds algorithm had better ASL, and slightly lower ASL, than the Eisner algorithm with pH. The study reports that neither difference is statistically significant.

Before ending this section, the study by Nivre (2008) is worth mentioning again, which further strengthens the fact that the pseudo-projective transformations improve accuracy across parsers. He reports improved accuracies for both Nivre’s arc-standard algorithm and the projective version of Covington’s algorithm, when using pH compared to p0, averaged over all languages of the CoNLL shared task 2006. It is worth noting that Coving-
ton’s non-projective algorithm has slightly higher accuracy than arc-eager and arc-standard using $p_H$. The advantage of the two latter algorithms is still their linear time complexities, compared to quadratic time complexity for the former algorithm.

Several other studies do not include a comparative analysis with and without the pseudo-projective transformations, but apply the pseudo-projective transformations presented in this chapter to their projective data-driven dependency parsers. These studies include Johansson and Nugues (2006); Sagae and Tsujii (2007); Titov and Henderson (2007b).

5.5 Conclusion

Given the pseudo-projective transformation, the non-projective structures are not impossible to parse for a projective parser. The experiments show that parsing accuracy can be improved for several languages and treebanks, and for more than one parser. However, there are also languages and treebanks for which no pseudo-projective transformation helps due to properties like deeply nested non-projectivity and small treebanks, such as for PADT and to some extent SDT. Subsection 5.4.4 indicates that the benefit of the pseudo-projective transformations is not restricted to MaltParser, but generalizes to other data-driven dependency parsers as well.

The main outcome of this chapter is that a data-driven dependency parser such as MaltParser can learn to distinguish lifted arcs from arcs that have not been lifted. The experiments also show that the success of the pseudo-projective transformation during parsing is tied to the proportion of non-projectivity, as the error reduction for Alpino is higher than for Tiger and PDT. The learning curve experiments for Alpino and Tiger imply that the increase in accuracy for the pseudo-projective transformation correlates with the total amount of data as well. In other words, the more non-projectivity and data, the better the result. The learning curve experiment together with the precision and recall figures for PADT show that the more complex the non-projective arcs, i.e., the more deeply the arcs are nested, the lower the probability to accurately recover them.
Chapter 6

Coordination and Verb Group Transformations

Similarly to the previous chapter, this chapter is also part of the transformation track and deals with transformations targeting specific linguistic constructions, viz., coordination and verb groups. The ideas presented here come from Nilsson et al. (2006), the study where the coordination and verb group transformations were applied for the first time. As for chapter 5, the experiments in Nilsson et al. (2007) form the basis for this chapter. The choice of linguistic constructions is based on an earlier analysis (Nilsson and Hall, 2005; Nilsson et al., 2005; Nivre et al., 2006), showing that these constructions are more difficult to parse for MaltParser.

The methodology of the tree transformations of this chapter is the same as for the pseudo-projective transformations, that is, combining preprocessing and postprocessing as described in section 5.1. In sections 6.1 and 6.2, the transformations and corresponding inverse transformations for coordination and verb groups, respectively, are presented. Section 6.3 contains the experimental setup, section 6.4 presents the treebank transformation experiments, and section 6.5 the parsing experiments. We end with a conclusion in section 6.6.

6.1 Coordination Transformations

Although dependency structure has many advantages, some dependency-based theories can be criticized for not being expressive enough for some kinds of linguistic phenomena. Coordination and apposition,\(^1\) and possibly also multi-word expressions,\(^2\) pose a problem for the notion of dependency as such. This section will discuss how coordination is handled using dependency structure, especially given definition 2 of a dependency tree.

Coordination is a linguistic phenomenon that is troublesome for most linguistic theories, including dependency-based ones. According to Mel’čuk

\(^{1}\)In *The pope John Paul II passed away*, it is hard to determine whether *The pope* is the head of *John Paul* or vice versa, as either one is optional when the other is present and either one further specifies the other (Fraser’s first and fourth criteria, see chapter 2).

\(^{2}\)E.g., what are the internal relationships between *John*, *Paul*, and *II*?
Coordination and Verb Group Transformations

(1988), the criticism of dependency formalisms can be divided into three major groups:

1. **Double dependency**: a word form can depend simultaneously on two different word forms.
2. **Mutual dependency**: two word forms depend simultaneously on each other.
3. **No dependency**: some constructions have no syntactic head, e.g., coordination where there is no dependency between the conjoined items.

As will be discussed below, points (1) and (3) are relevant for coordination. The criticism, as noted by Mel’čuk, is not restricted to coordination, but applies to several linguistic phenomena. A verb group construction, which will be discussed in detail in section 6.2, is another example.

### 6.1.1 Analysis of Coordination

The first important issue is to determine what is the head word in a coordination construction. For a sentence like

Skilled Swedes and Danes play soccer

there are normally three proposals for *Swedes and Danes* using dependency structure, but all of them have problems:

- One of the individual conjuncts.
- Both the individual conjuncts.
- The conjunction.

The first one is inadequate because one can claim that neither of the nouns have higher priority than the other. Interchanging the nouns (i.e., *skilled Danes and Swedes play soccer*) does not alter the interpretation (if *skilled* modifies both nouns), and moving one of the conjuncts lower down in the hierarchy than the other therefore seems wrong.

Bloomfield (1933) characterizes coordination as an endocentric construction due to the fact that either of the conjuncts can replace the whole coordination. Assigning both conjuncts the same hierarchical priority is supported by the second and third proposals. However, both are problematic whenever the conjuncts have a mutual dependent. The word *skilled* can semantically depend on both *Swedes* and *Danes*, which is incompatible with the definition of a dependency tree, because it would have more than one head. This is in accordance with criticism (1). In contrast to the third, the second proposal keeps the same direct relation between each conjunct and its head which would be present when the coordination is instead occupied by the conjunct alone. This holds for one of the individual conjuncts for the first proposal as well. The conjunction *and* cannot by itself be the subject, like *Swedes* and *Danes*, and is, e.g., not inflected according to the head of the subject (cf. Fraser’s third criterion; see section 2.2), as conjunctions never inflect.
On the other hand, having the conjunction as the head has the advantage that the conjunction semantically acts as a functor and the conjuncts as its arguments (e.g., AND(Conjunct 1, Conjunct 2)). In terms of dependency relations, it is then most natural to regard the functor as the head and its arguments as dependents, which is in line with the third proposal.

It is worth pointing out the complexity that coordination gives rise to for any linguistic theory, but especially for those based on dependency. Few full-fledged dependency-based theories are content with the rather simple definitions of dependency trees, definitions 1 and 2, in subsection 2.2.2, since it is hard to capture several types of important distinctions. One such distinction, semantic scope ambiguity, will be discussed further in connection with the transformations presented in subsection 6.1.2.

Tesnière (1959), who distinguishes between three types of syntactic relations, suggests that coordination internally is in fact not describable using ordinary dependency relations. Both conjuncts have a direct ordinary dependency relation to the same head, but the internal structure is sustained by a special junction-relation. Hudson’s Word Grammar (Hudson, 1990) adopts a similar analysis connecting all conjuncts to the same head. The conjunction plays no part at all in the dependency structure. The whole coordination is in a sense treated as an atomic string or group in the sentence. The leftmost picture in figure 6.1 depicts coordination in accordance with Tesnière and Hudson, denoted Tesnière style (TS), where $C_i$ is a conjunct and $S_j$ a conjunction (or comma). This analysis corresponds to point two in the list in the previous subsection.

Mel’čuk’s MTT (1988) adheres to the first proposal above, where one of the individual conjuncts is in fact the head. He argues that the apparent symmetry of coordination holds only at the semantic level and only for pure logical uses of coordination, which is a minority of all cases. In most cases, the order of the conjuncts cannot be reversed without changing the interpretation:

\[
\text{the Swedes played soccer and won} \neq \text{the Swedes won and played soccer}
\]

In MTT, the leftmost conjunct is the head, which governs the succeeding group with one conjunction and one conjunct. The motivation is that the group can usually be omitted, but not the leftmost conjunct. Mel’čuk further argues that the conjunction is the head of the succeeding conjunct for passive syntactic reasons; it is in a sense the conjunction that opens a slot
for the succeeding conjunct. MTT has a more complex handling of coordination than what is described here, but the core dependency structure for coordination can be constructed using definition 2. The structure is shown in the second picture in figure 6.1 and will be called Mel’čuk style (MS).

The other possibility that follows from the first candidate besides MS is to make the second conjunct depend directly on the first conjunct, while making either the first or second conjunct head of the conjunction. This construction resembles the relations of binary lexical relations in coordination used internally in Collins’ data-driven phrase-structure parser (1999) (see section 2.1). Its dependency structure, where the conjunction depends on the second conjunct, is shown in the third picture in figure 6.1 and is called Collins style (CS).

Whereas MS and CS are syntactically grounded, the rightmost picture in figure 6.1 can be motivated semantically. This analysis is for example adopted by FDG in the analytical layer, which is based on a linguistic tradition known as the Prague school (Sgall et al., 1986) (PS). The PS analysis is also proposed by Nikula (1986), whereas Lombardo and Lesmo (1998) conjecture that MS is more suitable than PS for incremental dependency parsing.

It is also worth noting that the picture is only schematic and hides several details. One such detail is the theories’ ability to distinguish between collective and disjunctive (or distributive) readings of a sentence containing for example two coordinated nouns. This issue is related to the situation discussed above, i.e., whether Skilled modifies both succeeding nouns or only the first. For instance, did the Swedes and Danes play together (collective reading), or did the Swedes play by themselves and the Danes by themselves (disjunctive reading). This insufficiency of dependency structure could be solved in more than way, e.g., by encoding such information by means of the dependency labels, or by using a notion of grouping, which is not too dissimilar to phrases (Mel’čuk, 1988). This discussion is, however, out of the scope of this thesis.

### 6.1.2 Step 1: Transformations

In contrast to the pseudo-projective transformations, the transformations for coordination are more dependent on the annotation of a particular treebank. The coordination experiments will be conducted on four treebanks, three of which adopt PS, Prague Dependency Treebank (PDT) (Hajič, 1998), Slovene Dependency Treebank (SDT) (Džeroski et al., 2006), and Prague Arabic Dependency Treebank (PADT) (Hajič et al., 2004). The dependency version of the Dutch Alpino Treebank (van der Beek et al., 2002) does not in general comply with PS, but treats coordination in a way very similar to PS, that is, the conjunction acts as the head with the conjuncts as dependents.

This subsection will present the transformations and inverse transformations for coordination. It will contain a more detailed presentation than
6.1 Coordination Transformations

Figure 6.2: “The final of the tournament was distinguished by great fighting spirit and unexpected hardness”. Coordination in PS for a Czech sentence from PDT.

Figure 6.3: Coordination in MS for the sentence in figure 6.2.

in Nilsson et al. (2006). The PS-MS transformations were the only ones presented in that paper, since in preliminary experiments it gave the best result compared to PS-CS. Besides, the experiments were performed only for PDT. Later experiments revealed that PS-MS is better than PS-CS for SDT and PADT as well. In this thesis, the PS-MS transformations will therefore only be applied to PDT, SDT and PADT. For Alpino, the PS-CS transformation will also be evaluated in section 6.5, and is accordingly presented below.

The description below is adapted to PDT, SDT and PADT. Since the annotation of coordination in Alpino is similar but not identical, a presentation of the differences in relation to PS ends this subsection.

Classification of Tokens

The transformation begins with the identification of a base conjunction, based on its dependency label and in some cases also its part-of-speech. The base conjunction has the dependency label Coord in PDT, SDT and PADT. For example, the word a (and) in figure 6.2 is identified as a base conjunction. The only tokens of interest in the transformation are the base conjunction and all its dependents. When a base conjunction has been identified, the transformation starts with a classification of the tokens into three linear ordered sets:

- Conjuncts $C_1, \ldots, C_n$
- Separators $S_{1_1}, \ldots, S_{m_{n-1}}$, where $S_{1_1}, \ldots, S_{p_i}$ is located between $C_i$ and $C_{i+1}$.
- Other dependents $D_1, \ldots, D_k$
The base conjunction is categorized as a separator. If the coordination consists of more than two conjuncts, it normally has one or more commas (usually labeled \textit{AuxX}) separating conjuncts, in addition to the base conjunction. They are also categorized as \textit{S}. The coordination in figure 6.2 contains no commas, so only the word \textit{a} will belong to \textit{S}.

The remaining dependents of the base conjunction need to be divided into conjuncts (\textit{C}) and other dependents (\textit{D}). The conjuncts are easily identified in PDT since they have labels suffixed \textunderscore \textit{Co}, which is the case for the words \textit{bojovnosti} and \textit{tvrdosti} in figure 6.2. However, special care must be taken for coordinated prepositional cases (\textit{AuxP}) and embedded clauses (\textit{AuxC}), as the suffix is located on their dependents instead of directly on \textit{AuxP} and \textit{AuxC} (cf. Böhmová et al., 2003).

Unfortunately, SDT and PADT do not add \textunderscore \textit{Co} to the label of conjuncts, making the distinction harder. Heuristic rules have thus been implemented. Although the heuristic rules also consist of a number of back-off conditions, the simple and accurate main rule says that if the base conjunction has dependents on both sides with the same label, they are selected as conjuncts \textit{C}. All other dependents of the base conjunction, not identified as \textit{S} or \textit{C}, are categorized as \textit{D}. Since there are no other dependents of \textit{a} in figure 6.2, \textit{D} is in the example an empty set.

\textbf{Arc Transformations}

Given the classification of the words involved in a coordination, the coordination transformation, which we denote \(\tau_c\) from now on, is straightforward and basically connects all the arcs in a chain:

1. Make \(C_1\) a dependent of the original head of the base conjunction.
2. Make each \(S_j\), located between two adjacent conjuncts \(C_i\) and \(C_{i+1}\) (\(C_i < S_1 < \ldots < S_p < C_{i+1}\)), a dependent of \(C_i\).
3. Make each \(C_{i+1}\) (\(i > 0\)) a dependent of \(S_p\), i.e., the separator closest to its left.
4. Make each \(d \in D\) a dependent of the \(C_i\) closest to its left, or of \(C_1\) if \(d\) is located to the left of \(C_1\).

In the rare cases when there is no separator between the conjuncts, the second conjunct depends on the previous separator, or on the base conjunction, if there is no previous separator. Also, the dependency types of the conjuncts for PDT are truncated by removing the suffix \textunderscore \textit{Co}.\textsuperscript{3} After the transformation \(\tau_c\), every coordination forms a left-headed chain, as illustrated in figure 6.3.

This new representation creates a problem, though. For instance, the word \textit{Velkou} in figure 6.3 is not distinguishable from a possible dependent in \(D\). It could have been a dependent of \textit{a} in figure 6.2, which is an obvious drawback when transforming back to PS. One way of distinguishing \(D\)

\textsuperscript{3}Preliminary experiments indicated that this increases parsing accuracy.
elements is to extend the set of dependency types. The dependency type $r$ of each $d \in D$ can be replaced by a completely new dependency type $r+$ (e.g., $Atr+$), thus theoretically increasing the number of dependency types to $2 \cdot |R|$. Consequently, this avoids a clash between the dependency type of the words in $D$ and dependency types such as $Atr$ for Velkou. We will denote this extended version of the transformation by $\tau_{c+}$.

**Adaptations for Alpino**

As mentioned above, the annotation for coordination in Alpino is similar but not identical to the annotation in PDT, SDT and PADT. Adaptations of the conversions to PS and CS have therefore been implemented. Besides a completely different set of dependency labels, there are three main distinguishing characteristics in Alpino:

1. The label to the head of the base conjunction and the label to the conjuncts are interchanged. In figure 6.2, this would correspond to the situation that the word $a$ has the label $\text{Obj}$ to its head and $\text{bojovností}$ and $\text{tvrdostí}$ the label $\text{Coord}$ ($\text{Coord}=\text{cnj}$ in Alpino).

2. Commas acting as separators are not dependents of the base conjunction. Generally, they are instead dependents of the token closest to its left, which usually is the conjunct or a token dominated by the conjunct.

3. Alpino does not distinguish between coordination dependents and conjunct dependents, i.e. the base conjunction has nothing but conjuncts as dependents.

These differences accordingly affect the classification into $C$, $S$ and $D$. The identification of commas into $S$, especially, is more complicated, since they now are dominated by the tokens classified as the conjuncts in $C$. In order to adapt the transformation with respect to the first distinguishing characteristic, their labels are interchanged before rearranging the arcs. The labels of conjuncts and conjunctions are in thus line with PS, as exemplified by figure 6.2.

To convert coordination into CS, each $C_{i+1}$ is basically made dependent on $C_i$, while all tokens in $S$ are simply left as they are. The exception is the base conjunction itself, which becomes a dependent of the conjunct closest to its left or a token dominated by this conjunct located to the left of the base conjunction. This means that CS is identical to the dependency tree in figure 6.3 with the exception that the head of $\text{tvrdostí}$ is $\text{bojovností}$, not $a$.

**6.1.3 Step 4: Inverse Transformations**

The inverse transformation for PDT, SDT and PADT again starts by identifying base conjunctions, using the same conditions as before. For each identified base conjunction, the inverse transformation calls a procedure
that performs the inverse transformation by traversing the chain of conjuncts and separators “upwards” (right-to-left), while collecting conjuncts (C), separators (S) and potential conjunction dependents (Dpot). When this is done, the former head of the leftmost conjunct (C1) becomes the head of the rightmost (base) conjunction (the rightmost member of S). In figure 6.3, the leftmost conjunct is bojovností, with the head vyznačovalo, and the rightmost (and only) conjunction is a, which will then have vyznačovalo as its new head. All conjuncts in C become dependents of the rightmost conjunction, which means that the structure is converted to PS, as depicted in figure 6.2.

As mentioned above, the original structure in figure 6.2 did not have any coordination dependents, except for Velkou which belongs to Dpot. The last step of the inverse transformation is therefore to sort out conjunction dependents from conjunct dependents, where the former will attach to the base conjunction. Four versions have been implemented, two of which take into account the fact that some dependency types occur more frequently as conjunction dependents (D) than as conjunct dependents in the training data set:

- τc: Do not extend arc labels in τc. Leave all words in Dpot in place.
- τc*: Do not extend arc labels in τc. Attach all words with the label AuxG, AuxX, AuxY or Pred4 to the base conjunction.
- τc+: Extend arc labels from r to r+ for D elements in τc. Attach all words with the label r+ to the base conjunction (and change the label to r).
- τc+*: Extend arc labels from r to r+ for D elements in τc, except for the labels AuxG, AuxX, AuxY and Pred. Attach all words with the label r+, AuxG, AuxX, AuxY, or Pred to the base conjunction (and change the label to r if necessary).

For the particular example at hand, with the word Velkou labeled Atr, all four versions would correctly recognize it as a conjunct dependent and not attach it to the base conjunction.

The inverse transformation for Alpino with CS is similar, with the difference that the separators (dominated by the conjuncts) are collected in another way as the chain only contains conjuncts. Finally, the labels of conjuncts and separators are interchanged in order to restore the original annotation. The label of the rightmost conjunct is chosen as the label of the conjunction (in case the conjuncts have different labels).

The above transformation has been presented schematically. It is worth pointing out that a large number of coordination structures in all four tree-banks are not as exemplary as described above, which in various ways is reflected in the experiments of section 6.5. All transformations handle the prototypical cases with high accuracy, but can be improved for special cases.

---

4For SDT and PADT Pred is excluded here, as well as for τc+*. 

70
6.2 Verb Group Transformation

So, to conclude this section, the more special cases that are covered by the transformations, the better the transformation accuracy. But this comes at the expense of generality, which is a reason why the transformations have to a large extent been kept fairly simple.

6.2 Verb Group Transformation

The relation between the verbs in a clause containing more than one verb is in general less problematic than coordination for linguistic theories. However, the best way to represent it is still not completely settled, neither in constituency-based theories nor in dependency-based ones. For the latter, the second type of criticism discussed in the beginning of section 6.1 – concerning mutual dependency – holds for the verbs involved in a verb group. The problem is also how the verbs relate to various complements and adjuncts in the clause, which is affected by the internal relation between the verbs.

6.2.1 Analysis of Verb Groups

Tesnière (1959) proposes that the relation between the auxiliary verb and the main verb is not a dependency relation, but the verbs instead form a nucleus. All their dependents (e.g., subjects, objects and adverbials) attach to the nucleus and not directly to the word. Although this approach can be attractive for other reasons, it does not conform to definition 2 of a dependency tree. Moreover, in languages such as English and Swedish, it is possible to insert certain complements between the verbs (e.g. did they play soccer?), resulting in discontinuous nuclei.

Verb groups are most naturally described as exocentric constructions in several languages such as English and Swedish, since the presence of a non-finite verb without an auxiliary verb normally yields an ungrammatical sentence, as well as vice versa. However, the internal relation is still open. If it has been settled that there should be a dependency relation between the verbs, there are essentially only two possibilities. First, the main verb can be the dependent of the auxiliary verb, where the latter consequently is the head of the whole clause. This analysis is advocated by Mel’čuk (1988), mainly for syntactic reasons. Most grammatically correct sentences have a finite verb whereas a non-finite verb is optional, which according to Fraser’s first criterion (section 2.2) strengthens this point of view. The third criterion also favors this analysis, since the possible presence of an auxiliary verb in English controls the inflection of the main verb. The left picture of figure 6.4 illustrates this dependency analysis, denoted MS. Mel’čuk further suggests that all complements and adjuncts to the left and to the right attach to the auxiliary verb and the main verb, respectively, but other sound solutions exist. Note that if there are multiple auxiliary verbs, then there are more possibilities.
The right picture in the same figure shows the second possibility, which instead has the main verb as the head of the clause with a direct governing relation to the auxiliary verb. This analysis is more easily motivated by semantic criteria, where the auxiliary verb can be regarded as a modifier of the main verb, somewhat in line with Fraser’s fourth criterion. Moreover, it is the main verb that determines the number of complements through its property of carrying valency. It subcategorizes and is in this perspective therefore in control of all complements, in compliance with the second criterion of Fraser. This approach is adopted by for instance FGD, according to the Prague school (PS).

It is worth noting that FGD does not admit auxiliary verbs to have dependents at all, although some linguists point out that auxiliary verbs can have their own adjuncts. Another option in PS is to make the subject (usually located to the left) depend on the auxiliary verb. This solution is one of many suggestions by Nikula (1986), even though he mainly supports MS.

As a main verb can be accompanied by more than one auxiliary verb, one may ask how they in turn relate to each other and to the main verb. The most common solution is to form a chain of arcs between the verbs when an auxiliary verb is the head (AuxV1 → AuxV2, AuxV2 → MainV), and the main verb is a direct head of all auxiliary verbs in PS (MainV → AuxV1, MainV → AuxV2).

It is again worth noting that Collins’ (1997) head-finding strategy for constructing the bilexical dependencies for his parser, applied to the Penn Treebank, regards the modal or auxiliary verb as the head and the main verb as dependent in verb groups. This is in line with MS.

### 6.2.2 Transformation and Inverse Transformation

PDT and SDT annotate verb groups according to PS, and they are therefore the two treebanks in the following experiments for which the verb group transformations have been performed. In comparison with the coordination transformation and inverse transformations, the corresponding transformations for verb groups are strikingly simple. The transformation algorithm, \( \tau_v \), starts by identifying all auxiliary verbs in a sentence. For two or more auxiliary verbs having the same head (i.e., the main verb), all but the leftmost are removed. The remaining auxiliary verbs will belong to the set \( A \).

A word \( a \in A \) iff \( m \xrightarrow{AuxV} a \), where \( m \) is the main verb. For example, the
6.2 Verb Group Transformation

Figure 6.5: “Her preparations will take several months”. Verb group in PS for a Czech sentence from PDT.

Figure 6.6: Verb group in MS for the sentence in figure 6.5.

word *bude* (will) in the Czech sentence of figure 6.5 will belong to $A$. Then, for each $a \in A$ processed from left to right in each sentence:

1. Replace the arc $m \rightarrow a$ with $h \rightarrow a$, where $h \rightarrow m$.
2. Replace the arc $h \rightarrow m$ with $a \rightarrow m$.
3. **For each** arc $m \rightarrow d$, $d < \text{left}(a, m)$ do:
   replace $m \rightarrow d$ with $\text{right}(a, m) \rightarrow d$.
4. **For each** arc $m \rightarrow d$, $d > \text{right}(a, m)$ do:
   replace $m \rightarrow d$ with $\text{right}(a, m) \rightarrow d$.

Figure 6.6 depicts the dependency tree of the same Czech sentence in MS. Lines 1 and 2 are not complicated. The transformation into MS reverses the relation between the verbs, i.e., $a \rightarrow m$, and the former head of $m$ becomes the new head of $a$.

The main verb can be located on either side of the auxiliary verb and can have other dependents, whereas auxiliary verbs never have dependents. Line (3) states that all common dependents of the two verbs located to the left of the rightmost verb becomes dependents of the left verb after the transformation. The only such arc in the example is the subject *příprava*, which becomes a dependent of the auxiliary verb after the transformation, i.e. the left-most verb (left($a, m$)). Line (4) takes care of all other dependents located to the right of the right-most verb (such as *několik*), which will then depend on the right-most verb (right($a, m$)).
Table 6.1: Information about coordination and verb groups in four of the data sets of CoNLL shared task 2006, where \( \#T(k) = 1000 \times \text{number of tokens} \), \( \#S(k) = 1000 \times \text{number of sentences} \), \( \%C = \text{Percentage of conjuncts} \), \( \%S = \text{Percentage of separators} \), \( \%VG = \text{Percentage of auxiliary verbs} \).

<table>
<thead>
<tr>
<th>Language (Code)</th>
<th>( #T(k) )</th>
<th>( #S(k) )</th>
<th>%C</th>
<th>%S</th>
<th>%VG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slovene (SDT)</td>
<td>29</td>
<td>1.5</td>
<td>9.3</td>
<td>5.1</td>
<td>8.8</td>
</tr>
<tr>
<td>Arabic (PADT)</td>
<td>54</td>
<td>1.5</td>
<td>8.5</td>
<td>4.6</td>
<td>-</td>
</tr>
<tr>
<td>Dutch (Alpino)</td>
<td>195</td>
<td>13.3</td>
<td>4.0</td>
<td>2.2</td>
<td>-</td>
</tr>
<tr>
<td>Czech (PDT)</td>
<td>1249</td>
<td>72.7</td>
<td>8.5</td>
<td>4.4</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Without lines (3) and (4), the arcs may cause non-projectivity, which happens in the case where the main verb is the left-most verb having dependents to the right of the auxiliary verb (that is \( m < a < d \), where \( m \rightarrow d \)). The arc \( m \rightarrow d \) would be non-projective in MS. Note that, in case a main verb has multiple auxiliary verbs in PS, the algorithm treats all but the leftmost auxiliary verb just like any other dependent.

Performing the inverse transformation for verb groups is quite simple. Each sentence is traversed from right to left in the search for pairs of arcs of the type \( h \xrightarrow{\text{AuxV}} a \) and \( a \xrightarrow{\text{Pred}} m \). For each such pair of arcs, \( h \) will be the new head of \( m \), and \( m \) the new head of \( a \). Furthermore, since auxiliary verbs do not have dependents in PS, all dependents of \( a \) in MS will become dependents of \( m \) in PS. Multiple auxiliary verbs, being dependents of the leftmost verb in MS, are treated as ordinary dependents, and are thus also transferred to \( m \) as dependents.

6.3 Experimental Setup

The experimental setup for the tree transformation and parsing experiments is exactly the same as in chapter 5, described in section 5.2. That is, the same parser version of MaltParser has been used as well as the same data sets. The discussion below is focused on properties of the data sets that are of interest for the coordination and for the verb group experiments.

Table 6.1 contains information about the number of conjuncts and separators (as described in section 6.1) and the amount of auxiliary verbs (as described in section 6.2) for the four treebanks used in experiments. The table shows that coordination is more common than verb groups in PDT. Only 1.3% of the tokens in the training data are identified as auxiliary verbs, whereas 12.9% of the tokens (identified as conjuncts and separators) are involved in coordination. Judging from table 6.1, it is likely that transformations for coordination will have a greater impact on the accuracy than the corresponding transformations for verb groups for PDT. For instance, the amount of conjuncts is 8.5%, while there are only 1.3% auxiliary verbs.

The proportions of conjuncts and separators in SDT are in fact quite similar to the proportions in PDT. The big difference is the proportion of
Table 6.2: Distortion results for coordination (left) and verb groups (right), with figures for ASU.

<table>
<thead>
<tr>
<th></th>
<th>τc</th>
<th>τc*</th>
<th>τc+</th>
<th>τc++</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDT</td>
<td>97.24</td>
<td>98.03</td>
<td>99.10</td>
<td>98.89</td>
</tr>
<tr>
<td>PADT</td>
<td>97.25</td>
<td>97.61</td>
<td>98.23</td>
<td>98.13</td>
</tr>
<tr>
<td>Alpino (MS)</td>
<td>99.40</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Alpino (CS)</td>
<td>99.71</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PDT</td>
<td>97.77</td>
<td>98.50</td>
<td>99.23</td>
<td>99.26</td>
</tr>
</tbody>
</table>

auxiliary verbs, with many more auxiliary verbs in SDT than in PDT. It is therefore plausible that the transformations for verb groups in SDT affect the accuracy more.

Arabic is not a Slavic language, like Czech and Slovene, and the annotation for PADT has been altered more than for SDT. One such example is that Arabic does not have auxiliary verbs, which results in the absence of the dependency label AuxV. Consequently, table 6.1 does not present the number of verb groups. The amount of coordination is on the other hand comparable to both PDT and SDT.

Moreover, the table also shows that coordination is less frequent in the CoNLL version of Alpino than in PDT, SDT and PADT, which probably will lower the effect of the coordination transformations for Alpino.

6.4 Treebank Transformations

In this section, the coordination and verb group transformations are applied to a treebank and then the inverse transformations are applied on the output, in order to investigate how much distortion the transformations cause. In other words, the section corresponds to section 5.4 for the pseudo-projective transformations, showing that the pseudo-projective transformations tend to be quite free from distortion.

The corresponding overall ASU figures for the coordination and verb group transformations are presented in table 6.2. As mentioned in sub-section 6.1.2, preliminary experiments indicated that the PS-to-CS transformation was less successful compared to the PS-to-MS transformation for PDT, SDT and PADT, which is the reason why only results for the latter are presented in the table and throughout the coming parsing experiment in section 6.5. Results for both PS-to-MS and PS-to-CS will, however, be presented for Alpino.

A comparison between the four versions of the coordination transformations reveals that the simplest version (τc), which does not extend the set of dependency types nor uses predefined lifts for coordination dependents, has the lowest ASU by a quite large margin.\(^5\) In other words, a substantial

\(^5\) Alpino does not distinguish between coordination dependents and conjunct dependents (cf. section 6.1.2), making all except τc unnecessary.
error reduction is obtained in this experiment by the two types of modifications designed to deal with dependents of coordinations and conjuncts. For instance, extending the set of dependency labels for PDT results in 69% fewer errors, 67% for SDT, and 35% for PADT. Whether this has a positive impact during parsing is investigated in section 6.5.

The figures also indicate that the less sophisticated solution with the predefined lifts ($\tau_{c^*}$) is less accurate compared to extending the set of dependency labels. It does, however, have the advantage that no additional burden is laid on the parser to make the distinction between dependents of coordination and conjuncts. Its influence during parsing is also investigated in section 6.5. Moreover, on the basis of the coordination figures, these observations can be made as well:

1. **CS** introduces less distortion than **MS** for Alpino,
2. which, in turn, introduces less distortion than for PDT,
3. which, in turn, introduces less distortion than for SDT and PADT.

The transformations for **CS** and back are less complicated than for **MS**, due to the annotation of commas in coordination in Alpino. As commas are not part of the actual coordination, their status is relatively similar in **CS** and **MS**. Observation (2) can probably be attributed to the fact that coordination is annotated in a simpler fashion for Alpino than in the three other treebanks using the Prague school annotation. Empirical observations during the implementation of the transformations indicated that the number of special cases is smaller in Alpino than in the others. A possible explanation can be that Alpino has been converted to a dependency structure. This may imply that different coordination types with different structures in the original data receive the same structure in the converted data, which probably also induces a linguistically less sound annotation for coordination. The other treebanks, on the other hand, were designed to be dependency treebanks from the beginning, which eliminates this risk. A plausible explanation for (3) is that the coordination transformations were originally designed for PDT, making them more biased to the intrinsic properties of coordination in PDT.

As mentioned in section 6.1.2, conjuncts in SDT and PADT are not explicitly marked with the suffix _Co_, as in PDT. To investigate the expected increased distortion due to this, a simple experiment was conducted on PDT using $\tau_c$. The same heuristic rules for identifying the conjuncts in SDT and PADT were applied to PDT after removing all occurrences of _Co_. The accuracy for this transformation was 97.69%, which is very close to the corresponding figure of 97.77% in the table, using the suffix information. This indicates that the heuristic rules are relatively accurate.

The left table reveals that the distortion of the verb group transformations for SDT and PDT is very close to negligible, especially for PDT. The complexity is less and the number of special cases for this transformation is much lower compared to the coordination transformations, which can explain the difference in accuracy. The difference between the two treebanks
can probably again be attributed to the fact that the transformations are slightly biased towards PDT, which they originally were designed for. Finally, the results for PDT here are in line with the corresponding results in Nilsson et al. (2006), for both coordination and verb groups, even though the data set is not exactly the same.

6.5 Parsing Experiments

In this section, we first evaluate the parsing accuracy obtained with Malt-Parser using the coordination and verb group transformations (in subsection 6.5.1). We then combine the coordination and verb group transformations with the pseudo-projective transformations (subsection 6.5.2). Subsection 6.5.3 studies precision and recall for the words directly involved in the transformations. The section continues with a discussion of the characteristics that may explain their influence on parsing accuracy (in subsection 6.5.4), followed by applying the error propagation metrics of chapter 4 (in subsection 6.5.5). We end with a discussion about previous studies using these transformations for other parsing algorithms in subsection 6.5.6.

6.5.1 Applying the Transformations

As the results of the parsing experiment in section 5.4 showed, the pseudo-projective transformations can be beneficial for improving parsing accuracy. The question that will be answered here is whether the transformations for coordination and verb groups, can be beneficial for parsing accuracy as well. Table 6.3 contains the overall results of the experiments, where the None column has the same values as in table 5.6.

The figures indicate that the improvement in accuracy in Nilsson et al. (2006) is confirmed. All versions of the coordination transformations from PS to MS and back during parsing increase accuracy for PDT. The table also shows that the improvement is not restricted to PDT, but holds for the other four treebanks as well. Since SDT, PADT and PDT have comparable proportions of coordination and since the coordination transformations for PDT are more accurate than for SDT and PADT (table 6.2), it is not very surprising that PDT exhibits the highest error reduction of 15.0% for ASU, compared to 9.1% for SDT and 10.0% for PADT. The error reduction result for ASL is in fact the lowest for PDT, but this could be attributed to the fact that PDT has the suffix _Co on the conjuncts’ dependency labels. The inverse transformation has to augment the dependency labels with the suffix, which is not a trivial issue (although linguistically sounder than without the suffix).

The same coordination transformation for Alpino, Alpino (MS), results in a higher accuracy as well, although it is less prominent than for the others (2.4% error reduction for ASU). This is quite expected, since the proportion of coordination is lower than for the others. However, it is interesting to note that the CS transformation for Alpino yields an even higher accuracy
### Table 6.3: Parsing results of development sets for coordination and verb group transformations using MaltParser, excluding punctuation.

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>$\tau_c$</th>
<th>$\tau_{c+}$</th>
<th>$\tau_{c^*}$</th>
<th>$\tau_{c^*+}$</th>
<th>$\tau_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDT</td>
<td>77.31</td>
<td><strong>79.33</strong></td>
<td>79.24</td>
<td>79.05</td>
<td>78.99</td>
<td><strong>77.92</strong></td>
</tr>
<tr>
<td>$\sigma_{AS_U}$</td>
<td>0.42</td>
<td>0.31</td>
<td>0.33</td>
<td>0.32</td>
<td>0.32</td>
<td>0.49</td>
</tr>
<tr>
<td>ASL</td>
<td>68.76</td>
<td><strong>70.45</strong></td>
<td>70.26</td>
<td>70.18</td>
<td>70.10</td>
<td><strong>69.36</strong></td>
</tr>
<tr>
<td>$\sigma_{AS_L}$</td>
<td>0.50</td>
<td>0.38</td>
<td>0.39</td>
<td>0.39</td>
<td>0.39</td>
<td>0.46</td>
</tr>
<tr>
<td>PADT</td>
<td>76.96</td>
<td>78.81</td>
<td>78.70</td>
<td><strong>79.05</strong></td>
<td>78.73</td>
<td>-</td>
</tr>
<tr>
<td>$\sigma_{AS_U}$</td>
<td>0.15</td>
<td>0.24</td>
<td>0.23</td>
<td>0.25</td>
<td>0.29</td>
<td>-</td>
</tr>
<tr>
<td>ASL</td>
<td>65.78</td>
<td>67.33</td>
<td>67.23</td>
<td><strong>67.61</strong></td>
<td>67.39</td>
<td>-</td>
</tr>
<tr>
<td>$\sigma_{AS_L}$</td>
<td>0.27</td>
<td>0.32</td>
<td>0.30</td>
<td>0.34</td>
<td>0.38</td>
<td>-</td>
</tr>
<tr>
<td>Alpino (MS)</td>
<td>82.75</td>
<td><strong>83.15</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\sigma_{AS_U}$</td>
<td>0.09</td>
<td>0.11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ASL</td>
<td>79.87</td>
<td><strong>80.21</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\sigma_{AS_L}$</td>
<td>0.10</td>
<td>0.11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Alpino (CS)</td>
<td>82.75</td>
<td><strong>83.38</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\sigma_{AS_U}$</td>
<td>0.09</td>
<td>0.14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ASL</td>
<td>79.87</td>
<td><strong>80.49</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\sigma_{AS_L}$</td>
<td>0.10</td>
<td>0.13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PDT</td>
<td>83.41</td>
<td>85.38</td>
<td><strong>85.51</strong></td>
<td>85.34</td>
<td>85.46</td>
<td><strong>83.58</strong></td>
</tr>
<tr>
<td>ASL</td>
<td>76.98</td>
<td>77.15</td>
<td><strong>77.22</strong></td>
<td>77.12</td>
<td>77.18</td>
<td><strong>77.11</strong></td>
</tr>
</tbody>
</table>

than **MS**. It has a 3.8% error reduction for $AS_U$, which is comparable to SDT and PADT, as Alpino has slightly less than half the proportion of coordination.

A comparison between the extended ($\tau_{c+}, \tau_{c^*+}$) and unextended ($\tau_c, \tau_{c^*}$) versions gives no clear answers. The extended ones have slightly lower accuracy than their corresponding unextended ones for SDT and PADT, whereas the opposite holds for PDT. This may indicate that the amount of training data may be of importance for the extended versions, but the differences in accuracy are too small to say anything conclusive.

A conclusive answer is also hard to give when comparing the versions with predefined lifts ($\tau_{c^*}, \tau_{c^*+}$) with those without ($\tau_c, \tau_{c+}$). They are beneficial for PADT but not for the others. The differences are in all cases relatively small, where the accuracy with predefined lifts is compared with the corresponding accuracy without.

The positive influence on accuracy holds for the verb group transformation, too, even though it gives less improvement compared to the coordination transformation. The results confirm the boosted accuracy for PDT in Nilsson et al. (2006). The error reduction is actually higher for SDT than for PDT (2.9% and 1.0% for $AS_U$, respectively), despite the fact that the transformation originally was originally designed for PDT, and thus distorts the PDT data less. But the result is still not surprising, since SDT contains more than six times as many arcs in verb groups (table 6.1).

A comparison between the coordination and verb group transformations indicates that the gain in overall error reduction is higher for coordination.
6.5 Parsing Experiments

<table>
<thead>
<tr>
<th>Trans.</th>
<th>Dev</th>
<th>Eval</th>
<th>Niv</th>
<th>McD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDT ( \tau_v \circ \tau_c ) ( AS_U )</td>
<td>80.40</td>
<td>82.01</td>
<td>78.72</td>
<td>83.17</td>
</tr>
<tr>
<td>SDT ( \tau_v \circ \tau_c ) ( AS_L )</td>
<td>71.06</td>
<td>72.44</td>
<td>70.30</td>
<td>73.44</td>
</tr>
<tr>
<td>PADT ( \tau_{c^*} \circ p_0 ) ( AS_U )</td>
<td>78.97</td>
<td>78.56</td>
<td>77.52</td>
<td>79.34</td>
</tr>
<tr>
<td>PADT ( \tau_{c^*} \circ p_0 ) ( AS_L )</td>
<td>67.63</td>
<td>67.58</td>
<td>66.71</td>
<td>66.91</td>
</tr>
<tr>
<td>Alpino (CS) ( \tau_c \circ p_{HP} ) ( AS_U )</td>
<td>87.63</td>
<td>82.85</td>
<td>81.35</td>
<td>83.57</td>
</tr>
<tr>
<td>Alpino (CS) ( \tau_c \circ p_{HP} ) ( AS_L )</td>
<td>84.02</td>
<td>79.73</td>
<td>78.59</td>
<td>79.19</td>
</tr>
<tr>
<td>Tiger ( p_{HP} ) ( AS_U )</td>
<td>89.74</td>
<td>89.08</td>
<td>88.76</td>
<td>90.38</td>
</tr>
<tr>
<td>Tiger ( p_{HP} ) ( AS_L )</td>
<td>86.97</td>
<td>86.20</td>
<td>85.82</td>
<td>87.34</td>
</tr>
<tr>
<td>PDT ( \tau_v \circ \tau_{c^+} \circ pp ) ( AS_U )</td>
<td>85.72</td>
<td>85.98</td>
<td>84.80</td>
<td>87.30</td>
</tr>
<tr>
<td>PDT ( \tau_v \circ \tau_{c^+} \circ pp ) ( AS_L )</td>
<td>78.56</td>
<td>78.80</td>
<td>78.42</td>
<td>80.18</td>
</tr>
</tbody>
</table>

Table 6.4: Parsing results when combining transformations, where Dev = development test, Eval = CoNLL test set., Niv = results of Nivre et al. (2006), McD = the best parser in the CoNLL shared task 2006 for all five treebanks, for both AS_L and AS_U.

than for verb groups, even if the figures are normalized according to frequency. Furthermore, since the linguistic arguments for coordination and verb groups are easier to motivate syntactically in \textit{MS} than in \textit{PS} (see sections 6.1 and 6.2), the experiments indicate that data-driven syntactic parsing should be based on syntactic grounds, at least for MaltParser.

### 6.5.2 Combining Transformations

The best performing transformation of each type for each language will here be combined in order to check whether the transformations give a cumulative improvement. The best figures for each type of transformation are boldfaced in tables 5.6 and 6.3. The outcome of this is shown in table 6.4. The experiment was performed on the development sets (Dev), as well as on the final test sets in the CoNLL shared task 2006 (Eval) for each language. The next column (Niv) contains the results reported in Nivre et al. (2006) for the same held out test sets as in the Eval-column, which all used only the pseudo-projective transformation \( p_{H-P} \). The last column (McD) shows the best individual results reported in the CoNLL shared task, which for all treebanks, for both AS_L and AS_U, is MSTParser (the same parser as the one discussed in chapter 4).

AS_U and AS_L in the Dev column for SDT are higher than any individual AS_U and AS_L in tables 6.3 and table 5.3 for SDT. The same is true for PDT and Alpino. The transformation for Tiger in table 6.4 in the Dev column is the same as in table 5.3, and consequently also the result. For PADT, Alpino and PDT, the sum of the increases of the individually applied transformations is slightly lower than the increase of the combination. However, it is in fact the other way around for SDT, where the accuracy increases by 3.09 percentage points when the coordination and verb group transformations are combined, and only 2.63 percentage points when summing the increases
of the individually applied coordination and verb group transformations for SDT. This indicates quite clearly that these types of transformation are not mutually harmful, but can instead be combined with an accumulated positive influence.

The final test using the held out test sets of the shared task further strengthens all previously conducted experiments during the development phase. AS$_U$ and AS$_L$ for all treebanks are higher in the Eval-column than in the Niv-column. Looking at AS$_L$, the official scoring metric of the CoNLL shared task 2006, we see that the combined effect of the three transformations boosts the performance of MaltParser for all treebanks and in two cases out of four outperforms McD (MSTParser). The total sum of the improvement is +4.91 percentage points, resulting in an average improvement for all languages from 80.19 to 80.60 for AS$_L$.

Just as in chapter 5, a discussion about the time it takes to apply the transformations is of practical interest, especially the time it takes to perform the inverse transformations. In line with the pseudo-projective transformations, all transformations here have worse than linear time complexity. As the parsing time for a test set using SVM can take hours for the larger treebanks and minutes for the smaller ones, and running the inverse transformations for coordination and verb groups takes a few seconds on the same test set, the increased running time is normally negligible. In contrast to the pseudo-projective transformations, these transformations can be applied without increasing the number of dependency labels, and consequently the number of distinct transitions. The prediction time of the classifier, if it is dependent on the number of possible transitions, will thus not increase.

The main goal of the parsing experiments is to investigate whether the transformations can improve parsing accuracy. The above results confirm that this goal has been reached for MaltParser. The next subsections study the above results more in depth.

### 6.5.3 Precision and Recall

Precision and recall for conjuncts and auxiliary verbs are presented in table 6.5. The relationship between precision and recall is that precision is higher than recall. This is in fact the same relationship between precision and recall as for non-projective arcs in table 5.7. The number of tokens marked as conjuncts and auxiliary verbs by the parsers is lower than the actual number of conjuncts and auxiliary verbs. As for non-projectivity, this again may indicate that the parsers can handle the “simpler” cases better than those without transformations, while they still fail to find the more complicated ones. This is, however, less clear for verb groups, at least for SDT, since its precision and recall are virtually unaffected by the transformation. The lower or absent increase in precision and recall for auxiliary verbs must mean that the verb group transformation is more beneficial for the surrounding dependency structure.

Although the increase in accuracy is lower for conjuncts than for non-
Table 6.5: Precision and recall for conjuncts $C$, and auxiliary verbs $A$, where None means that no transformation has been applied, and Tr that the transformation shown for each treebank is applied. The symbol * marks a statistically significant difference between None and Tr (McNemar’s test, $p < 0.01$).

6.5.4 Why do these Transformations Help?

The pseudo-projective transformations increase parsing accuracy because they enable the projective MaltParser to correctly parse all arcs, not just the projective ones, and the same positive effect has also been observed for other parsers. But why do the transformations for coordination and verb groups help? The list below contains three possible explanations:

- The average arc length is reduced.
- The average branching factor is reduced.
- The consistency is increased.

Increased consistency is hard to define and measure. It can be exemplified for coordination, for instance, by comparing the dependency structures for one sentence where the predicate has two coordinated subjects, with another one where the second subject and the conjunction are absent. Such a situation is illustrated in figure 6.7. In the upper left dependency tree for PS, the first subject (Sb) attaches to the conjunction, but the same subtree attaches to the predicate (Pred) in the lower dependency tree. On the other hand, for both MS (upper right) and CS, the first subject will always be the only token with a direct dependency relation to the predicate. This increases consistency in a sense, since the presence of a second subject (in this case and Danes) does not modify the existing dependency structure.

Compared to the third possible explanation, the first two are easy to measure. There is also a correlation between one and two, which will be discussed briefly at the end of this subsection. The focus here will be on the average arc length, even though the third possible explanation probably is
Figure 6.7: Example of consistency for the coordination transformation, where the dependency tree in the upper left is PS, the upper right is MS, and the lower is both PS and MS (and CS).

an important factor as well. The diagram of figure 6.8 illustrates how ASU precision and recall vary as the arc length increases for PDT with no transformation, with verb group transformation and with the best coordination transformations, where the x-value “0” represents all arcs attaching to the root. A similar pattern holds for the other treebanks, too.

The most important observation here is that both precision and recall drop with increasing arc length. For the curves for untransformed training data, it is in fact only the precision and recall for arcs of length 1-3 that are above the ASU of 78.97%. One can also observe that precision and recall are virtually unaffected by the coordination and verb group transformations, as the curves are very similar. In other words, long relations are roughly equally hard to parse irrespectively of whether the treebanks have been transformed or not.

This property may imply that if one can decrease the average arc length in the training data, the learning task becomes easier. In principle, the average arc length can be reduced to 1 by forming a chain where each arc attaches two neighboring tokens. The parsing task would then be trivial, but the inverse transformation extremely difficult.

The curves in figure 6.9 plot the proportion of arcs in the transformed training data of PDT compared to the untransformed training data for coordination and verb groups, divided into the same groups of arc lengths as in figure 6.8. This illustrates that the proportion of long arcs has decreased for both coordination and verb groups, whereas arcs of length 1 have increased. This in turn entails that the average arc length is lower for both the coordination transformation and the verb group transformation. In figures 6.8 and 6.9, the average arc length is 2.3 (100.0%) for the untransformed training data, 2.12 (92.2% of 2.3) for the coordination transformation, and 2.25 (98.0% of 2.3) for the verb group transformation. The trend is most prominent for coordination, which can be partly explained by the fact that PDT has more instances of coordination than verb groups, and may indicate that the average arc length has an impact on the difficulty of the learning task. In other words, the transformations increase the proportion of shorter arcs,
6.5 Parsing Experiments

Figure 6.8: Unlabeled precision (unbroken lines) and recall (dotted lines) per arc length for PDT with no (□), verb group (△) and coordination (○) transformations for PDT.

which according to figure 6.8 are easier to parse.

Before ending this subsection, it is worth noting that the arc length correlates with the branching factor, at least in any sensible dependency tree. This was discussed for coordination and verb groups in sections 6.1 and 6.2, where the lower branching factor of MS and CS compared to TS and PS was emphasized. A low branching factor could be of importance for MaltParser, as it is hard (although possible) to represent more than one child on each side of the topmost stack token in the feature set. However, it cannot be excluded that the increased consistency is the most important explanation, and that the other possible explanations will follow from this.

6.5.5 Error Propagation

The previous subsection indicates a correlation between accuracy and arc length for MaltParser. This subsection continues with an error propagation analysis of the coordination and verb group transformations by applying the same metrics as in chapter 4, i.e., \( \text{Pre} \) and \( \text{Post} \), and the error clustering. Could it be that the reason for the increased accuracy is that the transformations reduce the problem of error propagation? Table 6.6 contains the \( \text{Pre} \) and \( \text{Post} \) figures for all languages for the best transformations of each type, that is, the same transformations as in table 6.5.

First, all parsers exhibit the same behavior as in chapter 4, having higher \( \text{Pre} \) than \( \text{Post} \). Looking at the general impact of the coordination transformation, both \( \text{Pre} \) and \( \text{Post} \) are higher with the coordination transformation than without. The increase in percentage points is on average for all four treebanks in fact higher for \( \text{Pre} \) than \( \text{Post} \), (rounded to) exactly 1.00 percentage points for the former and 1.31 percentage points for the latter.
However, since the accuracies for Pre is higher than for Post, the error reduction figures in the same table constitute a fairer comparison. Here we clearly see that the highest error reduction takes place for Pre across all treebanks, where e.g., SDT has 10.85% error reduction for Pre but just 6.58% for Post. In other words, the coordination transformations do not reduce error propagation, but rather the contrary.

A plausible explanation for this result is that the coordination transformations are able to postpone the first error longer than without the coordination transformation. But when the first error occurs during parsing, the subsequent dependency structure will be less similar to what the inverse transformation is designed to deal with. It is therefore more likely to mess up the structure more than to help, and Post should suffer more from this than Pre.

The corresponding figures for the verb group transformation are higher for both Pre and Post compared to None for SDT and PDT, though slightly lower than for the coordination transformation. In contrast to both the coordination transformation, no clear conclusion can be drawn when studying the error reduction figures. The error reduction is generally low and the highest error reduction figure differs, as Post is higher for SDT and Pre for PDT.

The inconclusiveness holds when comparing the curves for the verb group transformation with the curves without any transformation for SDT (upper left) and PDT (lower right) in figure 6.10, containing the normalized error clustering. No clear trend can be seen, which may be partly attributed to the fact that the verb group transformations are less prone to distort the dependency structure.

The corresponding comparison for the coordination transformation is slightly more informative, at least for the three treebanks with the highest proportion of coordination (SDT, PADT and PDT). The proportion of...
6.5 Parsing Experiments

<table>
<thead>
<tr>
<th></th>
<th>Pre None</th>
<th>Coord</th>
<th>VG</th>
<th>Post None</th>
<th>Coord</th>
<th>VG</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDT</td>
<td>86.08</td>
<td>87.59</td>
<td>86.13</td>
<td>72.79</td>
<td>74.58</td>
<td>73.78</td>
</tr>
<tr>
<td>PADT</td>
<td>86.61</td>
<td>87.93</td>
<td>-</td>
<td>74.65</td>
<td>76.66</td>
<td>-</td>
</tr>
<tr>
<td>Alpino</td>
<td>89.25</td>
<td>89.63</td>
<td>-</td>
<td>77.32</td>
<td>77.91</td>
<td>-</td>
</tr>
<tr>
<td>PDT</td>
<td>89.10</td>
<td>89.85</td>
<td>89.24</td>
<td>79.50</td>
<td>80.31</td>
<td>79.56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Pre Coord</th>
<th>VG</th>
<th>Post Coord</th>
<th>VG</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDT</td>
<td>10.85</td>
<td>0.36</td>
<td>6.58</td>
<td>3.64</td>
</tr>
<tr>
<td>PADT</td>
<td>9.86</td>
<td>-</td>
<td>7.93</td>
<td>-</td>
</tr>
<tr>
<td>Alpino</td>
<td>3.53</td>
<td>-</td>
<td>2.60</td>
<td>-</td>
</tr>
<tr>
<td>PDT</td>
<td>7.22</td>
<td>1.65</td>
<td>4.14</td>
<td>0.49</td>
</tr>
</tbody>
</table>

Table 6.6: Normalized ASU for Pre and Post for SDT, PADT, PDT (above), and corresponding error reduction figures (below) for the same transformations evaluated in table 6.5.

errors in bin 1 (i.e., one token after an error), the most important bin in this perspective, is higher for the coordination transformation, and lasts until at least bin 3 for SDT, PADT and PDT. At the other end of the diagram (bins > 7) for these three treebanks, the coordination transformation curve is generally located below the curve with any transformation. This tendency is especially clear for SDT and PADT.

That is, the curves thus accord with the Pre and Post results for both the coordination and verb group transformations, where the former increases error propagation while the latter has no clear influence.

6.5.6 Related Work with Other Parsers

As in the previous chapter, we also want to present previous work investigating the influence of the coordination and verb group transformations on other parsing algorithms. This has been done in the study of Nilsson et al. (2007). The parsing experiments in this study use the non-projective Chu-Liu/Edmonds algorithm in MSTParser version 0.1, applying both the coordination and verb group transformations. The original training data of the PDT was used in the experiments, and evaluated on the dedicated evaluation test set (informally known as e-test). The reported results are in contrast to MaltParser negative, as the accuracy does not increase for the coordination and the verb group transformations. Whereas the verb group transformation has no influence on ASU at all, the coordination transformation decreases ASU by about 1 percentage point.

As noted in Nilsson (2007), a more detailed investigation is required to explain the outcome of the experiment, but one can speculate about the reason. A plausible explanation is that MaltParser’s one-pass incremental
6 Coordination and Verb Group Transformations

Figure 6.10: AS$_U$ clustering curves for Slovene (upper left), Arabic (upper right), Dutch (lower left) and Czech (lower right), where None = •, coordination transformations = △, and verb group transformation = ◦.

parsing approach makes it very aware of the partially constructed dependency structure to the left of the current parsing position, but has no knowledge of the dependency structure to the right. The chains going from left to right for both coordination and verb groups in MS therefore postpone crucial decisions. In coordination, for example, the words that are located to the right of the conjunction are often important for the words to the left, and vice versa. The coordination chains help to postpone such decisions. For MSTParser, on the other hand, the same type of problem does not exist. It has the ability to compare the scores of several dependency trees, which seems to make it less sensitive to whether coordination forms chains or not.

Moreover, the degradation in accuracy is less severe for longer dependencies for MSTParser compared to MaltParser (McDonald and Nivre, 2007). It entails that the lower average arc length – which according to subsection 6.5.4 seems important for MaltParser – is of less importance for MSTParser.

Even though the transformations for coordination and verb groups did not result in higher accuracy, it is still possible that there are other tree transformations that are beneficial for MSTParser but not for MaltParser. This, however, is a problem that has not been studied here.

6.6 Conclusion

The parsing results for coordination and verb group transformations generally show that parsing accuracy improves. Whereas the pseudo-projective
transformations have very little or no positive effect on SDT and PADT, significantly improved accuracies were recorded with the coordination transformations. This implies that the transformations are less dependent on the amount of training data compared to pseudo-projective parsing. The same conclusion holds for the verb group transformation, since it exhibits comparably higher accuracies for both SDT and PDT. The impact for the verb group transformation is less prominent than for the coordination transformation. For the original annotation in the investigated treebanks, the accuracy for words involved in verb groups is generally higher than for words involved in coordination. The room for improvement is therefore lower for the verb group transformation than for the coordination transformations.

As no transformation is flawless, it is likely that if one devotes more time to their optimization, improved accuracy can be expected, especially for coordination. One thing that could be of interest is to find the reason why $AS_U$ in most experiments increases more than $AS_L$. Is it an inherent property of the transformations, or is it simply the case that too little attention has been given to the quality of labels in the transformations? In any case, this is an issue that could need some more work to sort out.

As the transformations and inverse transformations inevitably introduce distortion in the data, these results may also be of interest for treebank creators. For instance, these issues were taken into consideration in the reconstruction of the Swedish treebank Talbanken (Einarsson, 1976a,b) into dependency structure (Nilsson et al. 2005; Nivre et al. 2006; Nilsson and Hall 2005). Without compromising linguistic correctness and expressiveness, parsing accuracy was one factor taken into consideration before the final dependency structure, for phenomena such as coordination and verb groups, was determined. Constructing new dependency treebanks, or constituency-based treebanks for that matter, in such a way that learning is facilitated is of interest, which consequently reduces the need for explicit tree transformations.
Chapter 7

Combining Classifiers

In chapters 5 and 6, we have investigated the impact of different tree transformations on parsing accuracy. In this chapter, we will start exploring the combination track, to see whether parsing accuracy can be improved through different types of system combination.

One observation from the CoNLL shared task 2007 (Nivre et al., 2007) is that some of the top-scoring systems in the multilingual track applied parser combination by reparsing (Sagae and Lavie, 2006). We will refer to this as combination on the graph level. For instance, the top-scoring system, Blended Malt (Hall et al., 2007), combines the output of a number of single transition-based dependency parsers implemented in MaltParser in this fashion. They report an improvement for all ten languages compared to the best single parser. Combining parsers seems to be especially beneficial for transition-based parsers, and one question here is whether the positive effect is due to the ability to counteract error propagation.

Transition-based parsing is essentially a simple sequence of classification tasks, where the dependency tree is built up by a sequence of parser transitions. This makes it possible to combine classifiers in two alternative ways – not just after the parse on the graph level – but also during parsing at each decision point (parser configuration). We will refer to the latter as combination on the transition level. In this chapter, we will compare classifiers on the transition level and on the graph level. In chapter 8, we will proceed to combinations of parsers that differ with respect to other dimensions than the classifiers used.

Section 7.1 describes the two combination approaches, and section 7.2 the experimental setup. The experiments are presented in section 7.3, followed by a conclusion in section 7.4.

7.1 Combination Methods

It has previously been shown by Sagae and Lavie (2006) that parsing accuracy can be improved by combining the output of different data-driven dependency parsers. The primary question addressed in this chapter is whether it is better to combine classifiers during parsing on the transition level (TL) or on the graph level (GL) using the approach of Sagae and Lavie (2006). Different classifiers have different properties and presumably
Combining Classifiers

different weaknesses and strengths. By combining them, their strengths can ideally be utilized, while compensating for their weaknesses.

At each decision point in a transition-based dependency parser with TL combination, a set of classifiers predicts the next transition, and a single transition is then chosen on the basis of this set of transitions. The selection strategy can, for instance, be based on machine learning. However, the strategy applied here is less complex, selecting a transition based on weighted and unweighted voting.

As mentioned in chapter 2, the GL parser combination method of Sagae and Lavie (2006) finds the optimal dependency structure given the set of weighted dependencies, using the Chu-Liu/Edmonds directed maximum spanning tree algorithm. The difference compared to MSTParser is that the weights are not learned using the treebanks directly but rather taken from the output of other dependency parsers. Whereas Sagae and Lavie in their study only handled unlabeled dependency structure, this will here be extended to labeled dependency structure.

The rest of this section will discuss the voting schemes applied in the experiment of this chapter. That is, how are the weights for the different classifiers on both TL and GL determined? The weighting formula will be similar to the one shown in section 2.6, but it will here be extended to labeled dependency structure. For GL, the score of the arc \( a \) labeled \( l \in R \), where \( R \) is the set of dependency labels, is

\[
s(a, l) = \sum_{i=1}^{m} w_i(a, l)_i
\]

where \( m \) is the number of parsers, \( w_i \) is the weight of the arc \( a \) with label \( l \) for parser \( i \), and \((a, l)_i = 1\) if \( a \) is an arc in the output of parser \( i \) with the label \( l \), and \((a, l)_i = 0\) otherwise. In contrast to the scoring function for unlabeled dependency structure, the dense graph containing all weighted arcs form a weighted and directed multigraph, with \( |R| \) arcs from one node to any other node.

The corresponding score for a transition \( t \) labeled \( l \) for TL is

\[
s(t, l) = \sum_{i=1}^{m} w_i(t, l)_i
\]

where each \( i \) instead represents a classifier, and \((t, l)_i = 1\) if classifier \( i \) proposes transition \( t \) with the label \( l \), and \((t, l)_i = 0\) otherwise. For transitions that are not accompanied by a label, e.g., Shift and Reduce for the arc-eager algorithm, \( l \) is a dummy label. Note that the same parsing algorithm must be used for all classifiers for TL, while no such restriction is required for GL.

Three different voting schemes are investigated:

- **Eq** weighs classifiers equally. For GL, each arc in each parser output is given the weight 1, i.e. all arcs have the weight \( w_i = 1 \). The weighting
strategy for TL is analogous, that is, each transition proposed by any classifier is given the weight 1.

- \textbf{Acc} weighs classifiers according to the $AS_L$ for each classifier. For GL, the weight $w_i$ of $a_i$ is $AS_L$, a real value between 0 and 1, computed using parsed held-out data. The weighting strategy for TL is again analogous. Each transition proposed by classifier $i$ is given the weight based on the average $AS_L$ on the held-out data for parser $i$.

- \textbf{TypeAcc} weighs classifiers according to the labeled precision of individual dependency labels. For GL, the weight $w_i$ of arc $a$ with label $l$ depends on the labeled precision that parser $i$ has on $l$, again computed using parsed held-out data. While Eq and Acc for GL were applied in Sagae and Lavie (2006), this is a new voting scheme for GL that has not been evaluated previously.

For simplicity and higher comparability to GL, the weights for TL also depend on the labeled precision of individual dependency labels. When classifier $i$ proposes a labeled left or right arc transition, the weight of this transition is based on the labeled precision of the transition’s label on the held-out data for parser $i$. All other transitions (i.e. shift and reduce for the arc-eager algorithm) are based on average $AS_L$ on the held-out data.\footnote{Another approach for computing the weights for TL would be to perform oracle parsing on the held-out data and make each classifier predict the next transition and compare it with the transition proposed by the oracle. The weight of each transition would then be directly based on the classification accuracy.}

The above voting schemes work fine for both labeled and unlabeled structure, but assigning a label to each arc in the combined dependency tree can be done in more than one way. Different labeling strategies were investigated in order to determine the labels in the combined dependency tree. The labeling strategy resulting in the highest $AS_L$ is described below.

Figure 7.1 exemplifies the labeling strategy, using six parsers. Given the voting scheme Eq, the scores of all labeled arcs with token 2 as the dependent are $s(0 \rightarrow 2, A) = 3$, $s(1 \rightarrow 2, A) = 2$ and $s(1 \rightarrow 2, B) = 1$. That is, the last labeled arc and its score are excluded. Only the labeled arc with the highest score for $1 \rightarrow 2$ will be used (i.e., $1 \xrightarrow{A} 2$ with its weight 2) despite the fact that three parsers voted for this arc. All labels except...
the one with the highest weight for each arc are therefore neglected, i.e.,
the weights for different labels of the same arc are not accumulated.\(^2\)

### 7.2 Experimental Setup

We will use data from four languages of the shared task 2007 (Czech, En-
glish, Greek and Italian), presented in chapter 3. Of the official training
data, 90\% (sets 1 . . . 9, divided as described in section 3.3) is used for train-
ing and the other 10\% (set 0) for automatically assessing the weights for all
coming experiments in this chapter. The three voting schemes mentioned
above are evaluated on the dedicated test sets for these four languages.

As in previous experiments, we will use MaltParser (version 1.0, hav-
ing only SVM built-in by default). In these experiments, both a quadratic
and a linear kernel will be used. In order to perform the experiments pre-

tended below, MaltParser has been augmented with two new classifiers us-
ing the plug-in mechanism of MaltParser, MaxEnt (McCallum, 2002) and
MBL (Daelemans and van den Bosch, 2005). The experiments are restricted
to the transition-based arc-eager algorithm, as it produced the highest ac-
curacies for these four languages compared to the arc-standard algorithm
during the development phase.\(^3\)

MaltParser has also been augmented with a mechanism for combining the
classifiers for TL, since this does not exist in the MaltParser 1.0 distribu-
tion. The three voting schemes for TL, presented in subsection 7.1, have
been implemented, which can be used for consulting multiple classifiers at
each decision point during the parse. The parser is therefore kept purely
deterministic by merging the various parser actions into one single decision,
when the classifiers disagree on the appropriate next parser action.

The Chu-Liu/Edmonds directed maximum spanning tree algorithm and
its voting schemes have, on the other hand, been implemented in a stand-
alone software.\(^4\)

### 7.3 Classifier Combination Experiments

The accuracies of the single parsers form the baseline. They are shown
for the four languages together in table 7.1 by concatenating the output of
all languages for each parser. SVM(Q) (where Q means quadratic kernel)
has similar settings as the Single MaltParser (Hall et al., 2007) in the 2007
shared task. There are however a number of factors making the figures
differ from previously published ones, such as only 90\% training data and
different versions of MaltParser. SVM(Q) has the highest \(A_{SL}\), and the

\(^2\) As an aside, preliminary experiments indicated that when the weights were accumu-
lated, slightly higher \(A_{SU}\) was recorded at the expense of slightly lower \(A_{SL}\).

\(^3\) Feature models and parameter values and other settings are listed at
http://w3.msi.vxu.se/users/jha/conll07/.

\(^4\) This implementation can be downloaded from: http://w3.msi.vxu.se/users/jni/blend/.
7.3 Classifier Combination Experiments

<table>
<thead>
<tr>
<th>SVM(Q)</th>
<th>SVM(L)</th>
<th>MaxEnt</th>
<th>MBL</th>
</tr>
</thead>
<tbody>
<tr>
<td>79.18</td>
<td>77.63</td>
<td>77.06</td>
<td>74.24</td>
</tr>
</tbody>
</table>

Table 7.1: AS$_L$ for single parsers for the four test data sets together.

differences to the others are statistically significant. However, the feature model optimized for SVM(Q) has for simplicity been used for all four classifiers. It is therefore worth noting that the optimal number of features for MBL/TiMBL is normally lower than for SVM. In other words, the figures can only be close to optimal for SVM(Q). In contrast to SVM and MaxEnt, having quite similar properties, most of the computation for MBL is performed during classification time.

7.3.1 Combining Classifiers

Table 7.2 reports AS$_L$ when combining the various classifiers of table 7.1 for TL and GL. The first row shows that combining all four classifiers on TL does not outperform the best single classifier SVM(Q), irrespectively of voting scheme. This is an interesting result, which may be attributed to the fact that the diversity of classifiers is low. SVM(L) (where L means linear kernel) is in different ways similar to both SVM(Q) and MaxEnt. This circumstance can to some extent explain why removing SVM(L) is beneficial and removing MBL hurts (or has virtually unaffected Acc) compared to using all four classifiers, as shown in the second and third rows.

The three last rows contain the corresponding figures for GL. The benefit from excluding SVM(L) is less than for TL. However, the most interesting observation is that GL results in an increased accuracy compared to SVM(Q). It confirms previous studies using the GL combination, e.g. Sagae and Lavie (2006), but the diversity is for the first time isolated to just machine learners. One can also see that the choice of voting scheme is more important for the GL combination than for TL, where TYPEAcc (i.e. the precision of individual dependency labels) turns out to be the best. Eq is especially inferior compared to TYPEAcc.

Removing SVM(L) is not as beneficial as for TL combination. It does, on the other hand, result in lower parsing time, since one parser less is needed. Removing MBL is harmful for both TL and GL combination, despite the

<table>
<thead>
<tr>
<th>TL</th>
<th>All 4</th>
<th>Eq</th>
<th>ACC</th>
<th>TYPEAcc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVM(L)</td>
<td>78.94</td>
<td>79.12</td>
<td>78.91</td>
</tr>
<tr>
<td></td>
<td>MBL</td>
<td>78.49</td>
<td>78.69</td>
<td>78.52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GL</th>
<th>All 4</th>
<th>Eq</th>
<th>ACC</th>
<th>TYPEAcc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVM(L)</td>
<td>79.15</td>
<td>79.88</td>
<td>80.07</td>
</tr>
<tr>
<td></td>
<td>MBL</td>
<td>78.91</td>
<td>79.23</td>
<td>79.43</td>
</tr>
</tbody>
</table>

Table 7.2: AS$_L$ results for the various voting schemes for TL and GL.
Combining Classifiers

<table>
<thead>
<tr>
<th></th>
<th>AS&lt;sub&gt;L&lt;/sub&gt;</th>
<th>Norm. Error Rate</th>
<th>Error Red.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pre Post</td>
<td>Pre Post</td>
<td></td>
</tr>
<tr>
<td>SVM(Q)</td>
<td>88.21 74.95</td>
<td>56.6 120.3</td>
<td>-0.1 -0.7</td>
</tr>
<tr>
<td>Best TL</td>
<td>88.13 74.88</td>
<td>56.8 120.3</td>
<td>-0.1 5.3</td>
</tr>
<tr>
<td>Best GL</td>
<td>88.20 76.28</td>
<td>59.7 119.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: Pre and Post for SVM(Q) and the best GL combination.

fact that MBL has the lowest AS<sub>L</sub>. Since MBL – being based on a lazy machine learning approach – is the most dissimilar of the four classifiers, the results are a clear indicator that diversity is more important than high accuracy when combining classifiers.

It is also worth noting that all differences for GL and ¬SVM(L) with TYPEACC (the lower bold figure in the table) to all other figures in table 7.2 are statistically significant (except GL and All 4 with TYPEACC).

### 7.3.2 Error Propagation

We will now study the error propagation for AS<sub>L</sub> for the best (boldfaced) systems in tables 7.1 and 7.2, SVM(Q), TL combination for ¬SVM(L) with ACC (denoted Best TL), and GL combination for ¬SVM(L) with TYPEACC (denoted Best GL). Pre and Post will be studied first, and the figures are reported in table 7.3. The result reveals that Pre is hardly affected at all for any of the systems compared to SVM(Q). Best TL and Best GL have lower Pre than SVM(Q), but optimization of all classifiers (not just SVM(Q)) can probably turn this around. However, a larger difference is recorded for Post, where SVM(Q) and Best TL have lower accuracies than Best GL.

In order words, the improvement for Best GL in relation to SVM(Q) takes place when at least one error has been committed. This is a clear indication that GL combination counteracts error propagation, whereas TL combination does not. The differences between Best GL compared to both SVM(Q) and Best TL for Post are also statistically significant, while none of the differences for Pre is.

For the sake of completeness, the AS<sub>L</sub> normalized error rates are also shown, supporting the same conclusion, that Best GL has the fewest errors in Post. The figures for SVM(Q) and Best TL are in fact close to identical for both Pre and Post. However, in contrast to table 4.1, it makes more sense to talk about error reduction here, comparing the two combination systems with the single parser. Since Post for Best GL is the only one with increasing accuracy, this is the only non-negative error reduction value. Its error reduction is 5.3%, compared to SVM(Q), while it has an error reduction of -0.1% for Pre.

A similar tendency can be discerned when studying the error propagation of the figures in table 7.4 for SVM(Q) and Best GL, where the upper half of the table shows the AS<sub>L</sub> error clustering and the lower half the corresponding normalized figures. In order to facilitate a comparison to the error
7.3 Classifier Combination Experiments

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>&gt;7</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM(Q)</td>
<td>31.9</td>
<td>23.1</td>
<td>19.1</td>
<td>17.4</td>
<td>16.9</td>
<td>16.7</td>
<td>11.5</td>
<td>12.3</td>
</tr>
<tr>
<td>Best GL</td>
<td>30.0</td>
<td>22.1</td>
<td>18.0</td>
<td>17.0</td>
<td>17.3</td>
<td>15.6</td>
<td>12.2</td>
<td>12.4</td>
</tr>
</tbody>
</table>

Normalized:

<table>
<thead>
<tr>
<th></th>
<th>153.0</th>
<th>111.0</th>
<th>91.6</th>
<th>83.4</th>
<th>81.2</th>
<th>80.4</th>
<th>55.0</th>
<th>59.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM(Q)</td>
<td>150.6</td>
<td>110.9</td>
<td>90.4</td>
<td>85.5</td>
<td>86.8</td>
<td>78.4</td>
<td>61.2</td>
<td>62.0</td>
</tr>
</tbody>
</table>

Table 7.4: $AS_L$ error clustering for SVM(Q) and Best GL combination.

clustering results in section 4.2, the focus is on the lower half. As Best TL clearly does not counteract error propagation, the comparison throughout the rest of this chapter is concentrated to just SVM(Q) and Best GL. The normalized figures are also depicted as a diagram in figure 7.2. Best GL in relation to SVM(Q) follows a behavior similar to that of MaltParser in relation to for instance LTH-parser (or any of the other parsers) in figure 4.1 (page 41). Best GL has lower normalized error rate for bins 1–3, and higher further away from the latest error (except for bin 6).

The difference between the curves of Best GL and SVM(Q) is less clear than between MaltParser and the other parsers in figure 4.1. But together with the result of table 7.3 the general conclusion is still that GL combination is a tool for dealing with error propagation.

7.3.3 Detailed Analysis

Can the improvement for the GL combination be attributed just to its ability to reduce the effects of error propagation? The relatively similar curves in figure 7.2 suggest that there are other explanations as well.

One property that distinguishes MSTParser from MaltParser is that the former guarantees that the output forms a tree with a single root. Transition-based parsing algorithms, on the other hand, can produce a fragmented output consisting of several subtrees. Even though MaltParser can trivially construct a single-rooted tree by attaching all subtrees to the virtual root – or allow a second parsing pass where only unattached tokens are processed (Hall et al., 2007) – it follows that the accuracy is affected negatively.

Table 7.5 contains accuracy figures for $AS_L$ for tokens attached to the virtual root and all other tokens not attached to the virtual root. The accuracy is consequently divided into precision and recall. When comparing SVM(Q) and Best GL, the pattern is similar for precision and recall accu-

<table>
<thead>
<tr>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Root</td>
</tr>
<tr>
<td>SVM(Q)</td>
<td>70.02</td>
</tr>
<tr>
<td>Best GL</td>
<td>76.24</td>
</tr>
</tbody>
</table>

Table 7.5: Precision and recall for root tokens and non-root tokens for SVM(Q) and the Best GL combination for $AS_L$. 

Combining Classifiers

Figure 7.2: Normalized $AS_L$ error clustering of table 7.4, where SVM(Q) = • and Best GL = ◦.

It is interesting to note that Best GL has a much higher precision and recall accuracy for tokens attached to the root, whereas the increases for precision and recall are smaller for non-root tokens. Especially the higher precision for tokens attached to the root can be tied to the fact that MaltParser produces fragmented parser output. GL combination guarantees – just like MSTParser – that the output forms a rooted tree. That is, GL combination turns a set of fragmented dependency graphs into a rooted tree.

The precision in table 7.5 deserves some more attention. One of the most interesting results in the comparison of MaltParser and MSTParser made by McDonald and Nivre (2007) is that precision increases for MaltParser the more deeply nested an arc is, whereas the situation is the opposite for MSTParser. They conclude that this is due to fragmented dependency graphs for MaltParser. It is therefore interesting to study how Best GL behaves in this perspective in relation to SVM(Q). These unnormalized curves are shown to the left in figure 7.3, where $x = 0$ represents the precision of tokens attached to the root (i.e. equivalent to the Root precision figures in table 7.5).

The curve for SVM(Q) has a very similar form compared to MaltParser in McDonald and Nivre (2007), having low precision closer to the root which then gradually increases. However, Best GL is flatter, with a higher precision closer to the root, and lower as the depth increases in relation to SVM(Q). In order words, Best GL again appears to behave like a hybrid between a transition-based and a graph-based parser. Having a high accuracy closer to the root is important, since the most important parts of a sentence (normally the predicate structure with objects and subjects) are located there.

Another interesting result in McDonald and Nivre (2007) is the reported
difference in precision between MaltParser and MSTParser for various dependency lengths. Both parsers exhibit decreasing accuracy as the length of the arc increases, but the degradation is more severe for MaltParser. The same metric applied to SVM(Q) and Best GL reveals that Best GL again borrows properties from graph-based parsers. The unnormalized curves for precision for arc lengths are shown to the right in figure 7.3. While SVM(Q) has a very small advantage for arcs shorter than 3, the most accurate parser for longer dependencies is Best GL. This behavior of Best GL is most likely explained by the fact that parser combination counteracts error propagation, which is especially harmful for longer dependencies (McDonald and Nivre, 2007).

7.4 Conclusion

Chapter 4 showed that mistakes lead to more mistakes for all parsers. However, mistakes are more severe for the closest tokens for MaltParser than for the others. The parsing experiments of section 7.3 showed that classifier combination is more beneficial for GL than TL. This can be partly attributed to the fact that the GL combination has an advantage over both the single parser and TL combination parser due to its ability to counteract error propagation. It can also be partly tied to the fact that the single parsers – as well as the TL combination parser – in MaltParser are not guaranteed to produce a tree. This is on the other hand guaranteed by the directed maximum spanning tree algorithm in the GL combination strategy. This together with reduced error propagation can account for why GL combination is especially beneficial for transition-based parsing algorithms. A GL combination of transition-based parsing algorithms, according to Sagae and Lavie (2006), consequently seems to combine the best of the transition-based and graph-based parsing strategies.
Chapter 8

Combining Transition-Based Parsers

The results of the previous chapter indicated that combining classifiers after parsing in transition-based dependency parsing is beneficial. This is the second chapter in the combination track, and takes up from the previous chapter. Instead of just combining parsers with different classifiers, transition-based parsers using different parsing algorithms and directions will be combined after parsing, in order to investigate whether this is even more beneficial. The combination of classifiers after parsing in the previous chapter will be referred to as classifier combination and combining parsing algorithms and parsing directions investigated in this chapter as algorithm combination.

The experiments of this chapter are essentially based on the two papers by Hall et al. (2007) and Hall et al. (2009), where the term Blended was used for the combination system now referred to as algorithm combination. The difference is that the results are studied in more depth, and with particular focus on Blended, which was briefly mentioned at the beginning of the previous chapter. Blended was the participating parser system with the highest overall score in the multilingual track of the CoNLL shared task 2007, where the task was to parse ten different treebanks (Nivre et al., 2007) as mentioned in section 3.3.

The chapter is structured as follows. Section 8.1 presents the single systems for the ten languages, referred to as Single Malt in Hall et al. (2007) and Hall et al. (2009),\(^1\) forming the basis of the combination system. The following section 8.2 presents the setup for the algorithm combination system and how it relates to the single systems. The experimental evaluation is presented in section 8.3, including an evaluation of error propagation. As usual, the chapter ends with a conclusion in section 8.4.

8.1 Single Systems

In contrast to chapter 7, where only the Czech, English, Greek and Italian data sets of the CoNLL shared task 2007 were used, all ten data sets will be used here. MaltParser is again the parser system used for the single systems. The parameters available in MaltParser can be divided into

\(^1\) Thanks to Gülsen Eryiğit, Johan Hall, Beáta Megyesi, Mattias Nilsson and Markus Saers for the optimization of the single systems.
three groups: parsing algorithm parameters, feature model parameters, and learning algorithm parameters, which were briefly described in section 3.4.\footnote{Complete specifications of all settings for all languages, for both the single systems and the algorithm combination, are available at http://w3.msi.vxu.se/users/jha/conll07/.
}

Note that the settings for the four languages in chapter 7 are the same as here. The evaluation presented for all languages in this chapter is on the dedicated test sets of the shared task data. As described in Hall et al. (2009), these are not the test sets that the settings were optimized on. During the development phase for the single systems, nine-fold cross-validation was used on 90\% (sets 1–9 according to section 3.3) of the training data for all languages with a training set size smaller than 300,000 tokens. For the remaining languages (Catalan, Chinese, Czech, English), 80\% (sets 1–8) was used for training and 10\% (set 9) for the development test. Labeled attachment score ($AS_L$) was used as the single optimization criterion. It is worth noting that the search for the best single systems was restricted to the two versions of the Nivre algorithm and their different parameters. For instance, the arc-standard algorithm improved parsing accuracy for Chinese compared to arc-eager, whereas the arc-eager algorithm turned out to be the best for all other languages. The single systems use the best classifier of the experiments in chapter 7, the LIBSVM implementation of SVM, as the learning method for all languages.

The columns labeled \textbf{Single} in table 8.2 (page 103) contain the results for the single systems, where sets 1–9 have been used for training. The average is 79.80 for $AS_L$ and 84.74 for $AS_U$ on the dedicated test sets (the official test sets) for all ten parsers, when concatenating the parser output before evaluation (as in chapter 7). The figures for the single systems form the baseline for the algorithm combination, which is in focus throughout the rest of the chapter.

\section*{8.2 Experimental Setup}

Just as in chapter 7, the algorithm combination system combines the output of several parsers. Here we weight the output based on the coarse parts-of-speech (CPOSTAG) for each parser, which conforms to Sagae and Lavie’s best weighting strategy. The algorithm combination system uses six component parsers, with two different parsing algorithms: the two variants of the Nivre algorithm (arc-eager and arc standard) and the non-projective version of the Covington algorithm, each of which is used to construct one left-to-right and one right-to-left parser. Thus, the six component parsers for each language are instances of the following:

1. Nivre arc-eager pseudo-projective left-to-right
2. Nivre arc-eager pseudo-projective right-to-left
3. Nivre arc-standard pseudo-projective left-to-right
4. Nivre arc-standard pseudo-projective right-to-left
5. Covington non-projective left-to-right

6. Covington non-projective right-to-left

For right-to-left parsing, a reverse transformation is applied on the training data before training. The reverse transformation is also applied on the test data before parsing. Since the parser then outputs the test data in the reverse order, yet another application of the reverse transformation is necessary to restore the original order. Before training and after parsing on the parser output, the head attributes are obviously updated accordingly. Note that the reverse transformation applies to tokens 1, ..., n. That is, the root token 0 remains the first token of the sentence. Figure 8.1 shows a dependency tree after the reverse transformation has been applied.

The final algorithm combination system is constructed by reusing the tuned parsers of the single systems for each language (arc-standard left-to-right for Chinese, arc-eager left-to-right for the remaining languages) and training five additional parsers with the same (or very similar) parameter settings, except for the following semi-mechanical adjustments:

1. Pseudo-projective parsing is not used for the two non-projective parsers, based on Covington’s non-projective algorithm.

2. The training data is split into smaller sets (see section 3.4) for some of the larger treebanks for the other five extrapolated single systems for each language. This speeds up training compared to not splitting the training data.

3. The training data is not split into smaller sets for some of the smaller treebanks (with split training data for the original single system). This makes it possible to increase accuracy compared to splitting the training data.

4. Having the feature model for the arc-eager version as the base, feature models are adjusted by adding and removing features defined on the dependency tree according to the constraints of different parsing algorithms. As mentioned in section 2.3, the arc-eager version has potential features that are always undefined for the arc-standard version, and vice versa. Moreover, the non-projective version of Covington’s
algorithm uses a context stack that the two others do not use, which provides more potential features.

See the settings page, referred to in section 8.1, about these semi-mechanical adjustments. In order to extrapolate the feature models of arc-eager and Covington for Chinese, where the arc-standard version formed the base, the adjustment list is basically applied in reverse for arc-eager and thereafter adapted to Covington.

Having trained all parsers on 90% of the training data for each language, the weights are determined by $AS_L$ for each distinct coarse part of speech for each parser on the remaining 10% of the data.

### 8.3 Algorithm Combination Experiments

Table 8.1 shows the outcome of the individual parsers for the three parsing algorithms for two parsing directions on the official test data sets. Note that these figures should be taken with a grain of salt due to the lack of proper tuning, but a few observations are nevertheless discernible.

#### 8.3.1 Parsing Direction and Parsing Algorithm

First, the row Average, where the output files have been concatenated, shows that arc-eager left-to-right has the highest overall accuracy. The differences to arc-standard and Covington are statistically significant ($p < 0.01$), partly explained by the fact that the optimization was performed for arc-eager for nine of the treebanks.

The overall accuracies for the parsers with the reverse transformation are all lower than for the left-to-right parsers. This should appeal to the psycholinguistic claim that parsing should be conducted in the way that
Table 8.2: The algorithm combination systems results compared to the single systems. Test results for the single systems (Single) and algorithm combination systems (AC) (with corrected test scores), where ER = error reduction for AC compared to Single. The scores are $AS_L$ are $AS_U$ on the dedicated test sets, where ' means correct results.

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>AS_L</th>
<th>AC</th>
<th>ER</th>
<th>Single</th>
<th>AS_U</th>
<th>AC</th>
<th>ER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ara</td>
<td>74.75</td>
<td>76.52</td>
<td>7.5</td>
<td>84.21</td>
<td>85.81</td>
<td>11.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bas</td>
<td>74.97</td>
<td>77.18</td>
<td>9.7</td>
<td>80.61</td>
<td>83.01</td>
<td>14.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cat</td>
<td>74.97</td>
<td>77.18</td>
<td>9.7</td>
<td>92.20</td>
<td>93.12</td>
<td>13.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chi</td>
<td>83.51</td>
<td>84.67</td>
<td>7.6</td>
<td>87.60</td>
<td>88.70</td>
<td>9.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cze</td>
<td>77.22</td>
<td>80.00</td>
<td>13.9</td>
<td>82.35</td>
<td>85.27</td>
<td>19.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eng</td>
<td>85.81</td>
<td>88.11</td>
<td>19.3</td>
<td>86.77</td>
<td>88.93</td>
<td>19.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gre</td>
<td>74.21</td>
<td>77.66</td>
<td>15.4</td>
<td>80.66</td>
<td>84.14</td>
<td>21.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hun</td>
<td>78.09</td>
<td>80.27</td>
<td>11.0</td>
<td>81.71</td>
<td>83.55</td>
<td>11.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ita</td>
<td>82.48</td>
<td>84.40</td>
<td>12.3</td>
<td>86.26</td>
<td>87.77</td>
<td>12.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tur</td>
<td>79.24</td>
<td>79.79</td>
<td>2.7</td>
<td>85.04</td>
<td>85.77</td>
<td>5.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average | 79.80 | 81.70 |10.4|84.74 | 86.51 |13.1|

humans parse sentences, incrementally from left to right (Marslen-Wilson, 1973; Frazier, 1987). All three differences between a left-to-right parser and its corresponding right-to-left parser are statistically significant ($p < 0.01$).

There are, however, three notable exceptions, Czech and Greek, which have the highest accuracies for the arc-standard right-to-left parsers, and Italian for right-to-left for Covington. The right-to-left parser for Covington is not statistically significantly better in comparison to arc-eager left-to-right for Italian. Nor is the arc-standard right-to-left parser for Greek compared to any of the three left-to-right parsers, but this might change as well with adequate optimization. The same holds for Czech.

A more proper comparison of the algorithms is in fact possible for the three right-to-left parsers, since none is properly optimized. We can see that the best parser under these circumstances is Covington followed by arc-standard, with arc-eager as the least accurate. This result is in line with Nivre (2008), reporting the same relationship between the parsers for unoptimized parsers on the CoNLL shared task data of 2006. The non-projective Covington parser (referred to as the non-projective list-based parser by Nivre, 2008) has slightly higher accuracy than arc-eager and arc-standard with pseudo-projective parsing, even though the differences are less than 0.15 percentage points. In this experiment for the right-to-left parsers, Covington is statistically significantly better than both the others ($p < 0.05$).


8.3.2 Combining Parsers

Table 8.2 shows the labeled attachment score results from the official test data for both of the single systems and algorithm combination systems. It should be pointed out that the test score for the algorithm combination system on Chinese is different from the official one (75.82), which was much lower than expected due to a corrupted specification file required by MaltParser. Restoring this file and rerunning the parser on the Chinese test set, without retraining the parser or changing any parameter settings, resulted in the score reported here. This also improved the average score from 80.32 to 81.20, the former being the highest reported official score. The latter figure is reported in Hall et al. (2009). However, the reverse arc-eager and arc-standard parsers for Basque, Czech and Greek suffered from another kind of corrupt specification files, yielding abnormally low accuracies. The figures in the AC columns contain the corrected results, where ' means that the result has been corrected. The average $A_{SL}$ reported here is 81.70, almost 1.4 percentage points above the official score.

We see that the algorithm combination systems outperform the single systems for all languages, with an average improvement for $A_{SL}$ of 1.90 percentage points, resulting in an average error reduction of 10.4%. Turkish exhibits the smallest improvement of +0.55 percentage points, having also the lowest error reduction of 2.7%. The highest improvement in percentage points is recorded for Greek (+4.7), while English has the highest error reduction of 19.3%. Even though the average improvement for $A_{SU}$ (+1.77) is lower than for $A_{SL}$, the error reduction is on the other hand higher (13.1%). This holds for every individual language as well (including Italian with one more decimal digit). That is, the maximum spanning tree algorithm is better suited for unlabeled than for labeled dependency trees. Each difference for an individual algorithm combination system for both $A_{SL}$ and $A_{SU}$ on the official test data in relation to its corresponding single system is statistically significant ($p < 0.05$ for $A_{SL}$ on Turkish, and $p < 0.01$ in all other cases). This holds for the algorithm combination systems compared to the single systems for the average $A_{SL}$ and $A_{SU}$ as well ($p < 0.01$).

Comparing the results for different languages, we see a tendency that languages with rich morphology, usually accompanied by flexible word order, get lower scores. Thus, the labeled attachment score is below 80% for Arabic, Basque, Czech, Greek, Hungarian, and Turkish. By comparison, the more configurational languages (Catalan, Chinese, English, and Italian) all have scores above 80%. Linguistic properties thus seem to be more important than, for example, training set size, which can be seen by comparing the results for Italian, with one of the smallest training sets, and Czech, with one of the largest. This is a property that holds not only for MaltParser, but for data-driven dependency parsers in general, as noted by Nivre et al. (2007). The development of parsing methods that are better suited for morphologically rich languages with flexible word order appears
as one of the most important goals for future research in this area.

The result for the algorithm combination systems can be compared to the classifier combination on the graph level in chapter 7. Restricted to the four treebanks used in the classifier combination experiment (Czech, English, Greek and Italian), the average $AS_L$ error reduction of 15.0% for the algorithm combination systems here, for only these four languages, is considerably higher than the $AS_L$ error reduction of 4.5% for the best classifier combination system (Best GL) compared to the best single parser SVM(Q). This clearly indicates that combining algorithms and directions is a better strategy than combining classifiers.

Before studying the results in more detail, it is worth discussing the additional time consumption for the parser combination systems compared to the single systems. Similarly to the transformations presented in the transformation track (i.e., the pseudo-projective, coordination and verb group transformations), the time complexity is worse than linear. Applying the Chu-Liu/Edmonds algorithm takes quadratic time. However, as mentioned in previous chapters, the parsing time in practice can take anything from minutes to hours with SVM for one parser, but performing the parser combination takes a few seconds on the same data set. Moreover, running $n$ parsers in sequence obviously takes $n$ times as long time, but note that the parsers are not dependent on each other, which makes it possible to run them in parallel. The additional time it takes for the algorithm combination systems compared to the single systems can thus be compressed considerably.

### 8.3.3 Error Propagation

As claimed by McDonald and Nivre (2007), MaltParser, when used as a single parser, tends to suffer from two problems: lower precision for long dependencies than short ones, and lower precision for dependencies originating in the artificial root node due to fragmented parses. Chapter 7 showed that these problems can to some extent be alleviated by the multiple views given by the component parsers in the classifier combination system. The focus here is on the single systems and algorithm combination systems averaged over all languages. The outcome of applying the Pre and Post metrics on these parsers is presented in table 8.3.

In contrast to the classifier combination on the graph level (cf. table 7.3, page 94), algorithm combination systems outperform the single systems for both Pre and Post, not only Post. Just as in table 7.3, it makes more sense to talk about error reduction for the algorithm combination systems compared to the single systems. This is shown in the last two columns, revealing an overall higher error reduction than for the classifier combination. These figures confirm the more positive effect after the first error (in Post) than before. The higher error reduction figures for $AS_U$ also confirm that the maximum spanning tree algorithm is better adapted to unlabeled dependency trees.
<table>
<thead>
<tr>
<th></th>
<th>AS_L</th>
<th>AS_U</th>
<th>Error reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Pre</td>
<td>Post</td>
</tr>
<tr>
<td>Single</td>
<td>79.80</td>
<td>88.05</td>
<td>75.33</td>
</tr>
<tr>
<td>Algorithm combination</td>
<td>81.70</td>
<td>88.67</td>
<td>77.72</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>Pre</td>
<td>Post</td>
</tr>
<tr>
<td>Single</td>
<td>84.61</td>
<td>91.71</td>
<td>79.48</td>
</tr>
<tr>
<td>Algorithm combination</td>
<td>86.50</td>
<td>92.27</td>
<td>81.96</td>
</tr>
</tbody>
</table>

Table 8.3: AS_L and AS_U for Pre and Post for the single systems and the algorithm combination systems, including error reduction figures.

This observation is strengthened by the error clustering curves in figure 8.2, the curve for the algorithm combination systems is at position 1 affected to a lower degree than the single systems, and to a higher degree after position 3, again indicating reduced error propagation. The difference in error propagation is even clearer than the corresponding curves for SVM(Q) and the classifier combination system (Best GL) in chapter 7, which is probably a result of the higher error reduction.

8.3.4 Detailed Analysis

Altering the parsing direction and parsing algorithm is thus more beneficial than combining classifiers for the same algorithm. The characteristics of parsing right-to-left are investigated here, in order to further assess its contribution to the results for the algorithm combination systems.

Table 8.4 shows the precision and recall for the three systems. It is noteworthy that Root-precision for the algorithm combination systems is higher than for the single systems, which is in line with the results in chapter 7, but the error reduction for Root-precision for the classifier combination in table 7.5 (22.3%, increasing from 70.02 to 76.72) is in fact higher than here (only 6.6%, increasing from 78.29 to 79.72).

However, higher Root-precision for the algorithm combination systems contradicts the figures reported by Hall et al. (2009) (i.e., with lower Root-precision for Blended). This can be explained by the use of corrupt specification files for some parsers of three languages (as explained above), which produced a huge number of fragmented parse trees for these languages, thus

<table>
<thead>
<tr>
<th></th>
<th>Precision (P)</th>
<th>Recall (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Root</td>
<td>Non-Root</td>
</tr>
<tr>
<td>Single</td>
<td>78.29</td>
<td>79.87</td>
</tr>
<tr>
<td>Reverse single</td>
<td>74.06</td>
<td>77.55</td>
</tr>
<tr>
<td>Algorithm combination</td>
<td>79.72</td>
<td>81.90</td>
</tr>
</tbody>
</table>

Table 8.4: Precision and recall for root tokens and non-root tokens for the single systems, reverse single systems and algorithm combination systems for AS_L.
8.3 Algorithm Combination Experiments

Figure 8.2: Normalized AS$_L$ error clustering for the single systems, the reverse single systems and the algorithm combination systems. The single systems = • and the algorithm combination systems = ○.

having a large impact on the overall Root-precision.

When comparing the single systems and the reverse single systems, the largest difference is for Root-precision, indicating a higher number of fragmented trees. Reverse single systems seem nevertheless to contribute. The diagram in figure 8.3 highlights the major strength of the reverse single systems and shows the consequence or error propagation. The diagram shows the AS$_L$ error rate in different locations in a sentence starting counting from the beginning of the sentence. All tokens at positions 1–5 are located in bin 5, tokens at positions 6–10 are located in bin 10, and so forth. The curves are normalized with respect to each parser overall AS$_L$ error rate.

The diagram shows that shorter sentences have a lower error rate (i.e., higher accuracy) for both parsing directions, which is consistent with previous parsing studies and agrees with common sense. Shorter sentences are, for instance, less ambiguous. Compared to left-to-right parsing, tokens closer to the end have a lower error rate for right-to-left parsing in relation to overall AS$_L$ error rate, and vice versa for tokens closer to the beginning of a sentence. This indicates why combining parsing directions is beneficial for the algorithm combination systems, which again is closely tied to error propagation. Where the single systems suffer from plenty of error propagation, the reverse single systems contribute with a dependency structure less contaminated by error propagation. The algorithm combination systems can probably be improved further by making the weighting strategy be influenced by the accuracy at various token positions.

Before concluding this chapter, a study follows of the three parsers in terms of accuracy at various arcs depths, as well as various dependency lengths, shown in figure 8.4. As in figure 7.3 (chapter 7), the curves are unnormalized.

In contrast to figure 7.3, we see that precision for all dependency lengths
improves across the board, especially for longer dependencies. There is a tendency that the error reduction for the algorithm combination systems in relation to the single systems is higher for longer dependencies (typically $> 15\%$ for arc lengths $\geq 7$) compared to shorter ones (typically $< 5\%$ for arc lengths $\leq 5$). This probably means that the effect of error propagation is mitigated by the use of an ensemble system, even if each of the component parsers is deterministic in itself. For instance, the situation is very similar to error reduction for the algorithm combination systems in relation to the reverse single systems. The algorithm combination systems have higher reduction for longer dependencies compared to shorter ones, where the reverse single systems also have the lowest accuracy for all dependency lengths. That is, higher precision for longer dependencies seems to be independent of which types of transition-based parsers are combined.

When studying the diagram to the right, the hybrid behavior identified

**Figure 8.3:** Normalized $A_{SL}$ error rate for grouped token positions. The single systems = $\bullet$, the reverse single systems = $\triangle$.

**Figure 8.4:** Precision for arcs at various depths (left) and dependency lengths (right) for $A_{SL}$, where the single systems = $\bullet$, the reverse single systems = $\triangle$, and the algorithm combination systems = $\circ$. X-value 0 = tokens attached to the virtual root in the left diagram.
8.4 Conclusions

This chapter has shown that combining the output of several dependency parsers is even more beneficial when combining parsing algorithms and parsing directions than when combining classifiers. In terms of error reduction, the results indicate that combining parsing algorithms is especially beneficial for unlabeled dependency structure. When comparing the single systems and the reverse single systems, the former systems are in most cases more accurate. However, the detailed analysis shows that right-to-left parsing is beneficial when combining, partly because the reverse single systems have fewer problems related to error propagation in the right half of the sentences than in the left.

The detailed analysis shows that the strength of the algorithm combination systems is mainly concentrated to deeply nested arcs and long dependencies. Finally, just as the classifier combination for chapter 7, the analysis indicates that error propagation is alleviated when combining transition-based parsing algorithms and parsing directions.
Chapter 9

Generic Tree Transformations and Combination

As previously discussed, data-driven syntactic parsing requires well-chosen syntactic representations in order to achieve state-of-the-art accuracy. This observation holds for both constituency-based and dependency-based parsing. The results in chapter 6 show that various tree transformations are beneficial for a dependency-based representation as well. Especially the coordination transformations are fairly complex. Moreover, the transformations described in chapter 6 were motivated on linguistic grounds. A question is whether the tree transformations have to be complex and linguistically motivated in order to improve accuracy.

Another observation is that combining the output of various parsers is beneficial, which again holds for both constituency-based and dependency-based parsing. For instance, Henderson and Brill (1999) report improved accuracy when combining constituency-based trees of various parsers with a simple majority voting strategy. Sagae and Lavie (2006) also report increased accuracy for both constituency structure and dependency structure for the Penn Treebank, and chapters 7 and 8 explored dependency parser combination further.

The content and results of this chapter are to a large extent based on Nilsson and Nivre (2008a). The aim is to bring the two tracks of the thesis – tree transformation and combination – together. A set of new tree transformations will be proposed here. In contrast to the previously proposed ones in chapter 6, the transformations here are both simpler and more general, and neither linguistically motivated nor tailored for any specific parsing algorithm. In general, we will investigate the result when these generic transformations are applied one at a time for a dependency parser in order to find the transformations that are the most beneficial. We will also investigate whether the dependency trees of parsers based on different transformations can be combined to improve accuracy and how this can be done to obtain the maximum improvement.

We begin this chapter by introducing the generic transformations in section 9.1, followed by the experimental setup in section 9.2. Thereafter, in section 9.3, parsing results for the general tree transformations for a wide range of languages are presented, where the transformations are evaluated
both individually and in combination. We end with a conclusion in section 9.4.

9.1 The Generic Transformations

The generic transformations presented in this section follow the same transformation methodology as the transformations presented in chapters 5 and 6, that is:

1. The tree transformation is applied to the training data.
2. A parser is trained on the transformed data.
3. New sentences are parsed.
4. The corresponding inverse transformation is applied to the output of the parser.

In contrast to the treebank dependent transformations in Nilsson et al. (2007) and chapter 6, the generic transformations are not constructed for a particular treebank annotation but are rather defined as general graph theoretic transformations. Four transformations and their corresponding inverse transformations are presented, which are called ChildSwap, ParentSwap, LiftChildren and DescendSiblings. The exact behavior of a transformation is less important for the systematic parsing experiments in section 9.3; the best transformations will nevertheless be found.

Each tree transformation starts in the same way by finding all tokens with a particular type of dependency to its parent. These tokens are called the focus tokens \( F \). The transformation is then applied to the focus tokens in a bottom-up and left-to-right order.

9.1.1 ChildSwap

In the ChildSwap transformation, the general idea is to make the focus token and one of its left children exchange parents. In case the focus token has two or more left children, this transformation is defined to always take the leftmost child, whereas the transformation is not applicable when left children are missing. The transformation is illustrated in figure 9.1. \( F \) is the focus token, \( P \) its original parent and \( C \) its original leftmost child, and the picture Pre CS shows the situation before the swap and Post CS after.

Also, swapped focus tokens are given a new unique dependency label \( (X^\ast) \), which distinguishes them from unswapped focus tokens. This will facilitate the inverse transformation.

9.1.2 ParentSwap

The ParentSwap transformation is essentially the same type of transformation as ChildSwap, with the difference that the focus token and its parent exchange parents instead. That is, the new parent of \( F \) is its former grandparent \( (G) \), while \( F \) becomes the new parent of \( P \). This is shown in
Pre PS and Post PS in figure 9.1. A distinction between swapped and unswapped focus tokens is not really necessary here, since it is only tokens with the root as parent that lack a grandparent.

Whereas the transformation is relatively simple, the inverse transformation is more complicated as any child of the focus token in Post PS could be the original parent. This is resolved by collecting a frequency list of all dependency labels of P (i.e., Y) in Pre PS during transformation. During the inverse transformation, the dependency type of the child with the highest frequency is then selected as the new parent of the focus token.

These pictures illustrate a potential problem with both swapping types, ChildSwap and ParentSwap. Depending on the linear order of the tokens involved in the swap, some arcs may introduce non-projectivity. In the situation in Pre PS, right siblings (S) of P lead to non-projectivity, and these will thus be given F as its new parent. Hence, both swapping types are designed not to introduce additional non-projectivity.

9.1.3 LiftChildren

Figure 9.2 depicts the LiftChildren transformation. It simply lifts all children of the focus word upward one step, making them into new siblings of the focus word. In order to facilitate the inverse transformation, the lifted tokens are distinguished from original siblings by augmenting their dependency types (Y’ and Z’).

9.1.4 DescendSiblings

The fourth and final transformation applied in this study is called DescendSiblings. If the arc of the focus token points to the left, its left siblings are turned into children of the focus token. In addition, the corresponding right siblings of right-pointing arcs are descended similarly during transformation. An example is shown in figure 9.2, where the arcs of \( S_2 \) are descended while the one for \( S_1 \) is left unchanged. Descended siblings are distinguished from original children by augmenting their dependency type (e.g. \( Z' \) for \( S_2 \)). The inverse transformation is thus fairly simple.

9.2 Experimental Setup

In the experiments, the ten treebanks of the CoNLL 2007 shared task (Nivre et al., 2007) will be used (Arabic, Basque, Catalan, Chinese, Czech, English,
Greek, Hungarian, Italian and Turkish). The official training data sets for all treebanks have been divided into three parts. The training set comprises 80% (sets 1–8 as described in chapter 3). The development test set \((D)\) used for parser selection throughout this chapter comprises 10% (set 9). The remaining 10% \((W)\) is reserved for estimating the arc weights of the parser combination. Similarly to the experiments in chapter 7, the weights are based on the labeled precision of the dependency types, which is a weighting strategy not applied by Sagae and Lavie (2006).

All labeled attachment score \((\text{AS}_L)\) figures presented below are based on the official test data sets of the shared task with \(\sim5000\) tokens per language. The ten parsed data sets for each parser have been concatenated into one file before evaluation.

MaltParser 1.0.4 was used for the experiments. The experiments have for simplicity employed very similar settings to the Single Malt system in chapter 8 and Hall et al. (2007). However, since a newer version of MaltParser is used, including only 80% of the training data, the figures presented here will differ somewhat.

## 9.3 Generic Transformation Experiments

The generic transformations introduced in section 9.1 will be empirically evaluated in this section. Their impact on the parsing result, individually as well as when they are combined using the parsing approach by Sagae and Lavie (2006), will be the focus of the coming experiments of subsections 9.3.1–9.3.3.

### 9.3.1 Generic Transformations

We will here investigate how the accuracy is affected by the generic transformations and their corresponding inverse transformations. That is, can some of the transformations – despite their simplicity – increase accuracy (1) individually and (2) when the output of the individual parsers using one single transformation each are combined?

As mentioned, four transformations are implemented, all having focus tokens chosen according to their dependency label. Theoretically, the number of transformations for a treebank is four times the number of distinct dependency labels. We assume that the least frequent labels are more unlikely to have an impact (positive or negative) on accuracy, so throughout
9.3 Generic Transformation Experiments

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Best</th>
<th>Base+9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inv</td>
<td>NoInv</td>
<td>Trans/Label</td>
</tr>
<tr>
<td>Arabic</td>
<td>73.7</td>
<td>75.9</td>
<td>72.9  CS/Coord</td>
</tr>
<tr>
<td>Basque</td>
<td>74.0</td>
<td>-</td>
<td>-     -</td>
</tr>
<tr>
<td>Catalan</td>
<td>87.3</td>
<td>87.0</td>
<td>85.3  PS/SUBORD</td>
</tr>
<tr>
<td>Chinese</td>
<td>83.5</td>
<td>83.6</td>
<td>80.4  DS/DUMMY2</td>
</tr>
<tr>
<td>Czech</td>
<td>75.6</td>
<td>76.1</td>
<td>74.8  DS/Pnom</td>
</tr>
<tr>
<td>English</td>
<td>86.3</td>
<td>86.3</td>
<td>82.9  CS/VC</td>
</tr>
<tr>
<td>Greek</td>
<td>73.2</td>
<td>73.9</td>
<td>67.6  DS/AuxP</td>
</tr>
<tr>
<td>Hungarian</td>
<td>77.5</td>
<td>77.4</td>
<td>77.3  CS/LOCY</td>
</tr>
<tr>
<td>Italian</td>
<td>82.5</td>
<td>82.5</td>
<td>74.5  DS/det</td>
</tr>
<tr>
<td>Turkish</td>
<td>78.9</td>
<td>78.8</td>
<td>75.2  CS/COORD.</td>
</tr>
<tr>
<td>Average</td>
<td>79.2</td>
<td>79.6</td>
<td>76.5  -</td>
</tr>
</tbody>
</table>

Table 9.1: Generic transformations results. Best = best single transformation result, Best Trans = best single transformation, NoInv = No inverse transformation for Best.

In this chapter we will only select transformations among the 16 most frequent labels for each treebank. Consequently, at most 65 \((4 \times 16 + 1)\) parsers with one transformation each will be considered, including the parser without any generic transformations. In this subsection, we will restrict the number of combined parsers to the parser without any generic transformations, plus the 9 parsers with transformations having the highest accuracy (on the data sets \(D\)).

Table 9.1 shows the result. The first column (Base) contains the figures without any generic transformations, which is the baseline. Column two (Best-Inv) presents the accuracies for the single best parsers. Selecting the best parser for the official test sets is not based on parsing the official test sets, but on both data sets \(D\) and \(W\), i.e. the remaining 20% of the official training data not used for training. The fourth row (Best-Trans/Label) shows which transformation was applied and for which dependency label, where a dash means that no transformation outperformed the baseline on the development set \((D+W)\).

The impact of the transformations varies much between treebanks. For instance, Arabic has a statistically significant \((p < 0.01)\) increased accuracy for the label Coord (+2.2 percentage points). This is an interesting observation, since this label is involved in coordination, also targeted by the complex coordination transformations in chapter 6. The figures are not exactly comparable, as different training and testing sets have been used, but it is still worth noting that the parsing experiments in chapter 6 report an increased accuracy of +1.8 percentage points for the coordination transformation on Arabic. As already mentioned, the difference is that the generic transformations presented here are not constructed for a specific type of annotation. Greek, too, which is also based on the Prague school, exhibits increased accuracy for some transformations involving coordination (e.g. Pred_Co for PARENTSWAP), but they are all slightly less prominent
than the best Greek transformation.

For many other treebanks the baseline is at least as accurate as **Best** on the test set. Nevertheless, even though the increases for **Best** are not statistically significant for Czech (+0.5 percentage points) and Greek (+0.7 percentage points) (partly due to the small test sets), the results give at least a small indication that these and other generic transformations could be beneficial.

The fifth column (**Base+9-Inv**) aims to answer question (2) at the beginning of this section. It shows the accuracy when the baseline parser and the 9 best parsers with the highest accuracy (selected using the data sets $$D$$) for each treebank are combined. In comparison to **Base**, the average accuracy increases, and the difference is statistically significant ($$p < 0.01$$). Three of the individual languages also have statistically significant differences ($$p < 0.01$$) for **Base+9** compared to **Base**, namely Arabic, Czech and Greek. The increase for the average is lower than in chapters 7 and 8, but confirms that combining various parsers is beneficial. It is shown for the first time that simple and generic transformations are beneficial while keeping all other settings unchanged. However, the situation is not as clear when determining its relationship to **Best-Inv**. Averaging over all languages, **Best-Inv** has the same accuracy as **Base+9-Inv**.

To summarize, treebank dependent and linguistically motivated transformations are not necessary in order to improve accuracy. Simple general transformations for a few languages are applicable with increased accuracy. Of course, the vast majority of these general transformations for each language decrease accuracy. However, by applying them systematically like above, we are able to find the transformations that are beneficial. Finally, in comparison to the combination strategy in **Base+9-Inv**, simply selecting the single best parser is often at least as good.

### 9.3.2 No Inverse Transformation

All generic transformations in subsection 9.3.1 combine pre-processing and post-processing according to the description at the beginning of section 9.1. In this subsection we will investigate the importance of the fourth step, the inverse transformation on the parser output. More precisely, how is accuracy affected when the output of the various parsers is not subjected to an inverse transformation?

The third column of table 9.1 (**Best-NoInv**) contains the accuracies of the single best parsers for **Best-Inv** without performing the inverse transformation. These figures are consistently lower than **Best**, which clearly indicates the importance of the inverse transformation for the single parsers with transformations. When comparing these figures with **Base**, some noticeable observations can be seen, such as the substantial drops for Greek ($$-5.6$$) and Italian ($$-8.0$$). The inverse transformations for these two and the other treebanks are able to restore the accuracy, and for some treebanks even surpass **Base**.
When combining parsers without inverse transformations, one could expect the same behavior. However, this is actually not the case. The last row in table 9.1 contains the average accuracies, showing that \textbf{Base+9-Inv} outperforms \textbf{Base}. The most interesting result is when the baseline parser is combined with the 9 highest scoring parsers (again selected using the data sets \(D\)) without performing inverse transformations. It is therefore plausible that the 9 selected parsers are not the same for these two combination strategies. This type of combination system is called \textbf{Base+9-NoInv} (last column). \textbf{Base+9-Inv} has slightly higher accuracy, but when rounded to one decimal, \(AS_L\) for \textbf{Base+9-NoInv} is 79.6\%, the same as both \textbf{Base+9-Inv} and \textbf{Best-Inv}.

The result may at a first glance seem somewhat surprising, but it can be explained by the adopted weighting strategy for the parser combination. The weights are estimated on the basis of the precision of individual dependency labels. A transformation modifies the structure close to a certain dependency label. It is hence likely that the transformation yields a low weight to the dependency labels that are often involved in this transformation.

This, in turn, results in a low impact for these dependency labels during the parser combination. However, the remaining dependency structure is kept relatively intact, and is parsed differently compared to other parsers because of the transformation. This is beneficial irrespectively of whether an inverse transformation has been applied or not. In other words, the parser diversity is consequently a very important property, further investigated below.

\textbf{9.3.3 Selection Strategy}

The results presented in subsection 9.3.2 indicate that the need for post-processing is important for the single parsers, while unnecessary when combining the parsers. However, it is still unsatisfactory that \textbf{Best-Inv} and \textbf{Base+9-Inv} have virtually the same average accuracy in subsection 9.3.1. We will here investigate whether the selection based on high accuracy, applied in 9.3.1, is the best strategy. In other words, can the moderate improvement in accuracy for \textbf{Base+9-Inv} in comparison with \textbf{Base} (+0.4) be a result of suboptimal parser selection? Before we turn our attention to this issue, we will study whether ten is an optimal number of parsers to combine for \textbf{Base+9-Inv} and \textbf{Base+9-NoInv}.

In figure 9.3, \textbf{Base+9-Inv} corresponds to the curve denoted \([\text{Inv}/\text{Desc}]\), which means that the selection is based on the output for data set \(D\) with the inverse transformation, which is sorted in descending order, and where the parser combination uses the output with the inverse transformation. \textbf{Base+9-NoInv} then corresponds to the curve \([\text{NoInv}/\text{Desc}]\), meaning that the selection is based on the output for data set \(D\) without the inverse transformation, which is sorted in descending order, and where the parser combination uses the output without the inverse transformation.
Both these curves in figure 9.3 have very similar behavior, climbing in the beginning up to about 10 parsers. Thereafter, as more parsers with lower accuracies are added, no further real increase or decrease is recorded. So no more than 8–10 are in practice needed in order to reach the upper limit. The difference between the curves is negligible, which indicates that the additional complexity that the inverse transformations impose is not really worth the effort.

The diagram contains two more curves, and we will begin by looking at [NoInv/Asc]. Just as for [NoInv/Desc], the inverse transformations are not used at all, but the parsers are instead added in ascending order. That is, the baseline parser is first combined with the parser having the lowest accuracy without an inverse transformation, and then with the parser having the second lowest accuracy, and so on. This curve has a completely different appearance, as it has a much clearer peak at 9 parsers. More interestingly, the peak is located well above the two previous curves, despite (1) the unintuitive order in which new parsers are added, and (2) there being no inverse transformations. This result contrasts with the assumption that the best parsers should be combined first, in descending order. The form of the fourth curve in the diagram, [Inv/Asc], is very similar to [NoInv/Asc] with an early peak, and decreasing accuracy as more parsers are added. The curve [Inv/Asc] then uses the inverse transformations and the parsers are added in ascending order.

The main conclusion in this subsection is that we can in fact improve accuracy more by selecting the parsers in a way that does not seem intuitive.
The fact that the curves behave almost identically strengthens the statement that parser selection is more important than high accuracy.

For the parser combination experiments, the main goal of this chapter has been to further investigate the properties of how to select parsers from a set of parsers, not primarily to achieve as high accuracy as possible. It is therefore not surprising that the results do not reach the same level as the algorithm combination in chapter 8 using 6 parsers (three parsing algorithms and two parsing directions), which reported an improved accuracy of +1.9 compared to their baseline, one single parser. The improvement for [NoInv/Asc] and [Inv/Asc] compared to the baseline parser of this chapter is only +0.8 (from 79.2% to 80.0%).

9.4 Conclusion

We have shown that generic transformations not motivated linguistically and not tailored for a specific parsing algorithm can improve accuracy for several treebanks. We have presented four, and it is certainly possible to construct other types of generic transformations that can also improve accuracy.

The generic transformations were also combined using the approach presented by Sagae and Lavie (2006), showing that they can be combined with an increased accuracy. We have also shown that the order in which various parsers are combined is of major influence. The results indicate that the most apparent combination strategy – combining the parsers with the highest accuracy – is not the best way. Instead, a less intuitive strategy is better: use one well-performing parser as the base, and then add the other parsers in ascending order, starting with the worst parser. This indicates that parser diversity is more important than high accuracy. Moreover, the results indicate that – instead of using as many as possible – only a small number of parsers are needed in order to obtain the highest accuracy.
Chapter 10

Conclusion

The main results of the two tracks of this Ph.D. thesis – the combination and transformation tracks – are summarized in this final chapter. The main contributions are presented in the first section, which is followed by some proposals for future work.

10.1 Contributions

To summarize the transformation track, one can claim that preprocessing and postprocessing are important not only in constituent-based parsing but also for data-driven dependency parsing. As stated by Johnson (1998), choosing the right base representation is an important task in constituency-based parsing, which his and many subsequent studies confirm. This is an assertion that can be generalized to data-driven dependency-based parsing as well. Chapter 5 presented the pseudo-projective transformations and parsing results dealing with non-projectivity, and chapter 6 the tree transformations and parsing results for coordination and verb groups.

The pseudo-projective transformation is shown to be the most generally applicable, as improvement in accuracy holds for several languages, treebanks and parsers. The pseudo-projective transformation is defined in a way that does not assume a certain treebank annotation, except that the dependency structure forms a labeled and rooted tree. It also seems that pseudo-projective transformation has been the most influential contribution to the community. After being introduced in Nivre and Nilsson (2005), it has been used in a large number of systems, including Johansson and Nugues (2006); Chang et al. (2007); Duan et al. (2007); Wu et al. (2007); Nivre et al. (2008); Titov and Henderson (2007b); Sagae and Tsujii (2007, 2008); Johansson and Nugues (2008); Bharati et al. (2008).

The tree transformations for coordination and verb groups have been less influential. However, Johansson (2008) opted for an asymmetric analysis for coordination (i.e., not according to the Prague school) when converting the Penn Treebank to a dependency structure, due to the higher accuracy that some parsers (e.g., MaltParser) obtain. Moreover, the coordination and verb group transformations are dependent on the dependency annotation, and sensitive to the parsing algorithm, but improve accuracy across languages and treebanks for MaltParser.
Turning to the combination track, chapter 7 compared two approaches for combining machine learners in transition-based dependency parsing, on the transition level during parsing and on the graph level after parsing. The results are unambiguously in favor of combining on the graph level. The analysis indicates that this is explained by the fact that the output of the graph level combination (Sagae and Lavie, 2006) forms a tree of the parse fragments, as the accuracy for tokens attached to the root increases substantially. Another largely contributing factor is that graph level combination does not suffer from error propagation to the same extent as transition-level combination.

The results of chapter 7 are strengthened by the findings in chapter 8, where various transition-based parsers – including both forward and backward parsing – are combined on the graph-level. Combining transition-based parsers is more beneficial than combining machine learners, probably due to the fact that the error propagation that each individual transition-based parser suffers from is alleviated. Transition-based parsers have been shown to perform at state-of-the-art level individually, and when combined, they achieved the best results in the CoNLL shared task 2007.

When constructing and applying in chapter 9 a large set of tree transformations that are not language- and treebank-dependent, a small set of beneficial tree transformations can be detected. The positive effect is not as profound as the specialized tree transformations for coordination and verb groups in chapter 6, which indicates that linguistic knowledge is often difficult to convey to a computer entirely. However, chapter 9 contributes with experimental results showing that it is not necessarily optimal to combine all parsers available, and that the parsers to combine need to be selected with care.

The error propagation metrics, presented in chapter 4, have been applied throughout this thesis. Assuming that the metrics do in fact capture the effects of error propagation, it is interesting to note that the analysis shows that at least the coordination transformation exhibits problems with increasing error propagation. This is in contrast to the graph-level combination, which decreases error propagation. Consequently, both graph-level combination and the coordination transformations improve accuracy for MaltParser, but apparently for different reasons. However, it should be remembered that the error propagation metrics remain to be validated.

To conclude, this thesis – together with several other studies – shows that accurate dependency parsing can be both efficient and robust using transition-based data-driven parsing techniques. Efficient parsing is very important when using syntactic parsing as a component in real-world applications. Even though the parsing techniques utilized in this thesis are highly accurate, high parsing speed is often at least as important in practice, a fact that is often neglected in the research community. All the approaches proposed and investigated in the two tracks both maintain efficient parsing and increase accuracy, which conforms with the overall goal of this thesis. The time it takes to apply any of the inverse transformations increases the
10.2 Future Work

Even though this thesis is completed, this does not mean that there is no room for further research on this topic. Here is a list of possible research tracks to explore:

- The classifier combination and parser combination experiments in chapters 7 and 8 showed that transition-based parsers have error propagation problems, and chapter 4 indicated that error propagation can to a large extent be alleviated by using beam search. Some examples in this direction are Johansson and Nugues (2006), Titov and Henderson (2007a), and Zhang and Clark (2008), and further research in this area can prove useful.

- The results for the pseudo-projective parsing experiment can be investigated in more detail by looking at the differences between projective parsing with pseudo-projective transformations and direct non-projective parsing. Which strategy is most sensitive to the amount of training data and the proportion of non-projectivity? Which one performs best for different types of non-projective arcs? Why does it seem that $AS_L$ is higher for direct non-projective parsing but not for $AS_U$?

- The experiments for the pseudo-projective transformations do not indicate any tangible differences in accuracy between the different encodings. Other encodings may, however, be even more beneficial for pseudo-projective parsing, while maintaining the black-box approach. In particular, we could investigate whether preprocessing in combination with machine learning can improve the accuracy of pseudo-projective parsing, while maintaining the black-box principle. This kind of comparison can be found in Nivre (2008) and Nivre (2009).

- Another possibility is to incorporate the decision of whether an arc is lifted or not as a separate classifying task inside the parser. The
classification is then presumably done after the arc has been assigned a dependency type. This is in contrast to the current approach, where the information about the lift is part of the dependency type of the lifted arc and/or the arcs along the lifting path, and consequently classified together. As a result, the set of dependency types does not need to be extended, which may be beneficial in order to avoid sparse data problems. However, the disadvantage is that the black-box approach is lost, making it more parser-dependent.

- Another possible research area that touches upon the work of Petrov et al. (2006) is to find the right level of granularity in the dependency labels in order to achieve the highest $\text{AS}_U$. One can expect that $\text{AS}_L$ will usually increase as the number of distinct dependency types decreases, but the question is whether the same holds for $\text{AS}_U$ and whether this can be done in an automatized manner. A treebank with fine-grained dependency types would serve as good material for such an experiment.

This list of suggestions for future work just includes some possibilities, and other research tracks within this topic may evolve eventually. Another interesting track for future research, going beyond natural languages into formal languages, is to further develop the technique presented by Nilsson et al. (2009a) and Nilsson et al. (2009b). As noted in these studies, program analysis tools must be robust and ought to be accurate for software maintenance tasks, such as program comprehension (understanding unknown code for fixing bugs or further development), quality assessment (judging code, e.g., in code reviews), and reverse-engineering (reifying the design documents for given source code). The robustness of MaltParser makes it applicable in circumstances where non-robust techniques known from the field of compiler construction may not be used in order to construct formal models (e.g., abstract syntax trees), which form the input of existing software maintenance tools. One situation requiring robustness is when analyzing dialects of C that lack tools for producing formal models. MaltParser trained on, for instance, standard C converted to dependency structure can then be applied. The experimental results show that MaltParser achieves higher accuracies for programming languages like Java, C/C++ and Python than for natural languages.

Finally, the robustness and the potentially high parsing speed of MaltParser makes it a possible component in various NLP systems as well. A more general and interesting goal would therefore be to incorporate MaltParser in systems for machine translation, question answering and information extraction, aiming at improving the quality of these systems. This thesis could be a small contribution in this direction.


Einarsson, J. (1976a). Talbankens Skrifter kskonkordans. Lund University, Department of Scandinavian Languages.

Einarsson, J. (1976b). Talbankens Talspråksskonkordans. Lund University, Department of Scandinavian Languages.


Acta Wexionensia

Below please find a listing of publications in the Acta Wexionensia series. For more information, please see www.vxu.se

Series III (ISSN 1404-4307). From 2006 and onward.


---

Växjö University Press
S-351 95 Växjö
www.vxu.se
vup@vxu.se